Stateful Protocol Composition and Typing

Andreas V. Hess* Sebastian Mödersheim* Achim D. Brucker†

May 20, 2020

*DTU Compute, Technical University of Denmark, Lyngby, Denmark
{avhe, samo}@dtu.dk

† Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk

Abstract

We provide in this AFP entry several relative soundness results for security protocols. In particular, we prove typing and compositionality results for stateful protocols (i.e., protocols with mutable state that may span several sessions), and that focuses on reachability properties. Such results are useful to simplify protocol verification by reducing it to a simpler problem: Typing results give conditions under which it is safe to verify a protocol in a typed model where only "well-typed" attacks can occur whereas compositionality results allow us to verify a composed protocol by only verifying the component protocols in isolation. The conditions on the protocols under which the results hold are furthermore syntactic in nature allowing for full automation. The foundation presented here is used in another entry to provide fully automated and formalized security proofs of stateful protocols.

Keywords: Security protocols, stateful protocols, relative soundness results, proof assistants, Isabelle/HOL, compositionality

Contents

1 Introduction

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. The formalization presented in this entry is described in more detail in several publications:

- The typing result (section 3.4 "Typing_Result") for stateless protocols, the TLS formalization (section 7.1 "Example_TLS"), and the theories depending on those (see Figure 1.1) are described in [2] and [1, chapter 3].
- The typing result for stateful protocols (section 4.2 "Stateful_Typing") and the keyserver example (section 7.2 "Example_Keyserver") are described in [3] and [1, chapter 4].
- The results on parallel composition for stateless protocols (section 5.2 "Parallel_Compositionality") and stateful protocols (section 6.2 "Stateful_Compositionality") are described in [4] and [1, chapter 5].

Overall, the structure of this document follows the theory dependencies (see Figure 1.1): we start with introducing the technical preliminaries of our formalization (chapter 2). Next, we introduce the typing results in chapter 3 and chapter 4. We introduce our compositionality results in chapter 5 and chapter 6. Finally, we present two example protocols chapter 7.

Acknowledgments This work was supported by the Sapere-Aude project "Composec: Secure Composition of Distributed Systems", grant 4184-00334B of the Danish Council for Independent Research.

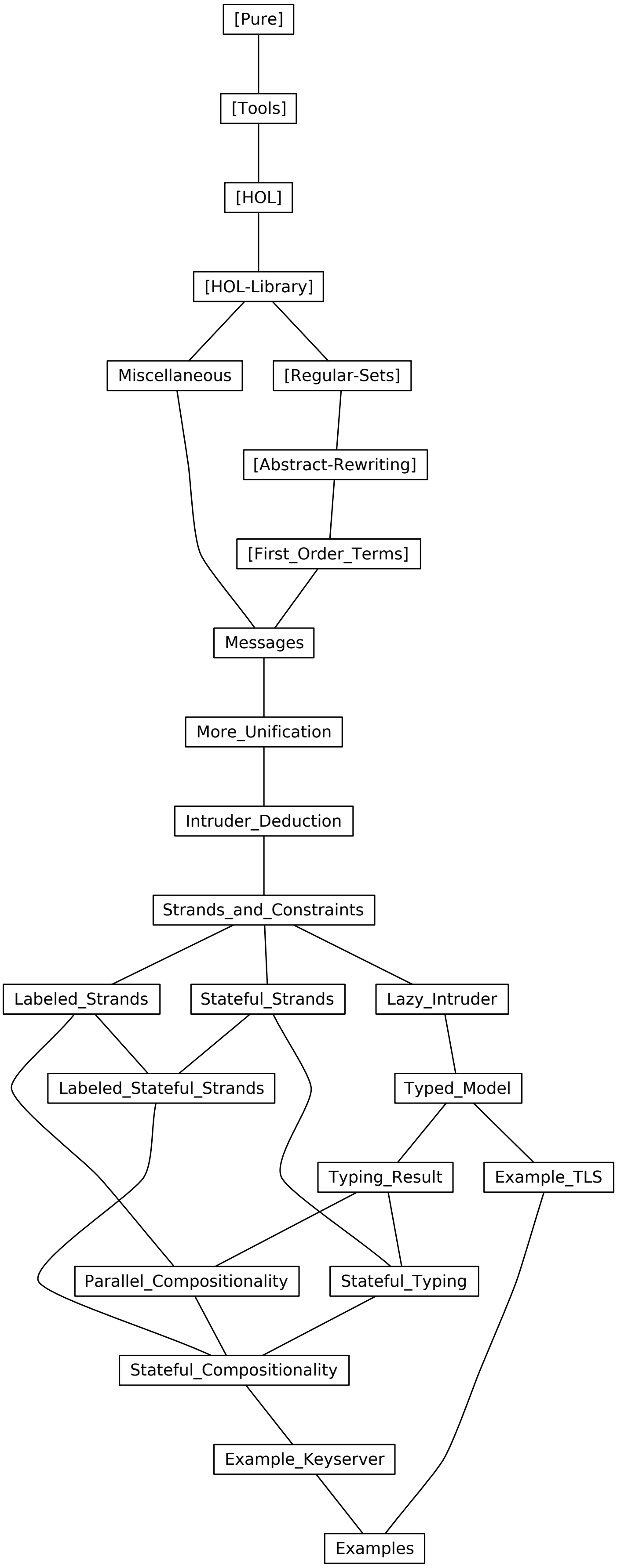

Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 Preliminaries and Intruder Model

In this chapter, we introduce the formal preliminaries, including the intruder model and related lemmata.

2.1 Miscellaneous Lemmata (Miscellaneous)

```
theory Miscellaneous
imports Main "HOL-Library.Sublist" "HOL-Library.While_Combinator"
begin
```

2.1.1 List: zip, filter, map

```
lemma zip_arg_subterm_split:
  assumes "(x,y) ∈ set (zip xs ys)"
  obtains xs' xs'' ys' ys'' where "xs = xs'@x#xs''" "ys = ys'@y#ys''" "length xs' = length ys'"
proof -
  from assms have "∃zs zs' vs vs'. xs = zs@x#zs' ∧ ys = vs@y#vs' ∧ length zs = length vs"
  proof (induction ys arbitrary: xs)
    case (Cons y' ys' xs)
    then obtain x' xs' where x': "xs = x'#xs'"
      by (metis empty_iff list.exhaust list.set(1) set_zip_leftD)
    show ?case
      by (cases "(x, y) ∈ set (zip xs' ys')",
          metis ‹xs = x'#xs'› Cons.IH[of xs'] Cons_eq_appendI list.size(4),
          use Cons.prems x' in fastforce)
  qed simp
  thus ?thesis using that by blast
qed

lemma zip_arg_index:
  assumes "(x,y) ∈ set (zip xs ys)"
  obtains i where "xs ! i = x" "ys ! i = y" "i < length xs" "i < length ys"
proof -
  obtain xs1 xs2 ys1 ys2 where "xs = xs1@x#xs2" "ys = ys1@y#ys2" "length xs1 = length ys1"
    using zip_arg_subterm_split[OF assms] by moura
  thus ?thesis using nth_append_length[of xs1 x xs2] nth_append_length[of ys1 y ys2] that by simp
qed

lemma filter_nth: "i < length (filter P xs) ⟹ P (filter P xs ! i)"
using nth_mem by force

lemma list_all_filter_eq: "list_all P xs ⟹ filter P xs = xs"
by (metis list_all_iff filter_True)

lemma list_all_filter_nil:
  assumes "list_all P xs"
    and "⋀x. P x ⟹ ¬Q x"
  shows "filter Q xs = []"
using assms by (induct xs) simp_all

lemma list_all_concat: "list_all (list_all f) P ⟷ list_all f (concat P)"
by (induct P) auto

lemma map_upt_index_eq:
  assumes "j < length xs"
  shows "(map (λi. xs ! is i) [0..<length xs]) ! j = xs ! is j"
using assms by (simp add: map_nth)
```

```
lemma map_snd_list_insert_distrib:
  assumes "∀(i,p) ∈ insert x (set xs). ∀(i',p') ∈ insert x (set xs). p = p' ⟶ i = i'"
  shows "map snd (List.insert x xs) = List.insert (snd x) (map snd xs)"
using assms
proof (induction xs rule: List.rev_induct)
  case (snoc y xs)
  hence IH: "map snd (List.insert x xs) = List.insert (snd x) (map snd xs)" by fastforce

  obtain iy py where y: "y = (iy,py)" by (metis surj_pair)
  obtain ix px where x: "x = (ix,px)" by (metis surj_pair)

  have "(ix,px) ∈ insert x (set (y#xs))" "(iy,py) ∈ insert x (set (y#xs))" using y x by auto
  hence *: "iy = ix" when "py = px" using that snoc.prems by auto

  show ?case
  proof (cases "px = py")
    case True
    hence "y = x" using * y x by auto
    thus ?thesis using IH by simp
  next
    case False
    hence "y ≠ x" using y x by simp
    have "List.insert x (xs@[y]) = (List.insert x xs)@[y]"
    proof -
      have 1: "insert y (set xs) = set (xs@[y])" by simp
      have 2: "x ∉ insert y (set xs) ∨ x ∈ set xs" using ‹y ≠ x› by blast
      show ?thesis using 1 2 by (metis (no_types) List.insert_def append_Cons insertCI)
    qed
    thus ?thesis using IH y x False by (auto simp add: List.insert_def)
  qed
qed simp

lemma map_append_inv: "map f xs = ys@zs ⟹ ∃vs ws. xs = vs@ws ∧ map f vs = ys ∧ map f ws = zs"
proof (induction xs arbitrary: ys zs)
  case (Cons x xs')
  note prems = Cons.prems
  note IH = Cons.IH

  show ?case
  proof (cases ys)
    case (Cons y ys')
    then obtain vs' ws where *: "xs' = vs'@ws" "map f vs' = ys'" "map f ws = zs"
      using prems IH[of ys' zs] by auto
    hence "x#xs' = (x#vs')@ws" "map f (x#vs') = y#ys'" using Cons prems by force+
    thus ?thesis by (metis Cons *(3))
  qed (use prems in simp)
qed simp
```

2.1.2 List: subsequences

```
lemma subseqs_set_subset:
  assumes "ys ∈ set (subseqs xs)"
  shows "set ys ⊆ set xs"
using assms subseqs_powset[of xs] by auto

lemma subset_sublist_exists:
  "ys ⊆ set xs ⟹ ∃zs. set zs = ys ∧ zs ∈ set (subseqs xs)"
proof (induction xs arbitrary: ys)
  case Cons thus ?case by (metis (no_types, lifting) Pow_iff imageE subseqs_powset)
qed simp

lemma map_subseqs: "map (map f) (subseqs xs) = subseqs (map f xs)"
```

```
proof (induct xs)
  case (Cons x xs)
  have "map (Cons (f x)) (map (map f) (subseqs xs)) = map (map f) (map (Cons x) (subseqs xs))"
    by (induct "subseqs xs") auto
  thus ?case by (simp add: Let_def Cons)
qed simp

lemma subseqs_Cons:
  assumes "ys ∈ set (subseqs xs)"
  shows "ys ∈ set (subseqs (x#xs))"
by (metis assms Un_iff set_append subseqs.simps(2))

lemma subseqs_subset:
  assumes "ys ∈ set (subseqs xs)"
  shows "set ys ⊆ set xs"
using assms by (metis Pow_iff image_eqI subseqs_powset)
```

2.1.3 List: prefixes, suffixes

```
lemma suffix_Cons': "suffix [x] (y#ys) ⟹ suffix [x] ys ∨ (y = x ∧ ys = [])"
using suffix_Cons[of "[x]"] by auto

lemma prefix_Cons': "prefix (x#xs) (x#ys) ⟹ prefix xs ys"
by simp

lemma prefix_map: "prefix xs (map f ys) ⟹ ∃zs. prefix zs ys ∧ map f zs = xs"
using map_append_inv unfolding prefix_def by fast

lemma length_prefix_ex:
  assumes "n ≤ length xs"
  shows "∃ys zs. xs = ys@zs ∧ length ys = n"
  using assms
proof (induction n)
  case (Suc n)
  then obtain ys zs where IH: "xs = ys@zs" "length ys = n" by moura
  hence "length zs > 0" using Suc.prems(1) by auto
  then obtain v vs where v: "zs = v#vs" by (metis Suc_length_conv gr0_conv_Suc)
  hence "length (ys@[v]) = Suc n" using IH(2) by simp
  thus ?case using IH(1) v by (metis append.assoc append_Cons append_Nil)
qed simp

lemma length_prefix_ex':
  assumes "n < length xs"
  shows "∃ys zs. xs = ys@xs ! n#zs ∧ length ys = n"
proof -
  obtain ys zs where xs: "xs = ys@zs" "length ys = n" using assms length_prefix_ex[of n xs] by moura
  hence "length zs > 0" using assms by auto
  then obtain v vs where v: "zs = v#vs" by (metis Suc_length_conv gr0_conv_Suc)
  hence "(ys@zs) ! n = v" using xs by auto
  thus ?thesis using v xs by auto
qed

lemma length_prefix_ex2:
  assumes "i < length xs" "j < length xs" "i < j"
  shows "∃ys zs vs. xs = ys@xs ! i#zs@xs ! j#vs ∧ length ys = i ∧ length zs = j - i - 1"
by (smt assms length_prefix_ex' nth_append append.assoc append.simps(2) add_diff_cancel_left'
      diff_Suc_1 length_Cons length_append)
```

2.1.4 List: products

```
lemma product_lists_Cons:
  "x#xs ∈ set (product_lists (y#ys)) ⟷ (xs ∈ set (product_lists ys) ∧ x ∈ set y)"
by auto
```

```
lemma product_lists_in_set_nth:
  assumes "xs ∈ set (product_lists ys)"
  shows "∀i<length ys. xs ! i ∈ set (ys ! i)"
proof -
  have 0: "length ys = length xs" using assms(1) by (simp add: in_set_product_lists_length)
  thus ?thesis using assms
  proof (induction ys arbitrary: xs)
    case (Cons y ys)
    obtain x xs' where xs: "xs = x#xs'" using Cons.prems(1) by (metis length_Suc_conv)
    hence "xs' ∈ set (product_lists ys) ⟹ ∀i<length ys. xs' ! i ∈ set (ys ! i)"
          "length ys = length xs'" "x#xs' ∈ set (product_lists (y#ys))"
      using Cons by simp_all
    thus ?case using xs product_lists_Cons[of x xs' y ys] by (simp add: nth_Cons')
  qed simp
qed

lemma product_lists_in_set_nth':
  assumes "∀i<length xs. ys ! i ∈ set (xs ! i)"
    and "length xs = length ys"
  shows "ys ∈ set (product_lists xs)"
using assms
proof (induction xs arbitrary: ys)
  case (Cons x xs)
  obtain y ys' where ys: "ys = y#ys'" using Cons.prems(2) by (metis length_Suc_conv)
  hence "ys' ∈ set (product_lists xs)" "y ∈ set x" "length xs = length ys'"
    using Cons by fastforce+
  thus ?case using ys product_lists_Cons[of y ys' x xs] by (simp add: nth_Cons')
qed simp
```

2.1.5 Other Lemmata

```
lemma inv_set_fset: "finite M ⟹ set (inv set M) = M"
unfolding inv_def by (metis (mono_tags) finite_list someI_ex)

lemma lfp_eqI':
  assumes "mono f"
    and "f C = C"
    and "∀X ∈ Pow C. f X = X ⟶ X = C"
  shows "lfp f = C"
by (metis PowI assms lfp_lowerbound lfp_unfold subset_refl)

lemma lfp_while':
  fixes f::"'a set ⇒ 'a set" and M::"'a set"
  defines "N ≡ while (λA. f A ≠ A) f {}"
  assumes f_mono: "mono f"
    and N_finite: "finite N"
    and N_supset: "f N ⊆ N"
  shows "lfp f = N"
proof -
  have *: "f X ⊆ N" when "X ⊆ N" for X using N_supset monoD[OF f_mono that] by blast
  show ?thesis
    using lfp_while[OF f_mono * N_finite]
    by (simp add: N_def)
qed

lemma lfp_while'':
  fixes f::"'a set ⇒ 'a set" and M::"'a set"
  defines "N ≡ while (λA. f A ≠ A) f {}"
  assumes f_mono: "mono f"
    and lfp_finite: "finite (lfp f)"
  shows "lfp f = N"
proof -
```

```
  have *: "f X ⊆ lfp f" when "X ⊆ lfp f" for X
    using lfp_fixpoint[OF f_mono] monoD[OF f_mono that]
    by blast
  show ?thesis
    using lfp_while[OF f_mono * lfp_finite]
    by (simp add: N_def)
qed

lemma preordered_finite_set_has_maxima:
  assumes "finite A" "A ≠ {}"
  shows "∃a::'a::{preorder} ∈ A. ∀b ∈ A. ¬(a < b)"
using assms
proof (induction A rule: finite_induct)
  case (insert a A) thus ?case
    by (cases "A = {}", simp, metis insert_iff order_trans less_le_not_le)
qed simp

lemma partition_index_bij:
  fixes n::nat
  obtains I k where
    "bij_betw I {0..<n} {0..<n}" "k ≤ n"
    "∀i. i < k ⟶ P (I i)"
    "∀i. k ≤ i ∧ i < n ⟶ ¬(P (I i))"
proof -
  define A where "A = filter P [0..<n]"
  define B where "B = filter (λi. ¬P i) [0..<n]"
  define k where "k = length A"
  define I where "I = (λn. (A@B) ! n)"

  note defs = A_def B_def k_def I_def

  have k1: "k ≤ n" by (metis defs(1,3) diff_le_self dual_order.trans length_filter_le length_upt)

  have "i < k ⟹ P (A ! i)" for i by (metis defs(1,3) filter_nth)
  hence k2: "i < k ⟹ P ((A@B) ! i)" for i by (simp add: defs nth_append)

  have "i < length B ⟹ ¬(P (B ! i))" for i by (metis defs(2) filter_nth)
  hence "i < length B ⟹ ¬(P ((A@B) ! (k + i)))" for i using k_def by simp
  hence "k ≤ i ∧ i < k + length B ⟹ ¬(P ((A@B) ! i))" for i
    by (metis add.commute add_less_imp_less_right le_add_diff_inverse2)
  hence k3: "k ≤ i ∧ i < n ⟹ ¬(P ((A@B) ! i))" for i by (simp add: defs sum_length_filter_compl)

  have *: "length (A@B) = n" "set (A@B) = {0..<n}" "distinct (A@B)"
    by (metis defs(1,2) diff_zero length_append length_upt sum_length_filter_compl)
       (auto simp add: defs)

  have I: "bij_betw I {0..<n} {0..<n}"
  proof (intro bij_betwI')
    fix x y show "x ∈ {0..<n} ⟹ y ∈ {0..<n} ⟹ (I x = I y) = (x = y)"
      by (metis *(1,3) defs(4) nth_eq_iff_index_eq atLeastLessThan_iff)
  next
    fix x show "x ∈ {0..<n} ⟹ I x ∈ {0..<n}"
      by (metis *(1,2) defs(4) atLeastLessThan_iff nth_mem)
  next
    fix y show "y ∈ {0..<n} ⟹ ∃x ∈ {0..<n}. y = I x"
      by (metis * defs(4) atLeast0LessThan distinct_Ex1 lessThan_iff)
  qed

  show ?thesis using k1 k2 k3 I that by (simp add: defs)
qed

lemma finite_lists_length_eq':
  assumes "⋀x. x ∈ set xs ⟹ finite {y. P x y}"
```

```
  shows "finite {ys. length xs = length ys ∧ (∀y ∈ set ys. ∃x ∈ set xs. P x y)}"
proof -
  define Q where "Q ≡ λys. ∀y ∈ set ys. ∃x ∈ set xs. P x y"
  define M where "M ≡ {y. ∃x ∈ set xs. P x y}"

  have 0: "finite M" using assms unfolding M_def by fastforce

  have "Q ys ⟷ set ys ⊆ M"
       "(Q ys ∧ length ys = length xs) ⟷ (length xs = length ys ∧ Q ys)"
    for ys
    unfolding Q_def M_def by auto
  thus ?thesis
    using finite_lists_length_eq[OF 0, of "length xs"]
    unfolding Q_def by presburger
qed

lemma trancl_eqI:
  assumes "∀(a,b) ∈ A. ∀(c,d) ∈ A. b = c ⟶ (a,d) ∈ A"
  shows "A = A⁺"
proof
  show "A⁺ ⊆ A"
  proof
    fix x assume x: "x ∈ A⁺"
    then obtain a b where ab: "x = (a,b)" by (metis surj_pair)
    hence "(a,b) ∈ A⁺" using x by metis
    hence "(a,b) ∈ A" using assms by (induct rule: trancl_induct) auto
    thus "x ∈ A" using ab by metis
  qed
qed auto

lemma trancl_eqI':
  assumes "∀(a,b) ∈ A. ∀(c,d) ∈ A. b = c ∧ a ≠ d ⟶ (a,d) ∈ A"
    and "∀(a,b) ∈ A. a ≠ b"
  shows "A = {(a,b) ∈ A⁺. a ≠ b}"
proof
  show "{(a,b) ∈ A⁺. a ≠ b} ⊆ A"
  proof
    fix x assume x: "x ∈ {(a,b) ∈ A⁺. a ≠ b}"
    then obtain a b where ab: "x = (a,b)" by (metis surj_pair)
    hence "(a,b) ∈ A⁺" "a ≠ b" using x by blast+
    hence "(a,b) ∈ A"
    proof (induction rule: trancl_induct)
      case base thus ?case by blast
    next
      case step thus ?case using assms(1) by force
    qed
    thus "x ∈ A" using ab by metis
  qed
qed (use assms(2) in auto)

lemma distinct_concat_idx_disjoint:
  assumes xs: "distinct (concat xs)"
    and ij: "i < length xs" "j < length xs" "i < j"
  shows "set (xs ! i) ∩ set (xs ! j) = {}"
proof -
  obtain ys zs vs where ys: "xs = ys@xs ! i#zs@xs ! j#vs" "length ys = i" "length zs = j - i - 1"
    using length_prefix_ex2[OF ij] by moura
  thus ?thesis
    using xs concat_append[of "ys@xs ! i#zs" "xs ! j#vs"]
          distinct_append[of "concat (ys@xs ! i#zs)" "concat (xs ! j#vs)"]
    by auto
qed
```

```
lemma remdups_ex2:
  "length (remdups xs) > 1 ⟹ ∃a ∈ set xs. ∃b ∈ set xs. a ≠ b"
by (metis distinct_Ex1 distinct_remdups less_trans nth_mem set_remdups zero_less_one zero_neq_one)

lemma trancl_minus_refl_idem:
  defines "cl ≡ λts. {(a,b) ∈ ts⁺. a ≠ b}"
  shows "cl (cl ts) = cl ts"
proof -
  have 0: "(ts⁺)⁺ = ts⁺" "cl ts ⊆ ts⁺" "(cl ts)⁺ ⊆ (ts⁺)⁺"
  proof -
    show "(ts⁺)⁺ = ts⁺" "cl ts ⊆ ts⁺" unfolding cl_def by auto
    thus "(cl ts)⁺ ⊆ (ts⁺)⁺" using trancl_mono[of _ "cl ts" "ts⁺"] by blast
  qed

  have 1: "t ∈ cl (cl ts)" when t: "t ∈ cl ts" for t
    using t 0 unfolding cl_def by fast

  have 2: "t ∈ cl ts" when t: "t ∈ cl (cl ts)" for t
  proof -
    obtain a b where ab: "t = (a,b)" by (metis surj_pair)
    have "t ∈ (cl ts)⁺" and a_neq_b: "a ≠ b" using t unfolding cl_def ab by force+
    hence "t ∈ ts⁺" using 0 by blast
    thus ?thesis using a_neq_b unfolding cl_def ab by blast
  qed

  show ?thesis using 1 2 by blast
qed
```

2.1.6 Infinite Paths in Relations as Mappings from Naturals to States

```
context
begin

private fun rel_chain_fun::"nat ⇒ 'a ⇒ 'a ⇒ ('a × 'a) set ⇒ 'a" where
  "rel_chain_fun 0 x _ _ = x"
| "rel_chain_fun (Suc i) x y r = (if i = 0 then y else SOME z. (rel_chain_fun i x y r, z) ∈ r)"

lemma infinite_chain_intro:
  fixes r::"('a × 'a) set"
  assumes "∀(a,b) ∈ r. ∃c. (b,c) ∈ r" "r ≠ {}"
  shows "∃f. ∀i::nat. (f i, f (Suc i)) ∈ r"
proof -
  from assms(2) obtain a b where "(a,b) ∈ r" by auto

  let ?P = "λi. (rel_chain_fun i a b r, rel_chain_fun (Suc i) a b r) ∈ r"
  let ?Q = "λi. ∃z. (rel_chain_fun i a b r, z) ∈ r"

  have base: "?P 0" using ‹(a,b) ∈ r› by auto

  have step: "?P (Suc i)" when i: "?P i" for i
  proof -
    have "?Q (Suc i)" using assms(1) i by auto
    thus ?thesis using someI_ex[OF ‹?Q (Suc i)›] by auto
  qed

  have "∀i::nat. (rel_chain_fun i a b r, rel_chain_fun (Suc i) a b r) ∈ r"
    using base step nat_induct[of ?P] by simp
  thus ?thesis by fastforce
qed

end

lemma infinite_chain_intro':
```

```
  fixes r::"('a × 'a) set"
  assumes base: "∃b. (x,b) ∈ r" and step: "∀b. (x,b) ∈ r⁺ ⟶ (∃c. (b,c) ∈ r)"
  shows "∃f. ∀i::nat. (f i, f (Suc i)) ∈ r"
proof -
  let ?s = "{(a,b) ∈ r. a = x ∨ (x,a) ∈ r⁺}"

  have "?s ≠ {}" using base by auto

  have "∃c. (b,c) ∈ ?s" when ab: "(a,b) ∈ ?s" for a b
  proof (cases "a = x")
    case False
    hence "(x,a) ∈ r⁺" using ab by auto
    hence "(x,b) ∈ r⁺" using ‹(a,b) ∈ ?s› by auto
    thus ?thesis using step by auto
  qed (use ab step in auto)
  hence "∃f. ∀i. (f i, f (Suc i)) ∈ ?s" using infinite_chain_intro[of ?s] ‹?s ≠ {}› by blast
  thus ?thesis by auto
qed

lemma infinite_chain_mono:
  assumes "S ⊆ T" "∃f. ∀i::nat. (f i, f (Suc i)) ∈ S"
  shows "∃f. ∀i::nat. (f i, f (Suc i)) ∈ T"
using assms by auto

end
```

2.2 Protocol Messages as (First-Order) Terms (Messages)

```
theory Messages
  imports Miscellaneous "First_Order_Terms.Term"
begin
```

2.2.1 Term-related definitions: subterms and free variables

```
abbreviation "the_Fun ≡ un_Fun1"
lemmas the_Fun_def = un_Fun1_def

fun subterms::"('a,'b) term ⇒ ('a,'b) terms" where
  "subterms (Var x) = {Var x}"
| "subterms (Fun f T) = {Fun f T} ∪ (⋃t ∈ set T. subterms t)"

abbreviation subtermeq (infix "⊑" 50) where "t' ⊑ t ≡ (t' ∈ subterms t)"
abbreviation subterm (infix "⊏" 50) where "t' ⊏ t ≡ (t' ⊑ t ∧ t' ≠ t)"

abbreviation "subterms_set M ≡ ⋃(subterms ` M)"
abbreviation subtermeqset (infix "⊑_set" 50) where "t ⊑_set M ≡ (t ∈ subterms_set M)"

abbreviation fv where "fv ≡ vars_term"
lemmas fv_simps = term.simps(17,18)

fun fv_set where "fv_set M = ⋃(fv ` M)"

abbreviation fv_pair where "fv_pair p ≡ case p of (t,t') ⇒ fv t ∪ fv t'"

fun fv_pairs where "fv_pairs F = ⋃(fv_pair ` set F)"

abbreviation ground where "ground M ≡ fv_set M = {}"
```

2.2.2 Variants that return lists insteads of sets

```
fun fv_list where
  "fv_list (Var x) = [x]"
```

```
| "fv_list (Fun f T) = concat (map fv_list T)"

definition fv_list_pairs where
  "fv_list_pairs F ≡ concat (map (λ(t,t'). fv_list t@fv_list t') F)"

fun subterms_list::"('a,'b) term ⇒ ('a,'b) term list" where
  "subterms_list (Var x) = [Var x]"
| "subterms_list (Fun f T) = remdups (Fun f T#concat (map subterms_list T))"

lemma fv_list_is_fv: "fv t = set (fv_list t)"
by (induct t) auto

lemma fv_list_pairs_is_fv_pairs: "fv_pairs F = set (fv_list_pairs F)"
by (induct F) (auto simp add: fv_list_is_fv fv_list_pairs_def)

lemma subterms_list_is_subterms: "subterms t = set (subterms_list t)"
by (induct t) auto
```

2.2.3 The subterm relation defined as a function

```
fun subterm_of where
  "subterm_of t (Var y) = (t = Var y)"
| "subterm_of t (Fun f T) = (t = Fun f T ∨ list_ex (subterm_of t) T)"

lemma subterm_of_iff_subtermeq[code_unfold]: "t ⊑ t' = subterm_of t t'"
proof (induction t')
  case (Fun f T) thus ?case
  proof (cases "t = Fun f T")
    case False thus ?thesis
      using Fun.IH subterm_of.simps(2)[of t f T]
      unfolding list_ex_iff by fastforce
  qed simp
qed simp

lemma subterm_of_ex_set_iff_subtermeqset[code_unfold]: "t ⊑_set M = (∃t' ∈ M. subterm_of t t')"
using subterm_of_iff_subtermeq by blast
```

2.2.4 The subterm relation is a partial order on terms

```
interpretation "term": order "(⊑)" "(⊏)"
proof
  show "s ⊑ s" for s :: "('a,'b) term"
    by (induct s rule: subterms.induct) auto

  show trans: "s ⊑ t ⟹ t ⊑ u ⟹ s ⊑ u" for s t u :: "('a,'b) term"
    by (induct u rule: subterms.induct) auto

  show "s ⊑ t ⟹ t ⊑ s ⟹ s = t" for s t :: "('a,'b) term"
  proof (induction s arbitrary: t rule: subterms.induct[case_names Var Fun])
    case (Fun f T)
    { assume 0: "t ≠ Fun f T"
      then obtain u::"('a,'b) term" where u: "u ∈ set T" "t ⊑ u" using Fun.prems(2) by auto
      hence 1: "Fun f T ⊑ u" using trans[OF Fun.prems(1)] by simp

      have 2: "u ⊑ Fun f T"
        by (cases u) (use u(1) in force, use u(1) subterms.simps(2)[of f T] in fastforce)
      hence 3: "u = Fun f T" using Fun.IH[OF u(1) _ 1] by simp

      have "u ⊑ t" using trans[OF 2 Fun.prems(1)] by simp
      hence 4: "u = t" using Fun.IH[OF u(1) _ u(2)] by simp

      have "t = Fun f T" using 3 4 by simp
      hence False using 0 by simp
```

```
    }
    thus ?case by auto
  qed simp
  thus "(s ⊏ t) = (s ⊑ t ∧ ¬(t ⊑ s))" for s t :: "('a,'b) term"
    by blast
qed
```

2.2.5 Lemmata concerning subterms and free variables

```
lemma fv_list_pairs_append: "fv_list_pairs (F@G) = fv_list_pairs F@fv_list_pairs G"
by (simp add: fv_list_pairs_def)

lemma distinct_fv_list_idx_fv_disjoint:
  assumes t: "distinct (fv_list t)" "Fun f T ⊑ t"
    and ij: "i < length T" "j < length T" "i < j"
  shows "fv (T ! i) ∩ fv (T ! j) = {}"
using t
proof (induction t rule: fv_list.induct)
  case (2 g S)
  have "distinct (fv_list s)" when s: "s ∈ set S" for s
    by (metis (no_types, lifting) s "2.prems"(1) concat_append distinct_append
          map_append split_list fv_list.simps(2) concat.simps(2) list.simps(9))
  hence IH: "fv (T ! i) ∩ fv (T ! j) = {}"
    when s: "s ∈ set S" "Fun f T ⊑ s" for s
    using "2.IH" s by blast

  show ?case
  proof (cases "Fun f T = Fun g S")
    case True
    define U where "U ≡ map fv_list T"

    have a: "distinct (concat U)"
      using "2.prems"(1) True unfolding U_def by auto

    have b: "i < length U" "j < length U"
      using ij(1,2) unfolding U_def by simp_all

    show ?thesis
      using b distinct_concat_idx_disjoint[OF a b ij(3)]
            fv_list_is_fv[of "T ! i"] fv_list_is_fv[of "T ! j"]
      unfolding U_def by force
  qed (use IH "2.prems"(2) in auto)
qed force

lemmas subtermeqI'[intro] = term.eq_refl

lemma subtermeqI''[intro]: "t ∈ set T ⟹ t ⊑ Fun f T"
by force

lemma finite_fv_set[intro]: "finite M ⟹ finite (fv_set M)"
by auto

lemma finite_fun_symbols[simp]: "finite (funs_term t)"
by (induct t) simp_all

lemma fv_set_mono: "M ⊆ N ⟹ fv_set M ⊆ fv_set N"
by auto

lemma subterms_set_mono: "M ⊆ N ⟹ subterms_set M ⊆ subterms_set N"
by auto

lemma ground_empty[simp]: "ground {}"
by simp
```

lemma *ground_subset: "M ⊆ N ⟹ ground N ⟹ ground M"*
by *auto*

lemma *fv_map_fv_set: "⋃(set (map fv L)) = fv$_{set}$ (set L)"*
by *(induct L) auto*

lemma *fv$_{set}$_union: "fv$_{set}$ (M ∪ N) = fv$_{set}$ M ∪ fv$_{set}$ N"*
by *auto*

lemma *finite_subset_Union:*
  fixes *A::"'a set"* **and** *f::"'a ⇒ 'b set"*
  assumes *"finite (⋃a ∈ A. f a)"*
  shows *"∃B. finite B ∧ B ⊆ A ∧ (⋃b ∈ B. f b) = (⋃a ∈ A. f a)"*
by *(metis assms eq_iff finite_subset_image finite_UnionD)*

lemma *inv_set_fv: "finite M ⟹ ⋃(set (map fv (inv set M))) = fv$_{set}$ M"*
using *fv_map_fv_set[of "inv set M"] inv_set_fset* **by** *auto*

lemma *ground_subterm: "fv t = {} ⟹ t' ⊑ t ⟹ fv t' = {}"* **by** *(induct t) auto*

lemma *empty_fv_not_var: "fv t = {} ⟹ t ≠ Var x"* **by** *auto*

lemma *empty_fv_exists_fun: "fv t = {} ⟹ ∃f X. t = Fun f X"* **by** *(cases t) auto*

lemma *vars_iff_subtermeq: "x ∈ fv t ⟷ Var x ⊑ t"* **by** *(induct t) auto*

lemma *vars_iff_subtermeq_set: "x ∈ fv$_{set}$ M ⟷ Var x ∈ subterms$_{set}$ M"*
using *vars_iff_subtermeq[of x]* **by** *auto*

lemma *vars_if_subtermeq_set: "Var x ∈ subterms$_{set}$ M ⟹ x ∈ fv$_{set}$ M"*
by *(metis vars_iff_subtermeq_set)*

lemma *subtermeq_set_if_vars: "x ∈ fv$_{set}$ M ⟹ Var x ∈ subterms$_{set}$ M"*
by *(metis vars_iff_subtermeq_set)*

lemma *vars_iff_subterm_or_eq: "x ∈ fv t ⟷ Var x ⊏ t ∨ Var x = t"*
by *(induct t) (auto simp add: vars_iff_subtermeq)*

lemma *var_is_subterm: "x ∈ fv t ⟹ Var x ∈ subterms t"*
by *(simp add: vars_iff_subtermeq)*

lemma *subterm_is_var: "Var x ∈ subterms t ⟹ x ∈ fv t"*
by *(simp add: vars_iff_subtermeq)*

lemma *no_var_subterm: "¬t ⊏ Var v"* **by** *auto*

lemma *fun_if_subterm: "t ⊏ u ⟹ ∃f X. u = Fun f X"* **by** *(induct u) simp_all*

lemma *subtermeq_vars_subset: "M ⊑ N ⟹ fv M ⊆ fv N"* **by** *(induct N) auto*

lemma *fv_subterms[simp]: "fv$_{set}$ (subterms t) = fv t"*
by *(induct t) auto*

lemma *fv_subterms_set[simp]: "fv$_{set}$ (subterms$_{set}$ M) = fv$_{set}$ M"*
using *subtermeq_vars_subset* **by** *auto*

lemma *fv_subset: "t ∈ M ⟹ fv t ⊆ fv$_{set}$ M"*
by *auto*

lemma *fv_subset_subterms: "t ∈ subterms$_{set}$ M ⟹ fv t ⊆ fv$_{set}$ M"*
using *fv_subset fv_subterms_set* **by** *metis*

```
lemma subterms_finite[simp]: "finite (subterms t)" by (induction rule: subterms.induct) auto

lemma subterms_union_finite: "finite M ⟹ finite (⋃t ∈ M. subterms t)"
by (induction rule: subterms.induct) auto

lemma subterms_subset: "t' ⊑ t ⟹ subterms t' ⊆ subterms t"
by (induction rule: subterms.induct) auto

lemma subterms_subset_set: "M ⊆ subterms t ⟹ subterms_set M ⊆ subterms t"
by (metis SUP_least contra_subsetD subterms_subset)

lemma subset_subterms_Union[simp]: "M ⊆ subterms_set M" by auto

lemma in_subterms_Union: "t ∈ M ⟹ t ∈ subterms_set M" using subset_subterms_Union by blast

lemma in_subterms_subset_Union: "t ∈ subterms_set M ⟹ subterms t ⊆ subterms_set M"
using subterms_subset by auto

lemma subterm_param_split:
  assumes "t ⊏ Fun f X"
  shows "∃pre x suf. t ⊑ x ∧ X = pre@x#suf"
proof -
  obtain x where "t ⊑ x" "x ∈ set X" using assms by auto
  then obtain pre suf where "X = pre@x#suf" "x ∉ set pre ∨ x ∉ set suf"
    by (meson split_list_first split_list_last)
  thus ?thesis using ‹t ⊑ x› by auto
qed

lemma ground_iff_no_vars: "ground (M::('a,'b) terms) ⟷ (∀v. Var v ∉ (⋃m ∈ M. subterms m))"
proof
  assume "ground M"
  hence "∀v. ∀m ∈ M. v ∉ fv m" by auto
  hence "∀v. ∀m ∈ M. Var v ∉ subterms m" by (simp add: vars_iff_subtermeq)
  thus "(∀v. Var v ∉ (⋃m ∈ M. subterms m))" by simp
next
  assume no_vars: "∀v. Var v ∉ (⋃m ∈ M. subterms m)"
  moreover
  { assume "¬ground M"
    then obtain v and m::"('a,'b) term" where "m ∈ M" "fv m ≠ {}" "v ∈ fv m" by auto
    hence "Var v ∈ (subterms m)" by (simp add: vars_iff_subtermeq)
    hence "∃v. Var v ∈ (⋃t ∈ M. subterms t)" using ‹m ∈ M› by auto
    hence False using no_vars by simp
  }
  ultimately show "ground M" by blast
qed

lemma index_Fun_subterms_subset[simp]: "i < length T ⟹ subterms (T ! i) ⊆ subterms (Fun f T)"
by auto

lemma index_Fun_fv_subset[simp]: "i < length T ⟹ fv (T ! i) ⊆ fv (Fun f T)"
using subtermeq_vars_subset by fastforce

lemma subterms_union_ground:
  assumes "ground M"
  shows "ground (subterms_set M)"
proof -
  { fix t assume "t ∈ M"
    hence "fv t = {}"
      using ground_iff_no_vars[of M] assms
      by auto
    hence "∀t' ∈ subterms t. fv t' = {}" using subtermeq_vars_subset[of _ t] by simp
    hence "ground (subterms t)" by auto
  }
```

```
  thus ?thesis by auto
qed

lemma Var_subtermeq: "t ⊑ Var v ⟹ t = Var v" by simp

lemma subtermeq_imp_funs_term_subset: "s ⊑ t ⟹ funs_term s ⊆ funs_term t"
by (induct t arbitrary: s) auto

lemma subterms_const: "subterms (Fun f []) = {Fun f []}" by simp

lemma subterm_subtermeq_neq: "⟦t ⊏ u; u ⊑ v⟧ ⟹ t ≠ v"
by (metis term.eq_iff)

lemma subtermeq_subterm_neq: "⟦t ⊑ u; u ⊏ v⟧ ⟹ t ≠ v"
by (metis term.eq_iff)

lemma subterm_size_lt: "x ⊏ y ⟹ size x < size y"
using not_less_eq size_list_estimation by (induct y, simp, fastforce)

lemma in_subterms_eq: "⟦x ∈ subterms y; y ∈ subterms x⟧ ⟹ subterms x = subterms y"
using term.antisym by auto

lemma Fun_gt_params: "Fun f X ∉ (⋃x ∈ set X. subterms x)"
proof -
  have "size_list size X < size (Fun f X)" by simp
  hence "Fun f X ∉ set X" by (meson less_not_refl size_list_estimation)
  hence "∀x ∈ set X. Fun f X ∉ subterms x ∨ x ∉ subterms (Fun f X)"
    by (metis term.antisym[of "Fun f X" _])
  moreover have "∀x ∈ set X. x ∈ subterms (Fun f X)" by fastforce
  ultimately show ?thesis by auto
qed

lemma params_subterms[simp]: "set X ⊆ subterms (Fun f X)" by auto

lemma params_subterms_Union[simp]: "subterms_set (set X) ⊆ subterms (Fun f X)" by auto

lemma Fun_subterm_inside_params: "t ⊏ Fun f X ⟷ t ∈ (⋃x ∈ (set X). subterms x)"
using Fun_gt_params by fastforce

lemma Fun_param_is_subterm: "x ∈ set X ⟹ x ⊏ Fun f X"
using Fun_subterm_inside_params by fastforce

lemma Fun_param_in_subterms: "x ∈ set X ⟹ x ∈ subterms (Fun f X)"
using Fun_subterm_inside_params by fastforce

lemma Fun_not_in_param: "x ∈ set X ⟹ ¬Fun f X ⊏ x"
using term.antisym by fast

lemma Fun_ex_if_subterm: "t ⊏ s ⟹ ∃f T. Fun f T ⊑ s ∧ t ∈ set T"
proof (induction s)
  case (Fun f T)
  then obtain s' where s': "s' ∈ set T" "t ⊑ s'" by auto
  show ?case
  proof (cases "t = s'")
    case True thus ?thesis using s' by blast
  next
    case False
    thus ?thesis
      using Fun.IH[OF s'(1)] s'(2) term.order_trans[OF _ Fun_param_in_subterms[OF s'(1), of f]]
      by metis
  qed
qed simp
```

```
lemma const_subterm_obtain:
  assumes "fv t = {}"
  obtains c where "Fun c [] ⊑ t"
using assms
proof (induction t)
  case (Fun f T) thus ?case by (cases "T = []") force+
qed simp

lemma const_subterm_obtain': "fv t = {} ⟹ ∃c. Fun c [] ⊑ t"
by (metis const_subterm_obtain)

lemma subterms_singleton:
  assumes "(∃v. t = Var v) ∨ (∃f. t = Fun f [])"
  shows "subterms t = {t}"
using assms by (cases t) auto

lemma subtermeq_Var_const:
  assumes "s ⊑ t"
  shows "t = Var v ⟹ s = Var v" "t = Fun f [] ⟹ s = Fun f []"
using assms by fastforce+

lemma subterms_singleton':
  assumes "subterms t = {t}"
  shows "(∃v. t = Var v) ∨ (∃f. t = Fun f [])"
proof (cases t)
  case (Fun f T)
  { fix s S assume "T = s#S"
    hence "s ∈ subterms t" using Fun by auto
    hence "s = t" using assms by auto
    hence False
      using Fun_param_is_subterm[of s "s#S" f] ‹T = s#S› Fun
      by auto
  }
  hence "T = []" by (cases T) auto
  thus ?thesis using Fun by simp
qed (simp add: assms)

lemma funs_term_subterms_eq[simp]:
  "(⋃s ∈ subterms t. funs_term s) = funs_term t"
  "(⋃s ∈ subterms_set M. funs_term s) = ⋃(funs_term ` M)"
proof -
  show "⋀t. ⋃(funs_term ` subterms t) = funs_term t"
    using term.order_refl subtermeq_imp_funs_term_subset by blast
  thus "⋃(funs_term ` (subterms_set M)) = ⋃(funs_term ` M)" by force
qed

lemmas subtermI'[intro] = Fun_param_is_subterm

lemma funs_term_Fun_subterm: "f ∈ funs_term t ⟹ ∃T. Fun f T ∈ subterms t"
proof (induction t)
  case (Fun g T)
  hence "f = g ∨ (∃s ∈ set T. f ∈ funs_term s)" by simp
  thus ?case
  proof
    assume "∃s ∈ set T. f ∈ funs_term s"
    then obtain s where "s ∈ set T" "∃T. Fun f T ∈ subterms s" using Fun.IH by auto
    thus ?thesis by auto
  qed (auto simp add: Fun)
qed simp

lemma funs_term_Fun_subterm': "Fun f T ∈ subterms t ⟹ f ∈ funs_term t"
by (induct t) auto
```

```
lemma zip_arg_subterm:
  assumes "(s,t) ∈ set (zip X Y)"
  shows "s ⊏ Fun f X" "t ⊏ Fun g Y"
proof -
  from assms have *: "s ∈ set X" "t ∈ set Y" by (meson in_set_zipE)+
  show "s ⊏ Fun f X" by (metis Fun_param_is_subterm[OF *(1)])
  show "t ⊏ Fun g Y" by (metis Fun_param_is_subterm[OF *(2)])
qed

lemma fv_disj_Fun_subterm_param_cases:
  assumes "fv t ∩ X = {}" "Fun f T ∈ subterms t"
  shows "T = [] ∨ (∃s∈set T. s ∉ Var ` X)"
proof (cases T)
  case (Cons s S)
  hence "s ∈ subterms t"
    using assms(2) term.order_trans[of _ "Fun f T" t]
    by auto
  hence "fv s ∩ X = {}" using assms(1) fv_subterms by force
  thus ?thesis using Cons by auto
qed simp

lemma fv_eq_FunI:
  assumes "length T = length S" "⋀i. i < length T ⟹ fv (T ! i) = fv (S ! i)"
  shows "fv (Fun f T) = fv (Fun g S)"
using assms
proof (induction T arbitrary: S)
  case (Cons t T S')
  then obtain s S where S': "S' = s#S" by (cases S') simp_all
  thus ?case using Cons by fastforce
qed simp

lemma fv_eq_FunI':
  assumes "length T = length S" "⋀i. i < length T ⟹ x ∈ fv (T ! i) ⟷ x ∈ fv (S ! i)"
  shows "x ∈ fv (Fun f T) ⟷ x ∈ fv (Fun g S)"
using assms
proof (induction T arbitrary: S)
  case (Cons t T S')
  then obtain s S where S': "S' = s#S" by (cases S') simp_all
  thus ?case using Cons by fastforce
qed simp

lemma finite_fv_pairs[simp]: "finite (fv_pairs x)" by auto

lemma fv_pairs_Nil[simp]: "fv_pairs [] = {}" by simp

lemma fv_pairs_singleton[simp]: "fv_pairs [(t,s)] = fv t ∪ fv s" by simp

lemma fv_pairs_Cons: "fv_pairs ((s,t)#F) = fv s ∪ fv t ∪ fv_pairs F" by simp

lemma fv_pairs_append: "fv_pairs (F@G) = fv_pairs F ∪ fv_pairs G" by simp

lemma fv_pairs_mono: "set M ⊆ set N ⟹ fv_pairs M ⊆ fv_pairs N" by auto

lemma fv_pairs_inI[intro]:
  "f ∈ set F ⟹ x ∈ fv_pair f ⟹ x ∈ fv_pairs F"
  "f ∈ set F ⟹ x ∈ fv (fst f) ⟹ x ∈ fv_pairs F"
  "f ∈ set F ⟹ x ∈ fv (snd f) ⟹ x ∈ fv_pairs F"
  "(t,s) ∈ set F ⟹ x ∈ fv t ⟹ x ∈ fv_pairs F"
  "(t,s) ∈ set F ⟹ x ∈ fv s ⟹ x ∈ fv_pairs F"
using UN_I by fastforce+

lemma fv_pairs_cons_subset: "fv_pairs F ⊆ fv_pairs (f#F)"
by auto
```

2.2.6 Other lemmata

```
lemma nonvar_term_has_composed_shallow_term:
  fixes t::"('f,'v) term"
  assumes "¬(∃x. t = Var x)"
  shows "∃f T. Fun f T ⊑ t ∧ (∀s ∈ set T. (∃c. s = Fun c []) ∨ (∃x. s = Var x))"
proof -
  let ?Q = "λS. ∀s ∈ set S. (∃c. s = Fun c []) ∨ (∃x. s = Var x)"
  let ?P = "λt. ∃g S. Fun g S ⊑ t ∧ ?Q S"
  { fix t::"('f,'v) term"
    have "(∃x. t = Var x) ∨ ?P t"
    proof (induction t)
      case (Fun h R) show ?case
      proof (cases "R = [] ∨ (∀r ∈ set R. ∃x. r = Var x)")
        case False
        then obtain r g S where "r ∈ set R" "?P r" "Fun g S ⊑ r" "?Q S" using Fun.IH by fast
        thus ?thesis by auto
      qed force
    qed simp
  } thus ?thesis using assms by blast
qed

end
```

2.3 Definitions and Properties Related to Substitutions and Unification (More_Unification)

```
theory More_Unification
  imports Messages "First_Order_Terms.Unification"
begin
```

2.3.1 Substitutions

```
abbreviation subst_apply_list (infix "·list" 51) where
  "T ·list ϑ ≡ map (λt. t · ϑ) T"

abbreviation subst_apply_pair (infixl "·p" 60) where
  "d ·p ϑ ≡ (case d of (t,t') ⇒ (t · ϑ, t' · ϑ))"

abbreviation subst_apply_pair_set (infixl "·pset" 60) where
  "M ·pset ϑ ≡ (λd. d ·p ϑ) ` M"

definition subst_apply_pairs (infix "·pairs" 51) where
  "F ·pairs ϑ ≡ map (λf. f ·p ϑ) F"

abbreviation subst_more_general_than (infixl "≼∘" 50) where
  "σ ≼∘ ϑ ≡ ∃γ. ϑ = σ ∘s γ"

abbreviation subst_support (infix "supports" 50) where
  "ϑ supports δ ≡ (∀x. ϑ x · δ = δ x)"

abbreviation rm_var where
  "rm_var v s ≡ s(v := Var v)"

abbreviation rm_vars where
  "rm_vars vs σ ≡ (λv. if v ∈ vs then Var v else σ v)"

definition subst_elim where
  "subst_elim σ v ≡ ∀t. v ∉ fv (t · σ)"

definition subst_idem where
  "subst_idem s ≡ s ∘s s = s"
```

```
lemma subst_support_def: "ϑ supports τ ⟷ τ = ϑ ∘s τ"
unfolding subst_compose_def by metis

lemma subst_supportD: "ϑ supports δ ⟹ ϑ ≼∘ δ"
using subst_support_def by auto

lemma rm_vars_empty[simp]: "rm_vars {} s = s" "rm_vars (set []) s = s"
by simp_all

lemma rm_vars_singleton: "rm_vars {v} s = rm_var v s"
by auto

lemma subst_apply_terms_empty: "M ⋅set Var = M"
by simp

lemma subst_agreement: "(t ⋅ r = t ⋅ s) ⟷ (∀v ∈ fv t. Var v ⋅ r = Var v ⋅ s)"
by (induct t) auto

lemma repl_invariance[dest?]: "v ∉ fv t ⟹ t ⋅ s(v := u) = t ⋅ s"
by (simp add: subst_agreement)

lemma subst_idx_map:
  assumes "∀i ∈ set I. i < length T"
  shows "(map ((!) T) I) ⋅list δ = map ((!) (map (λt. t ⋅ δ) T)) I"
using assms by auto

lemma subst_idx_map':
  assumes "∀i ∈ fvset (set K). i < length T"
  shows "(K ⋅list (!) T) ⋅list δ = K ⋅list ((!) (map (λt. t ⋅ δ) T))" (is "?A = ?B")
proof -
  have "T ! i ⋅ δ = (map (λt. t ⋅ δ) T) ! i"
    when "i < length T" for i
    using that by auto
  hence "T ! i ⋅ δ = (map (λt. t ⋅ δ) T) ! i"
    when "i ∈ fvset (set K)" for i
    using that assms by auto
  hence "k ⋅ (!) T ⋅ δ = k ⋅ (!) (map (λt. t ⋅ δ) T)"
    when "fv k ⊆ fvset (set K)" for k
    using that by (induction k) force+
  thus ?thesis by auto
qed

lemma subst_remove_var: "v ∉ fv s ⟹ v ∉ fv (t ⋅ Var(v := s))"
by (induct t) simp_all

lemma subst_set_map: "x ∈ set X ⟹ x ⋅ s ∈ set (map (λx. x ⋅ s) X)"
by simp

lemma subst_set_idx_map:
  assumes "∀i ∈ I. i < length T"
  shows "(!) T ` I ⋅set δ = (!) (map (λt. t ⋅ δ) T) ` I" (is "?A = ?B")
proof
  have *: "T ! i ⋅ δ = (map (λt. t ⋅ δ) T) ! i"
    when "i < length T" for i
    using that by auto

  show "?A ⊆ ?B" using * assms by blast
  show "?B ⊆ ?A" using * assms by auto
qed

lemma subst_set_idx_map':
  assumes "∀i ∈ fvset K. i < length T"
```

```
  shows "K ·set (!) T ·set δ = K ·set (!) (map (λt. t · δ) T)" (is "?A = ?B")
proof
  have "T ! i · δ = (map (λt. t · δ) T) ! i"
    when "i < length T" for i
    using that by auto
  hence "T ! i · δ = (map (λt. t · δ) T) ! i"
    when "i ∈ fvset K" for i
    using that assms by auto
  hence *: "k · (!) T · δ = k · (!) (map (λt. t · δ) T)"
    when "fv k ⊆ fvset K" for k
    using that by (induction k) force+

  show "?A ⊆ ?B" using * by auto
  show "?B ⊆ ?A" using * by force
qed

lemma subst_term_list_obtain:
  assumes "∀i < length T. ∃s. P (T ! i) s ∧ S ! i = s · δ"
    and "length T = length S"
  shows "∃U. length T = length U ∧ (∀i < length T. P (T ! i) (U ! i)) ∧ S = map (λu. u · δ) U"
using assms
proof (induction T arbitrary: S)
  case (Cons t T S')
  then obtain s S where S': "S' = s#S" by (cases S') auto

  have "∀i < length T. ∃s. P (T ! i) s ∧ S ! i = s · δ" "length T = length S"
    using Cons.prems S' by force+
  then obtain U where U:
      "length T = length U" "∀i < length T. P (T ! i) (U ! i)" "S = map (λu. u · δ) U"
    using Cons.IH by moura

  obtain u where u: "P t u" "s = u · δ"
    using Cons.prems(1) S' by auto

  have 1: "length (t#T) = length (u#U)"
    using Cons.prems(2) U(1) by fastforce

  have 2: "∀i < length (t#T). P ((t#T) ! i) ((u#U) ! i)"
    using u(1) U(2) by (simp add: nth_Cons')

  have 3: "S' = map (λu. u · δ) (u#U)"
    using U u S' by simp

  show ?case using 1 2 3 by blast
qed simp

lemma subst_mono: "t ⊑ u ⟹ t · s ⊑ u · s"
by (induct u) auto

lemma subst_mono_fv: "x ∈ fv t ⟹ s x ⊑ t · s"
by (induct t) auto

lemma subst_mono_neq:
  assumes "t ⊏ u"
  shows "t · s ⊏ u · s"
proof (cases u)
  case (Var v)
  hence False using ‹t ⊏ u› by simp
  thus ?thesis ..
next
  case (Fun f X)
  then obtain x where "x ∈ set X" "t ⊑ x" using ‹t ⊏ u› by auto
  hence "t · s ⊑ x · s" using subst_mono by metis
```

```
  obtain Y where "Fun f X · s = Fun f Y" by auto
  hence "x · s ∈ set Y" using ‹x ∈ set X› by auto
  hence "x · s ⊏ Fun f X · s" using ‹Fun f X · s = Fun f Y› Fun_param_is_subterm by simp
  hence "t · s ⊏ Fun f X · s" using ‹t · s ⊑ x · s› by (metis term.dual_order.trans term.eq_iff)
  thus ?thesis using ‹u = Fun f X› ‹t ⊏ u› by metis
qed

lemma subst_no_occs[dest]: "¬Var v ⊑ t ⟹ t · Var(v := s) = t"
by (induct t) (simp_all add: map_idI)

lemma var_comp[simp]: "σ ∘s Var = σ" "Var ∘s σ = σ"
unfolding subst_compose_def by simp_all

lemma subst_comp_all: "M ·set (δ ∘s ϑ) = (M ·set δ) ·set ϑ"
using subst_subst_compose[of _ δ ϑ] by auto

lemma subst_all_mono: "M ⊆ M' ⟹ M ·set s ⊆ M' ·set s"
by auto

lemma subst_comp_set_image: "(δ ∘s ϑ) ` X = δ ` X ·set ϑ"
using subst_compose by fastforce

lemma subst_ground_ident[dest?]: "fv t = {} ⟹ t · s = t"
by (induct t, simp, metis subst_agreement empty_iff subst_apply_term_empty)

lemma subst_ground_ident_compose:
  "fv (σ x) = {} ⟹ (σ ∘s ϑ) x = σ x"
  "fv (t · σ) = {} ⟹ t · (σ ∘s ϑ) = t · σ"
using subst_subst_compose[of t σ ϑ]
by (simp_all add: subst_compose_def subst_ground_ident)

lemma subst_all_ground_ident[dest?]: "ground M ⟹ M ·set s = M"
proof -
  assume "ground M"
  hence "⋀t. t ∈ M ⟹ fv t = {}" by auto
  hence "⋀t. t ∈ M ⟹ t · s = t" by (metis subst_ground_ident)
  moreover have "⋀t. t ∈ M ⟹ t · s ∈ M ·set s" by (metis imageI)
  ultimately show "M ·set s = M" by (simp add: image_cong)
qed

lemma subst_eqI[intro]: "(⋀t. t · σ = t · ϑ) ⟹ σ = ϑ"
proof -
  assume "⋀t. t · σ = t · ϑ"
  hence "⋀v. Var v · σ = Var v · ϑ" by auto
  thus "σ = ϑ" by auto
qed

lemma subst_cong: "⟦σ = σ'; ϑ = ϑ'⟧ ⟹ (σ ∘s ϑ) = (σ' ∘s ϑ')"
by auto

lemma subst_mgt_bot[simp]: "Var ≼∘ ϑ"
by simp

lemma subst_mgt_refl[simp]: "ϑ ≼∘ ϑ"
by (metis var_comp(1))

lemma subst_mgt_trans: "⟦ϑ ≼∘ δ; δ ≼∘ σ⟧ ⟹ ϑ ≼∘ σ"
by (metis subst_compose_assoc)

lemma subst_mgt_comp: "ϑ ≼∘ ϑ ∘s δ"
by auto
```

```
lemma subst_mgt_comp': "ϑ ∘s δ ≼∘ σ ⟹ ϑ ≼∘ σ"
by (metis subst_compose_assoc)

lemma var_self: "(λw. if w = v then Var v else Var w) = Var"
using subst_agreement by auto

lemma var_same[simp]: "Var(v := t) = Var ⟷ t = Var v"
by (intro iffI, metis fun_upd_same, simp add: var_self)

lemma subst_eq_if_eq_vars: "(⋀v. (Var v) · ϑ = (Var v) · σ) ⟹ ϑ = σ"
by (auto simp add: subst_agreement)

lemma subst_all_empty[simp]: "{} ·set ϑ = {}"
by simp

lemma subst_all_insert:"(insert t M) ·set δ = insert (t · δ) (M ·set δ)"
by auto

lemma subst_apply_fv_subset: "fv t ⊆ V ⟹ fv (t · δ) ⊆ fvset (δ ` V)"
by (induct t) auto

lemma subst_apply_fv_empty:
  assumes "fv t = {}"
  shows "fv (t · σ) = {}"
using assms subst_apply_fv_subset[of t "{}" σ]
by auto

lemma subst_compose_fv:
  assumes "fv (ϑ x) = {}"
  shows "fv ((ϑ ∘s σ) x) = {}"
using assms subst_apply_fv_empty
unfolding subst_compose_def by fast

lemma subst_compose_fv':
  fixes ϑ σ::"('a,'b) subst"
  assumes "y ∈ fv ((ϑ ∘s σ) x)"
  shows "∃z. z ∈ fv (ϑ x)"
using assms subst_compose_fv
by fast

lemma subst_apply_fv_unfold: "fv (t · δ) = fvset (δ ` fv t)"
by (induct t) auto

lemma subst_apply_fv_unfold': "fv (t · δ) = (⋃v ∈ fv t. fv (δ v))"
using subst_apply_fv_unfold by simp

lemma subst_apply_fv_union: "fvset (δ ` V) ∪ fv (t · δ) = fvset (δ ` (V ∪ fv t))"
proof -
  have "fvset (δ ` (V ∪ fv t)) = fvset (δ ` V) ∪ fvset (δ ` fv t)" by auto
  thus ?thesis using subst_apply_fv_unfold by metis
qed

lemma subst_elimI[intro]: "(⋀t. v ∉ fv (t · σ)) ⟹ subst_elim σ v"
by (auto simp add: subst_elim_def)

lemma subst_elimI'[intro]: "(⋀w. v ∉ fv (Var w · ϑ)) ⟹ subst_elim ϑ v"
by (simp add: subst_elim_def subst_apply_fv_unfold')

lemma subst_elimD[dest]: "subst_elim σ v ⟹ v ∉ fv (t · σ)"
by (auto simp add: subst_elim_def)

lemma subst_elimD'[dest]: "subst_elim σ v ⟹ σ v ≠ Var v"
by (metis subst_elim_def subst_apply_term.simps(1) term.set_intros(3))
```

```
lemma subst_elimD''[dest]: "subst_elim σ v ⟹ v ∉ fv (σ w)"
by (metis subst_elim_def subst_apply_term.simps(1))

lemma subst_elim_rm_vars_dest[dest]:
  "subst_elim (σ::('a,'b) subst) v ⟹ v ∉ vs ⟹ subst_elim (rm_vars vs σ) v"
proof -
  assume assms: "subst_elim σ v" "v ∉ vs"
  obtain f::"('a, 'b) subst ⇒ 'b ⇒ 'b" where
      "∀σ v. (∃w. v ∈ fv (Var w · σ)) = (v ∈ fv (Var (f σ v) · σ))"
    by moura
  hence *: "∀a σ. a ∈ fv (Var (f σ a) · σ) ∨ subst_elim σ a" by blast
  have "Var (f (rm_vars vs σ) v) · σ ≠ Var (f (rm_vars vs σ) v) · rm_vars vs σ
        ∨ v ∉ fv (Var (f (rm_vars vs σ) v) · rm_vars vs σ)"
    using assms(1) by fastforce
  moreover
  { assume "Var (f (rm_vars vs σ) v) · σ ≠ Var (f (rm_vars vs σ) v) · rm_vars vs σ"
    hence "rm_vars vs σ (f (rm_vars vs σ) v) ≠ σ (f (rm_vars vs σ) v)" by auto
    hence "f (rm_vars vs σ) v ∈ vs" by meson
    hence ?thesis using * assms(2) by force
  }
  ultimately show ?thesis using * by blast
qed

lemma occs_subst_elim: "¬Var v ⊏ t ⟹ subst_elim (Var(v := t)) v ∨ (Var(v := t)) = Var"
proof (cases "Var v = t")
  assume "Var v ≠ t" "¬Var v ⊏ t"
  hence "v ∉ fv t" by (simp add: vars_iff_subterm_or_eq)
  thus ?thesis by (auto simp add: subst_remove_var)
qed auto

lemma occs_subst_elim': "¬Var v ⊑ t ⟹ subst_elim (Var(v := t)) v"
proof -
  assume "¬Var v ⊑ t"
  hence "v ∉ fv t" by (auto simp add: vars_iff_subterm_or_eq)
  thus "subst_elim (Var(v := t)) v" by (simp add: subst_elim_def subst_remove_var)
qed

lemma subst_elim_comp: "subst_elim ϑ v ⟹ subst_elim (δ ∘s ϑ) v"
by (auto simp add: subst_elim_def)

lemma var_subst_idem: "subst_idem Var"
by (simp add: subst_idem_def)

lemma var_upd_subst_idem:
  assumes "¬Var v ⊑ t" shows "subst_idem (Var(v := t))"
unfolding subst_idem_def
proof
  let ?ϑ = "Var(v := t)"
  from assms have t_ϑ_id: "t · ?ϑ = t" by blast
  fix s show "s · (?ϑ ∘s ?ϑ) = s · ?ϑ"
    unfolding subst_compose_def
    by (induction s, metis t_ϑ_id fun_upd_def subst_apply_term.simps(1), simp)
qed
```

2.3.2 Lemmata: Domain and Range of Substitutions

```
lemma range_vars_alt_def: "range_vars s ≡ fvset (subst_range s)"
unfolding range_vars_def by simp

lemma subst_dom_var_finite[simp]: "finite (subst_domain Var)" by simp

lemma subst_range_Var[simp]: "subst_range Var = {}" by simp
```

```
lemma range_vars_Var[simp]: "range_vars Var = {}" by fastforce

lemma finite_subst_img_if_finite_dom: "finite (subst_domain σ) ⟹ finite (range_vars σ)"
unfolding range_vars_alt_def by auto

lemma finite_subst_img_if_finite_dom': "finite (subst_domain σ) ⟹ finite (subst_range σ)"
by auto

lemma subst_img_alt_def: "subst_range s = {t. ∃v. s v = t ∧ t ≠ Var v}"
by (auto simp add: subst_domain_def)

lemma subst_fv_img_alt_def: "range_vars s = (⋃t ∈ {t. ∃v. s v = t ∧ t ≠ Var v}. fv t)"
unfolding range_vars_alt_def by (auto simp add: subst_domain_def)

lemma subst_domI[intro]: "σ v ≠ Var v ⟹ v ∈ subst_domain σ"
by (simp add: subst_domain_def)

lemma subst_imgI[intro]: "σ v ≠ Var v ⟹ σ v ∈ subst_range σ"
by (simp add: subst_domain_def)

lemma subst_fv_imgI[intro]: "σ v ≠ Var v ⟹ fv (σ v) ⊆ range_vars σ"
unfolding range_vars_alt_def by auto

lemma subst_domain_subst_Fun_single[simp]:
  "subst_domain (Var(x := Fun f T)) = {x}" (is "?A = ?B")
unfolding subst_domain_def by simp

lemma subst_range_subst_Fun_single[simp]:
  "subst_range (Var(x := Fun f T)) = {Fun f T}" (is "?A = ?B")
by simp

lemma range_vars_subst_Fun_single[simp]:
  "range_vars (Var(x := Fun f T)) = fv (Fun f T)"
unfolding range_vars_alt_def by force

lemma var_renaming_is_Fun_iff:
  assumes "subst_range δ ⊆ range Var"
  shows "is_Fun t = is_Fun (t · δ)"
proof (cases t)
  case (Var x)
  hence "∃y. δ x = Var y" using assms by auto
  thus ?thesis using Var by auto
qed simp

lemma subst_fv_dom_img_subset: "fv t ⊆ subst_domain ϑ ⟹ fv (t · ϑ) ⊆ range_vars ϑ"
unfolding range_vars_alt_def by (induct t) auto

lemma subst_fv_dom_img_subset_set: "fv_set M ⊆ subst_domain ϑ ⟹ fv_set (M ·_set ϑ) ⊆ range_vars ϑ"
proof -
  assume assms: "fv_set M ⊆ subst_domain ϑ"
  obtain f::"'a set ⇒ (('b, 'a) term ⇒ 'a set) ⇒ ('b, 'a) terms ⇒ ('b, 'a) term" where
      "∀x y z. (∃v. v ∈ z ∧ ¬ y v ⊆ x) ⟷ (f x y z ∈ z ∧ ¬ y (f x y z) ⊆ x)"
    by moura
  hence *:
      "∀T g A. (¬ ⋃ (g ` T) ⊆ A ∨ (∀t. t ∉ T ∨ g t ⊆ A)) ∧
              (⋃ (g ` T) ⊆ A ∨ f A g T ∈ T ∧ ¬ g (f A g T) ⊆ A)"
    by (metis (no_types) SUP_le_iff)
  hence **: "∀t. t ∉ M ∨ fv t ⊆ subst_domain ϑ" by (metis (no_types) assms fv_set.simps)
  have "∀t::('b, 'a) term. ∀f T. t ∉ f ` T ∨ (∃t'::('b, 'a) term. t = f t' ∧ t' ∈ T)" by blast
  hence "f (range_vars ϑ) fv (M ·_set ϑ) ∉ M ·_set ϑ ∨
         fv (f (range_vars ϑ) fv (M ·_set ϑ)) ⊆ range_vars ϑ"
    by (metis (full_types) ** subst_fv_dom_img_subset)
```

```
  thus ?thesis by (metis (no_types) * fv_set.simps)
qed

lemma subst_fv_dom_ground_if_ground_img:
  assumes "fv t ⊆ subst_domain s" "ground (subst_range s)"
  shows "fv (t · s) = {}"
using subst_fv_dom_img_subset[OF assms(1)] assms(2) by force

lemma subst_fv_dom_ground_if_ground_img':
  assumes "fv t ⊆ subst_domain s" "⋀x. x ∈ subst_domain s ⟹ fv (s x) = {}"
  shows "fv (t · s) = {}"
using subst_fv_dom_ground_if_ground_img[OF assms(1)] assms(2) by auto

lemma subst_fv_unfold: "fv (t · s) = (fv t - subst_domain s) ∪ fv_set (s ` (fv t ∩ subst_domain s))"
proof (induction t)
  case (Var v) thus ?case
  proof (cases "v ∈ subst_domain s")
    case True thus ?thesis by auto
  next
    case False
    hence "fv (Var v · s) = {v}" "fv (Var v) ∩ subst_domain s = {}" by auto
    thus ?thesis by auto
  qed
next
  case Fun thus ?case by auto
qed

lemma subst_fv_unfold_ground_img: "range_vars s = {} ⟹ fv (t · s) = fv t - subst_domain s"
using subst_fv_unfold[of t s] unfolding range_vars_alt_def by auto

lemma subst_img_update:
  "⟦σ v = Var v; t ≠ Var v⟧ ⟹ range_vars (σ(v := t)) = range_vars σ ∪ fv t"
proof -
  assume "σ v = Var v" "t ≠ Var v"
  hence "(⋃s ∈ {s. ∃w. (σ(v := t)) w = s ∧ s ≠ Var w}. fv s) = fv t ∪ range_vars σ"
    unfolding range_vars_alt_def by (auto simp add: subst_domain_def)
  thus "range_vars (σ(v := t)) = range_vars σ ∪ fv t"
    by (metis Un_commute subst_fv_img_alt_def)
qed

lemma subst_dom_update1: "v ∉ subst_domain σ ⟹ subst_domain (σ(v := Var v)) = subst_domain σ"
by (auto simp add: subst_domain_def)

lemma subst_dom_update2: "t ≠ Var v ⟹ subst_domain (σ(v := t)) = insert v (subst_domain σ)"
by (auto simp add: subst_domain_def)

lemma subst_dom_update3: "t = Var v ⟹ subst_domain (σ(v := t)) = subst_domain σ - {v}"
by (auto simp add: subst_domain_def)

lemma var_not_in_subst_dom[elim]: "v ∉ subst_domain s ⟹ s v = Var v"
by (simp add: subst_domain_def)

lemma subst_dom_vars_in_subst[elim]: "v ∈ subst_domain s ⟹ s v ≠ Var v"
by (simp add: subst_domain_def)

lemma subst_not_dom_fixed: "⟦v ∈ fv t; v ∉ subst_domain s⟧ ⟹ v ∈ fv (t · s)" by (induct t) auto

lemma subst_not_img_fixed: "⟦v ∈ fv (t · s); v ∉ range_vars s⟧ ⟹ v ∈ fv t"
unfolding range_vars_alt_def by (induct t) force+

lemma ground_range_vars[intro]: "ground (subst_range s) ⟹ range_vars s = {}"
unfolding range_vars_alt_def by metis
```

```
lemma ground_subst_no_var[intro]: "ground (subst_range s) ⟹ x ∉ range_vars s"
using ground_range_vars[of s] by blast

lemma ground_img_obtain_fun:
  assumes "ground (subst_range s)" "x ∈ subst_domain s"
  obtains f T where "s x = Fun f T" "Fun f T ∈ subst_range s" "fv (Fun f T) = {}"
proof -
  from assms(2) obtain t where t: "s x = t" "t ∈ subst_range s" by moura
  hence "fv t = {}" using assms(1) by auto
  thus ?thesis using t that by (cases t) simp_all
qed

lemma ground_term_subst_domain_fv_subset:
  "fv (t · δ) = {} ⟹ fv t ⊆ subst_domain δ"
by (induct t) auto

lemma ground_subst_range_empty_fv:
  "ground (subst_range ϑ) ⟹ x ∈ subst_domain ϑ ⟹ fv (ϑ x) = {}"
by simp

lemma subst_Var_notin_img: "x ∉ range_vars s ⟹ t · s = Var x ⟹ t = Var x"
using subst_not_img_fixed[of x t s] by (induct t) auto

lemma fv_in_subst_img: "⟦s v = t; t ≠ Var v⟧ ⟹ fv t ⊆ range_vars s"
unfolding range_vars_alt_def by auto

lemma empty_dom_iff_empty_subst: "subst_domain ϑ = {} ⟷ ϑ = Var" by auto

lemma subst_dom_cong: "(⋀v t. ϑ v = t ⟹ δ v = t) ⟹ subst_domain ϑ ⊆ subst_domain δ"
by (auto simp add: subst_domain_def)

lemma subst_img_cong: "(⋀v t. ϑ v = t ⟹ δ v = t) ⟹ range_vars ϑ ⊆ range_vars δ"
unfolding range_vars_alt_def by (auto simp add: subst_domain_def)

lemma subst_dom_elim: "subst_domain s ∩ range_vars s = {} ⟹ fv (t · s) ∩ subst_domain s = {}"
proof (induction t)
  case (Var v) thus ?case
    using fv_in_subst_img[of s]
    by (cases "s v = Var v") (auto simp add: subst_domain_def)
next
  case Fun thus ?case by auto
qed

lemma subst_dom_insert_finite: "finite (subst_domain s) = finite (subst_domain (s(v := t)))"
proof
  assume "finite (subst_domain s)"
  have "subst_domain (s(v := t)) ⊆ insert v (subst_domain s)" by (auto simp add: subst_domain_def)
  thus "finite (subst_domain (s(v := t)))"
    by (meson ‹finite (subst_domain s)› finite_insert rev_finite_subset)
next
  assume *: "finite (subst_domain (s(v := t)))"
  hence "finite (insert v (subst_domain s))"
  proof (cases "t = Var v")
    case True
    hence "finite (subst_domain s - {v})" by (metis * subst_dom_update3)
    thus ?thesis by simp
  qed (metis * subst_dom_update2[of t v s])
  thus "finite (subst_domain s)" by simp
qed

lemma trm_subst_disj: "t · ϑ = t ⟹ fv t ∩ subst_domain ϑ = {}"
proof (induction t)
  case (Fun f X)
```

```
    hence "map (λx. x · ϑ) X = X" by simp
    hence "⋀x. x ∈ set X ⟹ x · ϑ = x" using map_eq_conv by fastforce
    thus ?case using Fun.IH by auto
  qed (simp add: subst_domain_def)

lemma trm_subst_ident[intro]: "fv t ∩ subst_domain ϑ = {} ⟹ t · ϑ = t"
proof -
  assume "fv t ∩ subst_domain ϑ = {}"
  hence "∀v ∈ fv t. ∀w ∈ subst_domain ϑ. v ≠ w" by auto
  thus ?thesis
    by (metis subst_agreement subst_apply_term.simps(1) subst_apply_term_empty subst_domI)
qed

lemma trm_subst_ident'[intro]: "v ∉ subst_domain ϑ ⟹ (Var v) · ϑ = Var v"
using trm_subst_ident by (simp add: subst_domain_def)

lemma trm_subst_ident''[intro]: "(⋀x. x ∈ fv t ⟹ ϑ x = Var x) ⟹ t · ϑ = t"
proof -
  assume "⋀x. x ∈ fv t ⟹ ϑ x = Var x"
  hence "fv t ∩ subst_domain ϑ = {}" by (auto simp add: subst_domain_def)
  thus ?thesis using trm_subst_ident by auto
qed

lemma set_subst_ident: "fv_set M ∩ subst_domain ϑ = {} ⟹ M ·set ϑ = M"
proof -
  assume "fv_set M ∩ subst_domain ϑ = {}"
  hence "∀t ∈ M. t · ϑ = t" by auto
  thus ?thesis by force
qed

lemma trm_subst_ident_subterms[intro]:
  "fv t ∩ subst_domain ϑ = {} ⟹ subterms t ·set ϑ = subterms t"
using set_subst_ident[of "subterms t" ϑ] fv_subterms[of t] by simp

lemma trm_subst_ident_subterms'[intro]:
  "v ∉ fv t ⟹ subterms t ·set Var(v := s) = subterms t"
using trm_subst_ident_subterms[of t "Var(v := s)"]
by (meson subst_no_occs trm_subst_disj vars_iff_subtermeq)

lemma const_mem_subst_cases:
  assumes "Fun c [] ∈ M ·set ϑ"
  shows "Fun c [] ∈ M ∨ Fun c [] ∈ ϑ ` fv_set M"
proof -
  obtain m where m: "m ∈ M" "m · ϑ = Fun c []" using assms by auto
  thus ?thesis by (cases m) force+
qed

lemma const_mem_subst_cases':
  assumes "Fun c [] ∈ M ·set ϑ"
  shows "Fun c [] ∈ M ∨ Fun c [] ∈ subst_range ϑ"
using const_mem_subst_cases[OF assms] by force

lemma fv_subterms_substI[intro]: "y ∈ fv t ⟹ ϑ y ∈ subterms t ·set ϑ"
using image_iff vars_iff_subtermeq by fastforce

lemma fv_subterms_subst_eq[simp]: "fv_set (subterms (t · ϑ)) = fv_set (subterms t ·set ϑ)"
using fv_subterms by (induct t) force+

lemma fv_subterms_set_subst: "fv_set (subterms_set M ·set ϑ) = fv_set (subterms_set (M ·set ϑ))"
using fv_subterms_subst_eq[of _ ϑ] by auto

lemma fv_subterms_set_subst': "fv_set (subterms_set M ·set ϑ) = fv_set (M ·set ϑ)"
using fv_subterms_set[of "M ·set ϑ"] fv_subterms_set_subst[of ϑ M] by simp
```

```
lemma fv_subst_subset: "x ∈ fv t ⟹ fv (ϑ x) ⊆ fv (t · ϑ)"
by (metis fv_subset image_eqI subst_apply_fv_unfold)

lemma fv_subst_subset': "fv s ⊆ fv t ⟹ fv (s · ϑ) ⊆ fv (t · ϑ)"
using fv_subst_subset by (induct s) force+

lemma fv_subst_obtain_var:
  fixes δ::"('a,'b) subst"
  assumes "x ∈ fv (t · δ)"
  shows "∃y ∈ fv t. x ∈ fv (δ y)"
using assms by (induct t) force+

lemma set_subst_all_ident: "fv_set (M ·set ϑ) ∩ subst_domain δ = {} ⟹ M ·set (ϑ ∘s δ) = M ·set ϑ"
by (metis set_subst_ident subst_comp_all)

lemma subterms_subst:
  "subterms (t · d) = (subterms t ·set d) ∪ subterms_set (d ` (fv t ∩ subst_domain d))"
by (induct t) (auto simp add: subst_domain_def)

lemma subterms_subst':
  fixes ϑ::"('a,'b) subst"
  assumes "∀x ∈ fv t. (∃f. ϑ x = Fun f []) ∨ (∃y. ϑ x = Var y)"
  shows "subterms (t · ϑ) = subterms t ·set ϑ"
using assms
proof (induction t)
  case (Var x) thus ?case
  proof (cases "x ∈ subst_domain ϑ")
    case True
    hence "(∃f. ϑ x = Fun f []) ∨ (∃y. ϑ x = Var y)" using Var by simp
    hence "subterms (ϑ x) = {ϑ x}" by auto
    thus ?thesis by simp
  qed auto
qed auto

lemma subterms_subst'':
  fixes ϑ::"('a,'b) subst"
  assumes "∀x ∈ fv_set M. (∃f. ϑ x = Fun f []) ∨ (∃y. ϑ x = Var y)"
  shows "subterms_set (M ·set ϑ) = subterms_set M ·set ϑ"
using subterms_subst'[of _ ϑ] assms by auto

lemma subterms_subst_subterm:
  fixes ϑ::"('a,'b) subst"
  assumes "∀x ∈ fv a. (∃f. ϑ x = Fun f []) ∨ (∃y. ϑ x = Var y)"
    and "b ∈ subterms (a · ϑ)"
  shows "∃c ∈ subterms a. c · ϑ = b"
using subterms_subst'[OF assms(1)] assms(2) by auto

lemma subterms_subst_subset: "subterms t ·set σ ⊆ subterms (t · σ)"
by (induct t) auto

lemma subterms_subst_subset': "subterms_set M ·set σ ⊆ subterms_set (M ·set σ)"
using subterms_subst_subset by fast

lemma subterms_set_subst:
  fixes ϑ::"('a,'b) subst"
  assumes "t ∈ subterms_set (M ·set ϑ)"
  shows "t ∈ subterms_set M ·set ϑ ∨ (∃x ∈ fv_set M. t ∈ subterms (ϑ x))"
using assms subterms_subst[of _ ϑ] by auto

lemma rm_vars_dom: "subst_domain (rm_vars V s) = subst_domain s - V"
by (auto simp add: subst_domain_def)
```

```
lemma rm_vars_dom_subset: "subst_domain (rm_vars V s) ⊆ subst_domain s"
by (auto simp add: subst_domain_def)

lemma rm_vars_dom_eq':
  "subst_domain (rm_vars (UNIV - V) s) = subst_domain s ∩ V"
using rm_vars_dom[of "UNIV - V" s] by blast

lemma rm_vars_img: "subst_range (rm_vars V s) = s ` subst_domain (rm_vars V s)"
by (auto simp add: subst_domain_def)

lemma rm_vars_img_subset: "subst_range (rm_vars V s) ⊆ subst_range s"
by (auto simp add: subst_domain_def)

lemma rm_vars_img_fv_subset: "range_vars (rm_vars V s) ⊆ range_vars s"
unfolding range_vars_alt_def by (auto simp add: subst_domain_def)

lemma rm_vars_fv_obtain:
  assumes "x ∈ fv (t · rm_vars X ϑ) - X"
  shows "∃y ∈ fv t - X. x ∈ fv (rm_vars X ϑ y)"
using assms by (induct t) (fastforce, force)

lemma rm_vars_apply: "v ∈ subst_domain (rm_vars V s) ⟹ (rm_vars V s) v = s v"
by (auto simp add: subst_domain_def)

lemma rm_vars_apply': "subst_domain δ ∩ vs = {} ⟹ rm_vars vs δ = δ"
by force

lemma rm_vars_ident: "fv t ∩ vs = {} ⟹ t · (rm_vars vs ϑ) = t · ϑ"
by (induct t) auto

lemma rm_vars_fv_subset: "fv (t · rm_vars X ϑ) ⊆ fv t ∪ fv (t · ϑ)"
by (induct t) auto

lemma rm_vars_fv_disj:
  assumes "fv t ∩ X = {}" "fv (t · ϑ) ∩ X = {}"
  shows "fv (t · rm_vars X ϑ) ∩ X = {}"
using rm_vars_ident[OF assms(1)] assms(2) by auto

lemma rm_vars_ground_supports:
  assumes "ground (subst_range ϑ)"
  shows "rm_vars X ϑ supports ϑ"
proof
  fix x
  have *: "ground (subst_range (rm_vars X ϑ))"
    using rm_vars_img_subset[of X ϑ] assms
    by (auto simp add: subst_domain_def)
  show "rm_vars X ϑ x · ϑ = ϑ x "
  proof (cases "x ∈ subst_domain (rm_vars X ϑ)")
    case True
    hence "fv (rm_vars X ϑ x) = {}" using * by auto
    thus ?thesis using True by auto
  qed (simp add: subst_domain_def)
qed

lemma rm_vars_split:
  assumes "ground (subst_range ϑ)"
  shows "ϑ = rm_vars X ϑ ∘s rm_vars (subst_domain ϑ - X) ϑ"
proof -
  let ?s1 = "rm_vars X ϑ"
  let ?s2 = "rm_vars (subst_domain ϑ - X) ϑ"

  have doms: "subst_domain ?s1 ⊆ subst_domain ϑ" "subst_domain ?s2 ⊆ subst_domain ϑ"
    by (auto simp add: subst_domain_def)
```

```
  { fix x assume "x ∉ subst_domain ϑ"
    hence "ϑ x = Var x" "?s1 x = Var x" "?s2 x = Var x" using doms by auto
    hence "ϑ x = (?s1 ∘s ?s2) x" by (simp add: subst_compose_def)
  } moreover {
    fix x assume "x ∈ subst_domain ϑ" "x ∈ X"
    hence "?s1 x = Var x" "?s2 x = ϑ x" using doms by auto
    hence "ϑ x = (?s1 ∘s ?s2) x" by (simp add: subst_compose_def)
  } moreover {
    fix x assume "x ∈ subst_domain ϑ" "x ∉ X"
    hence "?s1 x = ϑ x" "fv (ϑ x) = {}" using assms doms by auto
    hence "ϑ x = (?s1 ∘s ?s2) x" by (simp add: subst_compose subst_ground_ident)
  } ultimately show ?thesis by blast
qed

lemma rm_vars_fv_img_disj:
  assumes "fv t ∩ X = {}" "X ∩ range_vars ϑ = {}"
  shows "fv (t · rm_vars X ϑ) ∩ X = {}"
using assms
proof (induction t)
  case (Var x)
  hence *: "(rm_vars X ϑ) x = ϑ x" by auto
  show ?case
  proof (cases "x ∈ subst_domain ϑ")
    case True
    hence "ϑ x ∈ subst_range ϑ" by auto
    hence "fv (ϑ x) ∩ X = {}" using Var.prems(2) unfolding range_vars_alt_def by fastforce
    thus ?thesis using * by auto
  next
    case False thus ?thesis using Var.prems(1) by auto
  qed
next
  case Fun thus ?case by auto
qed

lemma subst_apply_dom_ident: "t · ϑ = t ⟹ subst_domain δ ⊆ subst_domain ϑ ⟹ t · δ = t"
proof (induction t)
  case (Fun f T) thus ?case by (induct T) auto
qed (auto simp add: subst_domain_def)

lemma rm_vars_subst_apply_ident:
  assumes "t · ϑ = t"
  shows "t · (rm_vars vs ϑ) = t"
using rm_vars_dom[of vs ϑ] subst_apply_dom_ident[OF assms, of "rm_vars vs ϑ"] by auto

lemma rm_vars_subst_eq:
  "t · δ = t · rm_vars (subst_domain δ - subst_domain δ ∩ fv t) δ"
by (auto intro: term_subst_eq)

lemma rm_vars_subst_eq':
  "t · δ = t · rm_vars (UNIV - fv t) δ"
by (auto intro: term_subst_eq)

lemma rm_vars_comp:
  assumes "range_vars δ ∩ vs = {}"
  shows "t · rm_vars vs (δ ∘s ϑ) = t · (rm_vars vs δ ∘s rm_vars vs ϑ)"
using assms
proof (induction t)
  case (Var x) thus ?case
  proof (cases "x ∈ vs")
    case True thus ?thesis using Var by auto
  next
    case False
```

```
    have "subst_domain (rm_vars vs ϑ) ∩ vs = {}" by (auto simp add: subst_domain_def)
    moreover have "fv (δ x) ∩ vs = {}"
      using Var False unfolding range_vars_alt_def by force
    ultimately have "δ x · (rm_vars vs ϑ) = δ x · ϑ"
      using rm_vars_ident by (simp add: subst_domain_def)
    moreover have "(rm_vars vs (δ ∘s ϑ)) x = (δ ∘s ϑ) x" by (metis False)
    ultimately show ?thesis using subst_compose by auto
  qed
next
  case Fun thus ?case by auto
qed

lemma rm_vars_fv_set_subst:
  assumes "x ∈ fv_set (rm_vars X ϑ ` Y)"
  shows "x ∈ fv_set (ϑ ` Y) ∨ x ∈ X"
using assms by auto

lemma disj_dom_img_var_notin:
  assumes "subst_domain ϑ ∩ range_vars ϑ = {}" "ϑ v = t" "t ≠ Var v"
  shows "v ∉ fv t" "∀v ∈ fv (t · ϑ). v ∉ subst_domain ϑ"
proof -
  have "v ∈ subst_domain ϑ" "fv t ⊆ range_vars ϑ"
    using fv_in_subst_img[of ϑ v t, OF assms(2)] assms(2,3)
    by (auto simp add: subst_domain_def)
  thus "v ∉ fv t" using assms(1) by auto

  have *: "fv t ∩ subst_domain ϑ = {}"
    using assms(1) ‹fv t ⊆ range_vars ϑ›
    by auto
  hence "t · ϑ = t" by blast
  thus "∀v ∈ fv (t · ϑ). v ∉ subst_domain ϑ" using * by auto
qed

lemma subst_sends_dom_to_img: "v ∈ subst_domain ϑ ⟹ fv (Var v · ϑ) ⊆ range_vars ϑ"
unfolding range_vars_alt_def by auto

lemma subst_sends_fv_to_img: "fv (t · s) ⊆ fv t ∪ range_vars s"
proof (induction t)
  case (Var v) thus ?case
  proof (cases "Var v · s = Var v")
    case True thus ?thesis by simp
  next
    case False
    hence "v ∈ subst_domain s" by (meson trm_subst_ident')
    hence "fv (Var v · s) ⊆ range_vars s"
      using subst_sends_dom_to_img by simp
    thus ?thesis by auto
  qed
next
  case Fun thus ?case by auto
qed

lemma ident_comp_subst_trm_if_disj:
  assumes "subst_domain σ ∩ range_vars ϑ = {}" "v ∈ subst_domain ϑ"
  shows "(ϑ ∘s σ) v = ϑ v"
proof -
  from assms have " subst_domain σ ∩ fv (ϑ v) = {}"
    using fv_in_subst_img unfolding range_vars_alt_def by auto
  thus "(ϑ ∘s σ) v = ϑ v" unfolding subst_compose_def by blast
qed

lemma ident_comp_subst_trm_if_disj': "fv (ϑ v) ∩ subst_domain σ = {} ⟹ (ϑ ∘s σ) v = ϑ v"
unfolding subst_compose_def by blast
```

```
lemma subst_idemI[intro]: "subst_domain σ ∩ range_vars σ = {} ⟹ subst_idem σ"
using ident_comp_subst_trm_if_disj[of σ σ]
      var_not_in_subst_dom[of _ σ]
      subst_eq_if_eq_vars[of σ]
by (metis subst_idem_def subst_compose_def var_comp(2))

lemma subst_idemI'[intro]: "ground (subst_range σ) ⟹ subst_idem σ"
proof (intro subst_idemI)
  assume "ground (subst_range σ)"
  hence "range_vars σ = {}" by (metis ground_range_vars)
  thus "subst_domain σ ∩ range_vars σ = {}" by blast
qed

lemma subst_idemE: "subst_idem σ ⟹ subst_domain σ ∩ range_vars σ = {}"
proof -
  assume "subst_idem σ"
  hence "⋀v. fv (σ v) ∩ subst_domain σ = {}"
    unfolding subst_idem_def subst_compose_def by (metis trm_subst_disj)
  thus ?thesis
    unfolding range_vars_alt_def by auto
qed

lemma subst_idem_rm_vars: "subst_idem ϑ ⟹ subst_idem (rm_vars X ϑ)"
proof -
  assume "subst_idem ϑ"
  hence "subst_domain ϑ ∩ range_vars ϑ = {}" by (metis subst_idemE)
  moreover have
      "subst_domain (rm_vars X ϑ) ⊆ subst_domain ϑ"
      "range_vars (rm_vars X ϑ) ⊆ range_vars ϑ"
    unfolding range_vars_alt_def by (auto simp add: subst_domain_def)
  ultimately show ?thesis by blast
qed

lemma subst_fv_bounded_if_img_bounded: "range_vars ϑ ⊆ fv t ∪ V ⟹ fv (t · ϑ) ⊆ fv t ∪ V"
proof (induction t)
  case (Var v) thus ?case unfolding range_vars_alt_def by (cases "ϑ v = Var v") auto
qed (metis (no_types, lifting) Un_assoc Un_commute subst_sends_fv_to_img sup.absorb_iff2)

lemma subst_fv_bound_singleton: "fv (t · Var(v := t')) ⊆ fv t ∪ fv t'"
using subst_fv_bounded_if_img_bounded[of "Var(v := t')" t "fv t'"]
unfolding range_vars_alt_def by (auto simp add: subst_domain_def)

lemma subst_fv_bounded_if_img_bounded':
  assumes "range_vars ϑ ⊆ fv_set M"
  shows "fv_set (M ·_set ϑ) ⊆ fv_set M"
proof
  fix v assume *:  "v ∈ fv_set (M ·_set ϑ)"

  obtain t where t: "t ∈ M" "t · ϑ ∈ M ·_set ϑ" "v ∈ fv (t · ϑ)"
  proof -
    assume **: "⋀t. ⟦t ∈ M; t · ϑ ∈ M ·_set ϑ; v ∈ fv (t · ϑ)⟧ ⟹ thesis"
    have "v ∈ ⋃ (fv ` ((λt. t · ϑ) ` M))" using * by (metis fv_set.simps)
    hence "∃t. t ∈ M ∧ v ∈ fv (t · ϑ)" by blast
    thus ?thesis using ** imageI by blast
  qed

  from ‹t ∈ M› obtain M' where "t ∉ M'" "M = insert t M'" by (meson Set.set_insert)
  hence "fv_set M = fv t ∪ fv_set M'" by simp
  hence "fv (t · ϑ) ⊆ fv_set M" using subst_fv_bounded_if_img_bounded assms by simp
  thus "v ∈ fv_set M" using assms ‹v ∈ fv (t · ϑ)› by auto
qed
```

```
lemma ground_img_if_ground_subst: "(⋀v t. s v = t ⟹ fv t = {}) ⟹ range_vars s = {}"
unfolding range_vars_alt_def by auto

lemma ground_subst_fv_subset: "ground (subst_range ϑ) ⟹ fv (t · ϑ) ⊆ fv t"
using subst_fv_bounded_if_img_bounded[of ϑ]
unfolding range_vars_alt_def by force

lemma ground_subst_fv_subset': "ground (subst_range ϑ) ⟹ fv_set (M ·_set ϑ) ⊆ fv_set M"
using subst_fv_bounded_if_img_bounded'[of ϑ M]
unfolding range_vars_alt_def by auto

lemma subst_to_var_is_var[elim]: "t · s = Var v ⟹ ∃w. t = Var w"
using subst_apply_term.elims by blast

lemma subst_dom_comp_inI:
  assumes "y ∉ subst_domain σ"
    and "y ∈ subst_domain δ"
  shows "y ∈ subst_domain (σ ∘_s δ)"
using assms subst_domain_subst_compose[of σ δ] by blast

lemma subst_comp_notin_dom_eq:
  "x ∉ subst_domain ϑ1 ⟹ (ϑ1 ∘_s ϑ2) x = ϑ2 x"
unfolding subst_compose_def by fastforce

lemma subst_dom_comp_eq:
  assumes "subst_domain ϑ ∩ range_vars σ = {}"
  shows "subst_domain (ϑ ∘_s σ) = subst_domain ϑ ∪ subst_domain σ"
proof (rule ccontr)
  assume "subst_domain (ϑ ∘_s σ) ≠ subst_domain ϑ ∪ subst_domain σ"
  hence "subst_domain (ϑ ∘_s σ) ⊂ subst_domain ϑ ∪ subst_domain σ"
    using subst_domain_compose[of ϑ σ] by (simp add: subst_domain_def)
  then obtain v where "v ∉ subst_domain (ϑ ∘_s σ)" "v ∈ subst_domain ϑ ∪ subst_domain σ" by auto
  hence v_in_some_subst: "ϑ v ≠ Var v ∨ σ v ≠ Var v" and "ϑ v · σ = Var v"
    unfolding subst_compose_def by (auto simp add: subst_domain_def)
  then obtain w where "ϑ v = Var w" using subst_to_var_is_var by fastforce
  show False
  proof (cases "v = w")
    case True
    hence "ϑ v = Var v" using ‹ϑ v = Var w› by simp
    hence "σ v ≠ Var v" using v_in_some_subst by simp
    thus False using ‹ϑ v = Var v› ‹ϑ v · σ = Var v› by simp
  next
    case False
    hence "v ∈ subst_domain ϑ" using v_in_some_subst ‹ϑ v · σ = Var v› by auto
    hence "v ∉ range_vars σ" using assms by auto
    moreover have "σ w = Var v" using ‹ϑ v · σ = Var v› ‹ϑ v = Var w› by simp
    hence "v ∈ range_vars σ" using ‹v ≠ w› subst_fv_imgI[of σ w] by simp
    ultimately show False ..
  qed
qed

lemma subst_img_comp_subset[simp]:
  "range_vars (ϑ1 ∘_s ϑ2) ⊆ range_vars ϑ1 ∪ range_vars ϑ2"
proof
  let ?img = "range_vars"
  fix x assume "x ∈ ?img (ϑ1 ∘_s ϑ2)"
  then obtain v t where vt: "x ∈ fv t" "t = (ϑ1 ∘_s ϑ2) v" "t ≠ Var v"
    unfolding range_vars_alt_def subst_compose_def by (auto simp add: subst_domain_def)

  { assume "x ∉ ?img ϑ1" hence "x ∈ ?img ϑ2"
      by (metis (no_types, hide_lams) fv_in_subst_img Un_iff subst_compose_def
                vt subsetCE subst_apply_term.simps(1) subst_sends_fv_to_img)
  }
```

```
  thus "x ∈ ?img ϑ1 ∪ ?img ϑ2" by auto
qed

lemma subst_img_comp_subset':
  assumes "t ∈ subst_range (ϑ1 ∘s ϑ2)"
  shows "t ∈ subst_range ϑ2 ∨ (∃t' ∈ subst_range ϑ1. t = t' · ϑ2)"
proof -
  obtain x where x: "x ∈ subst_domain (ϑ1 ∘s ϑ2)" "(ϑ1 ∘s ϑ2) x = t" "t ≠ Var x"
    using assms by (auto simp add: subst_domain_def)
  { assume "x ∉ subst_domain ϑ1"
    hence "(ϑ1 ∘s ϑ2) x = ϑ2 x" unfolding subst_compose_def by auto
    hence ?thesis using x by auto
  } moreover {
    assume "x ∈ subst_domain ϑ1" hence ?thesis using subst_compose x(2) by fastforce
  } ultimately show ?thesis by metis
qed

lemma subst_img_comp_subset'':
  "subterms_set (subst_range (ϑ1 ∘s ϑ2)) ⊆
   subterms_set (subst_range ϑ2) ∪ ((subterms_set (subst_range ϑ1)) ·set ϑ2)"
proof
  fix t assume "t ∈ subterms_set (subst_range (ϑ1 ∘s ϑ2))"
  then obtain x where x: "x ∈ subst_domain (ϑ1 ∘s ϑ2)" "t ∈ subterms ((ϑ1 ∘s ϑ2) x)"
    by auto
  show "t ∈ subterms_set (subst_range ϑ2) ∪ (subterms_set (subst_range ϑ1) ·set ϑ2)"
  proof (cases "x ∈ subst_domain ϑ1")
    case True thus ?thesis
      using subst_compose[of ϑ1 ϑ2] x(2) subterms_subst
      by fastforce
  next
    case False
    hence "(ϑ1 ∘s ϑ2) x = ϑ2 x" unfolding subst_compose_def by auto
    thus ?thesis using x by (auto simp add: subst_domain_def)
  qed
qed

lemma subst_img_comp_subset''':
  "subterms_set (subst_range (ϑ1 ∘s ϑ2)) - range Var ⊆
   subterms_set (subst_range ϑ2) - range Var ∪ ((subterms_set (subst_range ϑ1) - range Var) ·set ϑ2)"
proof
  fix t assume t: "t ∈ subterms_set (subst_range (ϑ1 ∘s ϑ2)) - range Var"
  then obtain f T where fT: "t = Fun f T" by (cases t) simp_all
  then obtain x where x: "x ∈ subst_domain (ϑ1 ∘s ϑ2)" "Fun f T ∈ subterms ((ϑ1 ∘s ϑ2) x)"
    using t by auto
  have "Fun f T ∈ subterms_set (subst_range ϑ2) ∪ (subterms_set (subst_range ϑ1) - range Var ·set ϑ2)"
  proof (cases "x ∈ subst_domain ϑ1")
    case True
    hence "Fun f T ∈ (subterms_set (subst_range ϑ2)) ∪ (subterms (ϑ1 x) ·set ϑ2)"
      using x(2) subterms_subst[of "ϑ1 x" ϑ2]
      unfolding subst_compose[of ϑ1 ϑ2 x] by auto
    moreover have ?thesis when *: "Fun f T ∈ subterms (ϑ1 x) ·set ϑ2"
    proof -
      obtain s where s: "s ∈ subterms (ϑ1 x)" "Fun f T = s · ϑ2" using * by moura
      show ?thesis
      proof (cases s)
        case (Var y)
        hence "Fun f T ∈ subst_range ϑ2" using s by force
        thus ?thesis by blast
      next
        case (Fun g S)
        hence "Fun f T ∈ (subterms (ϑ1 x) - range Var) ·set ϑ2" using s by blast
        thus ?thesis using True by auto
      qed
```

```
    qed
    ultimately show ?thesis by blast
  next
    case False
    hence "(ϑ1 ∘s ϑ2) x = ϑ2 x" unfolding subst_compose_def by auto
    thus ?thesis using x by (auto simp add: subst_domain_def)
  qed
  thus "t ∈ subtermsset (subst_range ϑ2) - range Var ∪
          (subtermsset (subst_range ϑ1) - range Var ·set ϑ2)"
    using fT by auto
qed

lemma subst_img_comp_subset_const:
  assumes "Fun c [] ∈ subst_range (ϑ1 ∘s ϑ2)"
  shows "Fun c [] ∈ subst_range ϑ2 ∨ Fun c [] ∈ subst_range ϑ1 ∨
        (∃x. Var x ∈ subst_range ϑ1 ∧ ϑ2 x = Fun c [])"
proof (cases "Fun c [] ∈ subst_range ϑ2")
  case False
  then obtain t where t: "t ∈ subst_range ϑ1" "Fun c [] = t · ϑ2"
    using subst_img_comp_subset'[OF assms] by auto
  thus ?thesis by (cases t) auto
qed (simp add: subst_img_comp_subset'[OF assms])

lemma subst_img_comp_subset_const':
  fixes δ τ::"('f,'v) subst"
  assumes "(δ ∘s τ) x = Fun c []"
  shows "δ x = Fun c [] ∨ (∃z. δ x = Var z ∧ τ z = Fun c [])"
proof (cases "δ x = Fun c []")
  case False
  then obtain t where "δ x = t" "t · τ = Fun c []" using assms unfolding subst_compose_def by auto
  thus ?thesis by (cases t) auto
qed simp

lemma subst_img_comp_subset_ground:
  assumes "ground (subst_range ϑ1)"
  shows "subst_range (ϑ1 ∘s ϑ2) ⊆ subst_range ϑ1 ∪ subst_range ϑ2"
proof
  fix t assume t: "t ∈ subst_range (ϑ1 ∘s ϑ2)"
  then obtain x where x: "x ∈ subst_domain (ϑ1 ∘s ϑ2)" "t = (ϑ1 ∘s ϑ2) x" by auto

  show "t ∈ subst_range ϑ1 ∪ subst_range ϑ2"
  proof (cases "x ∈ subst_domain ϑ1")
    case True
    hence "fv (ϑ1 x) = {}" using assms ground_subst_range_empty_fv by fast
    hence "t = ϑ1 x" using x(2) unfolding subst_compose_def by blast
    thus ?thesis using True by simp
  next
    case False
    hence "t = ϑ2 x" "x ∈ subst_domain ϑ2"
      using x subst_domain_compose[of ϑ1 ϑ2]
      by (metis subst_comp_notin_dom_eq, blast)
    thus ?thesis using x by simp
  qed
qed

lemma subst_fv_dom_img_single:
  assumes "v ∉ fv t" "σ v = t" "⋀w. v ≠ w ⟹ σ w = Var w"
  shows "subst_domain σ = {v}" "range_vars σ = fv t"
proof -
  show "subst_domain σ = {v}" using assms by (fastforce simp add: subst_domain_def)
  have "fv t ⊆ range_vars σ" by (metis fv_in_subst_img assms(1,2) vars_iff_subterm_or_eq)
  moreover have "⋀v. σ v ≠ Var v ⟹ σ v = t" using assms by fastforce
  ultimately show "range_vars σ = fv t"
```

```
    unfolding range_vars_alt_def
    by (auto simp add: subst_domain_def)
qed

lemma subst_comp_upd1:
  "ϑ(v := t) ∘s σ = (ϑ ∘s σ)(v := t · σ)"
unfolding subst_compose_def by auto

lemma subst_comp_upd2:
  assumes "v ∉ subst_domain s" "v ∉ range_vars s"
  shows "s(v := t) = s ∘s (Var(v := t))"
unfolding subst_compose_def
proof -
  { fix w
    have "(s(v := t)) w = s w · Var(v := t)"
    proof (cases "w = v")
      case True
      hence "s w = Var w" using ‹v ∉ subst_domain s› by (simp add: subst_domain_def)
      thus ?thesis using ‹w = v› by simp
    next
      case False
      hence "(s(v := t)) w = s w" by simp
      moreover have "s w · Var(v := t) = s w" using ‹w ≠ v› ‹v ∉ range_vars s›
        by (metis fv_in_subst_img fun_upd_apply insert_absorb insert_subset
                  repl_invariance subst_apply_term.simps(1) subst_apply_term_empty)
      ultimately show ?thesis ..
    qed
  }
  thus "s(v := t) = (λw. s w · Var(v := t))" by auto
qed

lemma ground_subst_dom_iff_img:
  "ground (subst_range σ) ⟹ x ∈ subst_domain σ ⟷ σ x ∈ subst_range σ"
by (auto simp add: subst_domain_def)

lemma finite_dom_subst_exists:
  "finite S ⟹ ∃σ::('f,'v) subst. subst_domain σ = S"
proof (induction S rule: finite.induct)
  case (insertI A a)
  then obtain σ::"('f,'v) subst" where "subst_domain σ = A" by blast
  fix f::'f
  have "subst_domain (σ(a := Fun f [])) = insert a A"
    using ‹subst_domain σ = A›
    by (auto simp add: subst_domain_def)
  thus ?case by metis
qed (auto simp add: subst_domain_def)

lemma subst_inj_is_bij_betw_dom_img_if_ground_img:
  assumes "ground (subst_range σ)"
  shows "inj σ ⟷ bij_betw σ (subst_domain σ) (subst_range σ)" (is "?A ⟷ ?B")
proof
  show "?A ⟹ ?B" by (metis bij_betw_def injD inj_onI subst_range.simps)
next
  assume ?B
  hence "inj_on σ (subst_domain σ)" unfolding bij_betw_def by auto
  moreover have "⋀x. x ∈ UNIV - subst_domain σ ⟹ σ x = Var x" by auto
  hence "inj_on σ (UNIV - subst_domain σ)"
    using inj_onI[of "UNIV - subst_domain σ"]
    by (metis term.inject(1))
  moreover have "⋀x y. x ∈ subst_domain σ ⟹ y ∉ subst_domain σ ⟹ σ x ≠ σ y"
    using assms by (auto simp add: subst_domain_def)
  ultimately show ?A by (metis injI inj_onD subst_domI term.inject(1))
qed
```

```
lemma bij_finite_ground_subst_exists:
  assumes "finite (S::'v set)" "infinite (U::('f,'v) term set)" "ground U"
  shows "∃σ::('f,'v) subst. subst_domain σ = S
                          ∧ bij_betw σ (subst_domain σ) (subst_range σ)
                          ∧ subst_range σ ⊆ U"
proof -
  obtain T' where "T' ⊆ U" "card T' = card S" "finite T'"
    by (meson assms(2) finite_Diff2 infinite_arbitrarily_large)
  then obtain f::"'v ⇒ ('f,'v) term" where f_bij: "bij_betw f S T'"
    using finite_same_card_bij[OF assms(1)] by metis
  hence *: "⋀v. v ∈ S ⟹ f v ≠ Var v"
    using ‹ground U› ‹T' ⊆ U› bij_betwE
    by fastforce

  let ?σ = "λv. if v ∈ S then f v else Var v"
  have "subst_domain ?σ = S"
  proof
    show "subst_domain ?σ ⊆ S" by (auto simp add: subst_domain_def)

    { fix v assume "v ∈ S" "v ∉ subst_domain ?σ"
      hence "f v = Var v" by (simp add: subst_domain_def)
      hence False using *[OF ‹v ∈ S›] by metis
    }
    thus "S ⊆ subst_domain ?σ" by blast
  qed
  hence "⋀v w. ⟦v ∈ subst_domain ?σ; w ∉ subst_domain ?σ⟧ ⟹ ?σ w ≠ ?σ v"
    using ‹ground U› bij_betwE[OF f_bij] set_rev_mp[OF _ ‹T' ⊆ U›]
    by (metis (no_types, lifting) UN_iff empty_iff vars_iff_subterm_or_eq fv_set.simps)
  hence "inj_on ?σ (subst_domain ?σ)"
    using f_bij ‹subst_domain ?σ = S›
    unfolding bij_betw_def inj_on_def
    by metis
  hence "bij_betw ?σ (subst_domain ?σ) (subst_range ?σ)"
    using inj_on_imp_bij_betw[of ?σ] by simp
  moreover have "subst_range ?σ = T'"
    using ‹bij_betw f S T'› ‹subst_domain ?σ = S›
    unfolding bij_betw_def by auto
  hence "subst_range ?σ ⊆ U" using ‹T' ⊆ U› by auto
  ultimately show ?thesis using ‹subst_domain ?σ = S› by (metis (lifting))
qed

lemma bij_finite_const_subst_exists:
  assumes "finite (S::'v set)" "finite (T::'f set)" "infinite (U::'f set)"
  shows "∃σ::('f,'v) subst. subst_domain σ = S
                          ∧ bij_betw σ (subst_domain σ) (subst_range σ)
                          ∧ subst_range σ ⊆ (λc. Fun c []) ` (U - T)"
proof -
  obtain T' where "T' ⊆ U - T" "card T' = card S" "finite T'"
    by (meson assms(2,3) finite_Diff2 infinite_arbitrarily_large)
  then obtain f::"'v ⇒ 'f" where f_bij: "bij_betw f S T'"
    using finite_same_card_bij[OF assms(1)] by metis

  let ?σ = "λv. if v ∈ S then Fun (f v) [] else Var v"
  have "subst_domain ?σ = S" by (simp add: subst_domain_def)
  moreover have "⋀v w. ⟦v ∈ subst_domain ?σ; w ∉ subst_domain ?σ⟧ ⟹ ?σ w ≠ ?σ v" by auto
  hence "inj_on ?σ (subst_domain ?σ)"
    using f_bij unfolding bij_betw_def inj_on_def
    by (metis ‹subst_domain ?σ = S› term.inject(2))
  hence "bij_betw ?σ (subst_domain ?σ) (subst_range ?σ)"
    using inj_on_imp_bij_betw[of ?σ] by simp
  moreover have "subst_range ?σ = ((λc. Fun c []) ` T')"
    using ‹bij_betw f S T'› unfolding bij_betw_def inj_on_def by (auto simp add: subst_domain_def)
```

```
  hence "subst_range ?σ ⊆ ((λc. Fun c []) ` (U - T))" using ‹T' ⊆ U - T› by auto
  ultimately show ?thesis by (metis (lifting))
qed

lemma bij_finite_const_subst_exists':
  assumes "finite (S::'v set)" "finite (T::('f,'v) terms)" "infinite (U::'f set)"
  shows "∃σ::('f,'v) subst. subst_domain σ = S
                          ∧ bij_betw σ (subst_domain σ) (subst_range σ)
                          ∧ subst_range σ ⊆ ((λc. Fun c []) ` U) - T"
proof -
  have "finite (⋃(funs_term ` T))" using assms(2) by auto
  then obtain σ where σ:
      "subst_domain σ = S" "bij_betw σ (subst_domain σ) (subst_range σ)"
      "subst_range σ ⊆ (λc. Fun c []) ` (U - (⋃(funs_term ` T)))"
    using bij_finite_const_subst_exists[OF assms(1) _ assms(3)] by blast
  moreover have "(λc. Fun c []) ` (U - (⋃(funs_term ` T))) ⊆ ((λc. Fun c []) ` U) - T" by auto
  ultimately show ?thesis by blast
qed

lemma bij_betw_iteI:
  assumes "bij_betw f A B" "bij_betw g C D" "A ∩ C = {}" "B ∩ D = {}"
  shows "bij_betw (λx. if x ∈ A then f x else g x) (A ∪ C) (B ∪ D)"
proof -
  have "bij_betw (λx. if x ∈ A then f x else g x) A B"
    by (metis bij_betw_cong[of A f "λx. if x ∈ A then f x else g x" B] assms(1))
  moreover have "bij_betw (λx. if x ∈ A then f x else g x) C D"
    using bij_betw_cong[of C g "λx. if x ∈ A then f x else g x" D] assms(2,3) by force
  ultimately show ?thesis using bij_betw_combine[OF _ _ assms(4)] by metis
qed

lemma subst_comp_split:
  assumes "subst_domain ϑ ∩ range_vars ϑ = {}"
  shows "ϑ = (rm_vars (subst_domain ϑ - V) ϑ) ∘s (rm_vars V ϑ)" (is ?P)
    and "ϑ = (rm_vars V ϑ) ∘s (rm_vars (subst_domain ϑ - V) ϑ)" (is ?Q)
proof -
  let ?rm1 = "rm_vars (subst_domain ϑ - V) ϑ" and ?rm2 = "rm_vars V ϑ"
  have "subst_domain ?rm2 ∩ range_vars ?rm1 = {}"
       "subst_domain ?rm1 ∩ range_vars ?rm2 = {}"
    using assms unfolding range_vars_alt_def by (force simp add: subst_domain_def)+
  hence *: "⋀v. v ∈ subst_domain ?rm1 ⟹ (?rm1 ∘s ?rm2) v = ϑ v"
           "⋀v. v ∈ subst_domain ?rm2 ⟹ (?rm2 ∘s ?rm1) v = ϑ v"
    using ident_comp_subst_trm_if_disj[of ?rm2 ?rm1]
          ident_comp_subst_trm_if_disj[of ?rm1 ?rm2]
    by (auto simp add: subst_domain_def)
  hence "⋀v. v ∉ subst_domain ?rm1 ⟹ (?rm1 ∘s ?rm2) v = ϑ v"
        "⋀v. v ∉ subst_domain ?rm2 ⟹ (?rm2 ∘s ?rm1) v = ϑ v"
    unfolding subst_compose_def by (auto simp add: subst_domain_def)
  hence "⋀v. (?rm1 ∘s ?rm2) v = ϑ v" "⋀v. (?rm2 ∘s ?rm1) v = ϑ v" using * by blast+
  thus ?P ?Q by auto
qed

lemma subst_comp_eq_if_disjoint_vars:
  assumes "(subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"
  shows "γ ∘s δ = δ ∘s γ"
proof -
  { fix x assume "x ∈ subst_domain γ"
    hence "(γ ∘s δ) x = γ x" "(δ ∘s γ) x = γ x"
      using assms unfolding range_vars_alt_def by (force simp add: subst_compose)+
    hence "(γ ∘s δ) x = (δ ∘s γ) x" by metis
  } moreover
  { fix x assume "x ∈ subst_domain δ"
    hence "(γ ∘s δ) x = δ x" "(δ ∘s γ) x = δ x"
      using assms
```

```
      unfolding range_vars_alt_def by (auto simp add: subst_compose subst_domain_def)
    hence "(γ ∘s δ) x = (δ ∘s γ) x" by metis
  } moreover
  { fix x assume "x ∉ subst_domain γ" "x ∉ subst_domain δ"
    hence "(γ ∘s δ) x = (δ ∘s γ) x" by (simp add: subst_compose subst_domain_def)
  } ultimately show ?thesis by auto
qed

lemma subst_eq_if_disjoint_vars_ground:
  fixes ξ δ::"('f,'v) subst"
  assumes "subst_domain δ ∩ subst_domain ξ = {}" "ground (subst_range ξ)" "ground (subst_range δ)"
  shows "t · δ · ξ = t · ξ · δ"
by (metis assms subst_comp_eq_if_disjoint_vars range_vars_alt_def
          subst_subst_compose sup_bot.right_neutral)

lemma subst_img_bound: "subst_domain δ ∪ range_vars δ ⊆ fv t ⟹ range_vars δ ⊆ fv (t · δ)"
proof -
  assume "subst_domain δ ∪ range_vars δ ⊆ fv t"
  hence "subst_domain δ ⊆ fv t" by blast
  thus ?thesis
    by (metis (no_types) range_vars_alt_def le_iff_sup subst_apply_fv_unfold
              subst_apply_fv_union subst_range.simps)
qed

lemma subst_all_fv_subset: "fv t ⊆ fv_set M ⟹ fv (t · ϑ) ⊆ fv_set (M ·set ϑ)"
proof -
  assume *: "fv t ⊆ fv_set M"
  { fix v assume "v ∈ fv t"
    hence "v ∈ fv_set M" using * by auto
    then obtain t' where "t' ∈ M" "v ∈ fv t'" by auto
    hence "fv (ϑ v) ⊆ fv (t' · ϑ)"
      by (metis subst_apply_term.simps(1) subst_apply_fv_subset subst_apply_fv_unfold
                subtermeq_vars_subset vars_iff_subtermeq)
    hence "fv (ϑ v) ⊆ fv_set (M ·set ϑ)" using ‹t' ∈ M› by auto
  }
  thus ?thesis using subst_apply_fv_unfold[of t ϑ] by auto
qed

lemma subst_support_if_mgt_subst_idem:
  assumes "ϑ ≼∘ δ" "subst_idem ϑ"
  shows "ϑ supports δ"
proof -
  from ‹ϑ ≼∘ δ› obtain σ where σ: "δ = ϑ ∘s σ" by blast
  hence "⋀v. ϑ v · δ = Var v · (ϑ ∘s ϑ ∘s σ)" by simp
  hence "⋀v. ϑ v · δ = Var v · (ϑ ∘s σ)" using ‹subst_idem ϑ › unfolding subst_idem_def by simp
  hence "⋀v. ϑ v · δ = Var v · δ" using σ by simp
  thus "ϑ supports δ" by simp
qed

lemma subst_support_iff_mgt_if_subst_idem:
  assumes "subst_idem ϑ"
  shows "ϑ ≼∘ δ ⟷ ϑ supports δ"
proof
  show "ϑ ≼∘ δ ⟹ ϑ supports δ" by (fact subst_support_if_mgt_subst_idem[OF _ ‹subst_idem ϑ›])
  show "ϑ supports δ ⟹ ϑ ≼∘ δ" by (fact subst_supportD)
qed

lemma subst_support_comp:
  fixes ϑ δ ℐ::"('a,'b) subst"
  assumes "ϑ supports ℐ" "δ supports ℐ"
  shows "(ϑ ∘s δ) supports ℐ"
by (metis (no_types) assms subst_agreement subst_apply_term.simps(1) subst_subst_compose)
```

```
lemma subst_support_comp':
  fixes ϑ δ σ::"('a,'b) subst"
  assumes "ϑ supports δ"
  shows "ϑ supports (δ ∘s σ)" "σ supports δ ⟹ ϑ supports (σ ∘s δ)"
using assms unfolding subst_support_def by (metis subst_compose_assoc, metis)

lemma subst_support_comp_split:
  fixes ϑ δ ℐ::"('a,'b) subst"
  assumes "(ϑ ∘s δ) supports ℐ"
  shows "subst_domain ϑ ∩ range_vars ϑ = {} ⟹ ϑ supports ℐ"
  and "subst_domain ϑ ∩ subst_domain δ = {} ⟹ δ supports ℐ"
proof -
  assume "subst_domain ϑ ∩ range_vars ϑ = {}"
  hence "subst_idem ϑ" by (metis subst_idemI)
  have "ϑ ≼∘ ℐ" using assms subst_compose_assoc[of ϑ δ ℐ] unfolding subst_compose_def by metis
  show "ϑ supports ℐ" using subst_support_if_mgt_subst_idem[OF ‹ϑ ≼∘ ℐ› ‹subst_idem ϑ›] by auto
next
  assume "subst_domain ϑ ∩ subst_domain δ = {}"
  moreover have "∀v ∈ subst_domain (ϑ ∘s δ). (ϑ ∘s δ) v · ℐ = ℐ v" using assms by metis
  ultimately have "∀v ∈ subst_domain δ. δ v · ℐ = ℐ v"
    using var_not_in_subst_dom unfolding subst_compose_def
    by (metis IntI empty_iff subst_apply_term.simps(1))
  thus "δ supports ℐ" by force
qed

lemma subst_idem_support: "subst_idem ϑ ⟹ ϑ supports ϑ ∘s δ"
unfolding subst_idem_def by (metis subst_support_def subst_compose_assoc)

lemma subst_idem_iff_self_support: "subst_idem ϑ ⟷ ϑ supports ϑ"
using subst_support_def[of ϑ ϑ] unfolding subst_idem_def by auto

lemma subterm_subst_neq: "t ⊏ t' ⟹ t · s ≠ t' · s"
by (metis subst_mono_neq)

lemma fv_Fun_subst_neq: "x ∈ fv (Fun f T) ⟹ σ x ≠ Fun f T · σ"
using subterm_subst_neq[of "Var x" "Fun f T"] vars_iff_subterm_or_eq[of x "Fun f T"] by auto

lemma subterm_subst_unfold:
  assumes "t ⊑ s · ϑ"
  shows "(∃s'. s' ⊑ s ∧ t = s' · ϑ) ∨ (∃x ∈ fv s. t ⊏ ϑ x)"
using assms
proof (induction s)
  case (Fun f T) thus ?case
  proof (cases "t = Fun f T · ϑ")
    case True thus ?thesis using Fun by auto
  next
    case False
    then obtain s' where s': "s' ∈ set T" "t ⊑ s' · ϑ" using Fun by auto
    hence "(∃s''. s'' ⊑ s' ∧ t = s'' · ϑ) ∨ (∃x ∈ fv s'. t ⊏ ϑ x)" by (metis Fun.IH)
    thus ?thesis using s'(1) by auto
  qed
qed simp

lemma subterm_subst_img_subterm:
  assumes "t ⊑ s · ϑ" "⋀s'. s' ⊑ s ⟹ t ≠ s' · ϑ"
  shows "∃w ∈ fv s. t ⊏ ϑ w"
using subterm_subst_unfold[OF assms(1)] assms(2) by force

lemma subterm_subst_not_img_subterm:
  assumes "t ⊑ s · ℐ" "¬(∃w ∈ fv s. t ⊑ ℐ w)"
  shows "∃f T. Fun f T ⊑ s ∧ t = Fun f T · ℐ"
proof (rule ccontr)
  assume "¬(∃f T. Fun f T ⊑ s ∧ t = Fun f T · ℐ)"
```

```
  hence "⋀f T. Fun f T ⊑ s ⟹ t ≠ Fun f T · ℐ" by simp
  moreover have "⋀x. Var x ⊑ s ⟹ t ≠ Var x · ℐ"
    using assms(2) vars_iff_subtermeq by force
  ultimately have "⋀s'. s' ⊑ s ⟹ t ≠ s' · ℐ" by (metis "term.exhaust")
  thus False using assms subterm_subst_img_subterm by blast
qed

lemma subst_apply_img_var:
  assumes "v ∈ fv (t · δ)" "v ∉ fv t"
  obtains w where "w ∈ fv t" "v ∈ fv (δ w)"
using assms by (induct t) auto

lemma subst_apply_img_var':
  assumes "x ∈ fv (t · δ)" "x ∉ fv t"
  shows "∃y ∈ fv t. x ∈ fv (δ y)"
by (metis assms subst_apply_img_var)

lemma nth_map_subst:
  fixes ϑ::"('f,'v) subst" and T::"('f,'v) term list" and i::nat
  shows "i < length T ⟹ (map (λt. t · ϑ) T) ! i = (T ! i) · ϑ"
by (fact nth_map)

lemma subst_subterm:
  assumes "Fun f T ⊑ t · ϑ"
  shows "(∃S. Fun f S ⊑ t ∧ Fun f S · ϑ = Fun f T) ∨
         (∃s ∈ subst_range ϑ. Fun f T ⊑ s)"
using assms subterm_subst_not_img_subterm by (cases "∃s ∈ subst_range ϑ. Fun f T ⊑ s") fastforce+

lemma subst_subterm':
  assumes "Fun f T ⊑ t · ϑ"
  shows "∃S. length S = length T ∧ (Fun f S ⊑ t ∨ (∃s ∈ subst_range ϑ. Fun f S ⊑ s))"
using subst_subterm[OF assms] by auto

lemma subst_subterm'':
  assumes "s ∈ subterms (t · ϑ)"
  shows "(∃u ∈ subterms t. s = u · ϑ) ∨ s ∈ subterms_set (subst_range ϑ)"
proof (cases s)
  case (Var x)
  thus ?thesis
    using assms subterm_subst_not_img_subterm vars_iff_subtermeq
    by (cases "s = t · ϑ") fastforce+
next
  case (Fun f T)
  thus ?thesis
    using subst_subterm[of f T t ϑ] assms
    by fastforce
qed
```

2.3.3 More Small Lemmata

```
lemma funs_term_subst: "funs_term (t · ϑ) = funs_term t ∪ (⋃x ∈ fv t. funs_term (ϑ x))"
by (induct t) auto

lemma fv_set_subst_img_eq:
  assumes "X ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv_set (δ ` (Y - X)) = fv_set (δ ` Y) - X"
using assms unfolding range_vars_alt_def by force

lemma subst_Fun_index_eq:
  assumes "i < length T" "Fun f T · δ = Fun g T' · δ"
  shows "T ! i · δ = T' ! i · δ"
proof -
  have "map (λx. x · δ) T = map (λx. x · δ) T'" using assms by simp
```

```
  thus ?thesis by (metis assms(1) length_map nth_map)
qed

lemma fv_exists_if_unifiable_and_neq:
  fixes t t'::"('a,'b) term" and δ ϑ::"('a,'b) subst"
  assumes "t ≠ t'" "t · ϑ = t' · ϑ"
  shows "fv t ∪ fv t' ≠ {}"
proof
  assume "fv t ∪ fv t' = {}"
  hence "fv t = {}" "fv t' = {}" by auto
  hence "t · ϑ = t" "t' · ϑ = t'" by auto
  hence "t = t'" using assms(2) by metis
  thus False using assms(1) by auto
qed

lemma const_subterm_subst: "Fun c [] ⊑ t ⟹ Fun c [] ⊑ t · σ"
by (induct t) auto

lemma const_subterm_subst_var_obtain:
  assumes "Fun c [] ⊑ t · σ" "¬Fun c [] ⊑ t"
  obtains x where "x ∈ fv t" "Fun c [] ⊑ σ x"
using assms by (induct t) auto

lemma const_subterm_subst_cases:
  assumes "Fun c [] ⊑ t · σ"
  shows "Fun c [] ⊑ t ∨ (∃x ∈ fv t. x ∈ subst_domain σ ∧ Fun c [] ⊑ σ x)"
proof (cases "Fun c [] ⊑ t")
  case False
  then obtain x where "x ∈ fv t" "Fun c [] ⊑ σ x"
    using const_subterm_subst_var_obtain[OF assms] by moura
  thus ?thesis by (cases "x ∈ subst_domain σ") auto
qed simp

lemma fv_pairs_subst_fv_subset:
  assumes "x ∈ fv_pairs F"
  shows "fv (ϑ x) ⊆ fv_pairs (F ·_pairs ϑ)"
  using assms
proof (induction F)
  case (Cons f F)
  then obtain t t' where f: "f = (t,t')" by (metis surj_pair)
  show ?case
  proof (cases "x ∈ fv_pairs F")
    case True thus ?thesis
      using Cons.IH
      unfolding subst_apply_pairs_def
      by auto
  next
    case False
    hence "x ∈ fv t ∪ fv t'" using Cons.prems f by simp
    hence "fv (ϑ x) ⊆ fv (t · ϑ) ∪ fv (t' · ϑ)" using fv_subst_subset[of x] by force
    thus ?thesis using f unfolding subst_apply_pairs_def by auto
  qed
qed simp

lemma fv_pairs_step_subst: "fv_set (δ ` fv_pairs F) = fv_pairs (F ·_pairs δ)"
proof (induction F)
  case (Cons f F)
  obtain t t' where "f = (t,t')" by moura
  thus ?case
    using Cons
    by (simp add: subst_apply_pairs_def subst_apply_fv_unfold)
qed (simp_all add: subst_apply_pairs_def)
```

```
lemma fv_pairs_subst_obtain_var:
  fixes δ::"('a,'b) subst"
  assumes "x ∈ fv_pairs (F ·_pairs δ)"
  shows "∃y ∈ fv_pairs F. x ∈ fv (δ y)"
  using assms
proof (induction F)
  case (Cons f F)
  then obtain t s where f: "f = (t,s)" by (metis surj_pair)

  from Cons.IH show ?case
  proof (cases "x ∈ fv_pairs (F ·_pairs δ)")
    case False
    hence "x ∈ fv (t · δ) ∨ x ∈ fv (s · δ)"
      using f Cons.prems
      by (simp add: subst_apply_pairs_def)
    hence "(∃y ∈ fv t. x ∈ fv (δ y)) ∨ (∃y ∈ fv s. x ∈ fv (δ y))" by (metis fv_subst_obtain_var)
    thus ?thesis using f by (auto simp add: subst_apply_pairs_def)
  qed (auto simp add: Cons.IH)
qed (simp add: subst_apply_pairs_def)

lemma pair_subst_ident[intro]: "(fv t ∪ fv t') ∩ subst_domain ϑ = {} ⟹ (t,t') ·_p ϑ = (t,t')"
by auto

lemma pairs_substI[intro]:
  assumes "subst_domain ϑ ∩ (⋃(s,t) ∈ M. fv s ∪ fv t) = {}"
  shows "M ·_pset ϑ = M"
proof -
  { fix m assume M: "m ∈ M"
    then obtain s t where m: "m = (s,t)" by (metis surj_pair)
    hence "(fv s ∪ fv t) ∩ subst_domain ϑ = {}" using assms M by auto
    hence "m ·_p ϑ = m" using m by auto
  } thus ?thesis by (simp add: image_cong)
qed

lemma fv_pairs_subst: "fv_pairs (F ·_pairs ϑ) = fv_set (ϑ ` (fv_pairs F))"
proof (induction F)
  case (Cons g G)
  obtain t t' where "g = (t,t')" by (metis surj_pair)
  thus ?case
    using Cons.IH
    by (simp add: subst_apply_pairs_def subst_apply_fv_unfold)
qed (simp add: subst_apply_pairs_def)

lemma fv_pairs_subst_subset:
  assumes "fv_pairs (F ·_pairs δ) ⊆ subst_domain σ"
  shows  "fv_pairs F ⊆ subst_domain σ ∪ subst_domain δ"
  using assms
proof (induction F)
  case (Cons g G)
  hence IH: "fv_pairs G ⊆ subst_domain σ ∪ subst_domain δ"
    by (simp add: subst_apply_pairs_def)
  obtain t t' where g: "g = (t,t')" by (metis surj_pair)
  hence "fv (t · δ) ⊆ subst_domain σ" "fv (t' · δ) ⊆ subst_domain σ"
    using Cons.prems by (simp_all add: subst_apply_pairs_def)
  hence "fv t ⊆ subst_domain σ ∪ subst_domain δ" "fv t' ⊆ subst_domain σ ∪ subst_domain δ"
    using subst_apply_fv_unfold[of _ δ] by force+
  thus ?case using IH g by (simp add: subst_apply_pairs_def)
qed (simp add: subst_apply_pairs_def)

lemma pairs_subst_comp: "F ·_pairs δ ∘_s ϑ = ((F ·_pairs δ) ·_pairs ϑ)"
by (induct F) (auto simp add: subst_apply_pairs_def)

lemma pairs_substI'[intro]:
```

```
  "subst_domain ϑ ∩ fv_pairs F = {} ⟹ F ·pairs ϑ = F"
by (induct F) (force simp add: subst_apply_pairs_def)+

lemma subst_pair_compose[simp]: "d ·p (δ ∘s I) = d ·p δ ·p I"
proof -
  obtain t s where "d = (t,s)" by moura
  thus ?thesis by auto
qed

lemma subst_pairs_compose[simp]: "D ·pset (δ ∘s I) = D ·pset δ ·pset I"
by auto

lemma subst_apply_pair_pair: "(t, s) ·p I = (t · I, s · I)"
by (rule prod.case)

lemma subst_apply_pairs_nil[simp]: "[] ·pairs δ = []"
unfolding subst_apply_pairs_def by simp

lemma subst_apply_pairs_singleton[simp]: "[(t,s)] ·pairs δ = [(t · δ,s · δ)]"
unfolding subst_apply_pairs_def by simp

lemma subst_apply_pairs_Var[iff]: "F ·pairs Var = F" by (simp add: subst_apply_pairs_def)

lemma subst_apply_pairs_pset_subst: "set (F ·pairs ϑ) = set F ·pset ϑ"
unfolding subst_apply_pairs_def by force
```

2.3.4 Finite Substitutions

```
inductive_set fsubst::"('a,'b) subst set" where
  fvar:     "Var ∈ fsubst"
| FUpdate:  "⟦ϑ ∈ fsubst; v ∉ subst_domain ϑ; t ≠ Var v⟧ ⟹ ϑ(v := t) ∈ fsubst"

lemma finite_dom_iff_fsubst:
  "finite (subst_domain ϑ) ⟷ ϑ ∈ fsubst"
proof
  assume "finite (subst_domain ϑ)" thus "ϑ ∈ fsubst"
  proof (induction "subst_domain ϑ" arbitrary: ϑ rule: finite.induct)
    case emptyI
    hence "ϑ = Var" using empty_dom_iff_empty_subst by metis
    thus ?case using fvar by simp
  next
    case (insertI ϑ'dom v) thus ?case
    proof (cases "v ∈ ϑ'dom")
      case True
      hence "ϑ'dom = subst_domain ϑ" using ‹insert v ϑ'dom = subst_domain ϑ› by auto
      thus ?thesis using insertI.hyps(2) by metis
    next
      case False
      let ?ϑ' = "λw. if w ∈ ϑ'dom then ϑ w else Var w"
      have "subst_domain ?ϑ' = ϑ'dom"
        using ‹v ∉ ϑ'dom› ‹insert v ϑ'dom = subst_domain ϑ›
        by (auto simp add: subst_domain_def)
      hence "?ϑ' ∈ fsubst" using insertI.hyps(2) by simp
      moreover have "?ϑ'(v := ϑ v) = (λw. if w ∈ insert v ϑ'dom then ϑ w else Var w)" by auto
      hence "?ϑ'(v := ϑ v) = ϑ"
        using ‹insert v ϑ'dom = subst_domain ϑ›
        by (auto simp add: subst_domain_def)
      ultimately show ?thesis
        using FUpdate[of ?ϑ' v "ϑ v"] False insertI.hyps(3)
        by (auto simp add: subst_domain_def)
    qed
  qed
next
```

```
  assume "ϑ ∈ fsubst" thus "finite (subst_domain ϑ)"
  by (induct ϑ, simp, metis subst_dom_insert_finite)
qed

lemma fsubst_induct[case_names fvar FUpdate, induct set: finite]:
  assumes "finite (subst_domain δ)" "P Var"
  and "⋀ϑ v t. ⟦finite (subst_domain ϑ); v ∉ subst_domain ϑ; t ≠ Var v; P ϑ⟧ ⟹ P (ϑ(v := t))"
  shows "P δ"
using assms finite_dom_iff_fsubst fsubst.induct by metis

lemma fun_upd_fsubst: "s(v := t) ∈ fsubst ⟷ s ∈ fsubst"
using subst_dom_insert_finite[of s] finite_dom_iff_fsubst by blast

lemma finite_img_if_fsubst: "s ∈ fsubst ⟹ finite (subst_range s)"
using finite_dom_iff_fsubst finite_subst_img_if_finite_dom' by blast
```

2.3.5 Unifiers and Most General Unifiers (MGUs)

```
abbreviation Unifier::"('f,'v) subst ⇒ ('f,'v) term ⇒ ('f,'v) term ⇒ bool" where
  "Unifier σ t u ≡ (t · σ = u · σ)"

abbreviation MGU::"('f,'v) subst ⇒ ('f,'v) term ⇒ ('f,'v) term ⇒ bool" where
  "MGU σ t u ≡ Unifier σ t u ∧ (∀ϑ. Unifier ϑ t u ⟶ σ ≼∘ ϑ)"

lemma MGUI[intro]:
  shows "⟦t · σ = u · σ; ⋀ϑ::('f,'v) subst. t · ϑ = u · ϑ ⟹ σ ≼∘ ϑ⟧ ⟹ MGU σ t u"
by auto

lemma UnifierD[dest]:
  fixes σ::"('f,'v) subst" and f g::'f and X Y::"('f,'v) term list"
  assumes "Unifier σ (Fun f X) (Fun g Y)"
  shows "f = g" "length X = length Y"
proof -
  from assms show "f = g" by auto

  from assms have "Fun f X · σ = Fun g Y · σ" by auto
  hence "length (map (λx. x · σ) X) = length (map (λx. x · σ) Y)" by auto
  thus "length X = length Y" by auto
qed

lemma MGUD[dest]:
  fixes σ::"('f,'v) subst" and f g::'f and X Y::"('f,'v) term list"
  assumes "MGU σ (Fun f X) (Fun g Y)"
  shows "f = g" "length X = length Y"
using assms by (auto intro!: UnifierD[of f X σ g Y])

lemma MGU_sym[sym]: "MGU σ s t ⟹ MGU σ t s" by auto
lemma Unifier_sym[sym]: "Unifier σ s t ⟹ Unifier σ t s" by auto

lemma MGU_nil: "MGU Var s t ⟷ s = t" by fastforce

lemma Unifier_comp: "Unifier (ϑ ∘s δ) t u ⟹ Unifier δ (t · ϑ) (u · ϑ)"
by simp

lemma Unifier_comp': "Unifier δ (t · ϑ) (u · ϑ) ⟹ Unifier (ϑ ∘s δ) t u"
by simp

lemma Unifier_excludes_subterm:
  assumes ϑ: "Unifier ϑ t u"
  shows "¬t ⊏ u"
proof
  assume "t ⊏ u"
  hence "t · ϑ ⊏ u · ϑ" using subst_mono_neq by metis
```

```
  hence "t · ϑ ≠ u · ϑ" by simp
  moreover from ϑ have "t · ϑ = u · ϑ" by auto
  ultimately show False ..
qed

lemma MGU_is_Unifier: "MGU σ t u ⟹ Unifier σ t u" by (rule conjunct1)

lemma MGU_Var1:
  assumes "¬Var v ⊏ t"
  shows "MGU (Var(v := t)) (Var v) t"
proof (intro MGUI exI)
  show "Var v · (Var(v := t)) = t · (Var(v := t))" using assms subst_no_occs by fastforce
next
  fix ϑ::"('a,'b) subst" assume th: "Var v · ϑ = t · ϑ"
  show "ϑ = (Var(v := t)) ∘s ϑ"
  proof
    fix s show "s · ϑ = s · ((Var(v := t)) ∘s ϑ)" using th by (induct s) auto
  qed
qed

lemma MGU_Var2: "v ∉ fv t ⟹ MGU (Var(v := t)) (Var v) t"
by (metis (no_types) MGU_Var1 vars_iff_subterm_or_eq)

lemma MGU_Var3: "MGU Var (Var v) (Var w) ⟷ v = w" by fastforce

lemma MGU_Const1: "MGU Var (Fun c []) (Fun d []) ⟷ c = d" by fastforce

lemma MGU_Const2: "MGU ϑ (Fun c []) (Fun d []) ⟹ c = d" by auto

lemma MGU_Fun:
  assumes "MGU ϑ (Fun f X) (Fun g Y)"
  shows "f = g" "length X = length Y"
proof -
  let ?F = "λϑ X. map (λx. x · ϑ) X"
  from assms have
    "⟦f = g; ?F ϑ X = ?F ϑ Y; ∀ϑ'. f = g ∧ ?F ϑ' X = ?F ϑ' Y ⟶ ϑ ≼∘ ϑ'⟧ ⟹ length X = length Y"
    using map_eq_imp_length_eq by auto
  thus "f = g" "length X = length Y" using assms by auto
qed

lemma Unifier_Fun:
  assumes "Unifier ϑ (Fun f (x#X)) (Fun g (y#Y))"
  shows "Unifier ϑ x y" "Unifier ϑ (Fun f X) (Fun g Y)"
using assms by simp_all

lemma Unifier_subst_idem_subst:
  "subst_idem r ⟹ Unifier s (t · r) (u · r) ⟹ Unifier (r ∘s s) (t · r) (u · r)"
by (metis (no_types, lifting) subst_idem_def subst_subst_compose)

lemma subst_idem_comp:
  "subst_idem r ⟹ Unifier s (t · r) (u · r) ⟹
    (⋀q. Unifier q (t · r) (u · r) ⟹ s ∘s q = q) ⟹
    subst_idem (r ∘s s)"
by (frule Unifier_subst_idem_subst, blast, metis subst_idem_def subst_compose_assoc)

lemma Unifier_mgt: "⟦Unifier δ t u; δ ≼∘ ϑ⟧ ⟹ Unifier ϑ t u" by auto

lemma Unifier_support: "⟦Unifier δ t u; δ supports ϑ⟧ ⟹ Unifier ϑ t u"
using subst_supportD Unifier_mgt by metis

lemma MGU_mgt: "⟦MGU σ t u; MGU δ t u⟧ ⟹ σ ≼∘ δ" by auto

lemma Unifier_trm_fv_bound:
```

```
  "⟦Unifier s t u; v ∈ fv t⟧ ⟹ v ∈ subst_domain s ∪ range_vars s ∪ fv u"
proof (induction t arbitrary: s u)
  case (Fun f X)
  hence "v ∈ fv (u · s) ∨ v ∈ subst_domain s" by (metis subst_not_dom_fixed)
  thus ?case by (metis (no_types) Un_iff contra_subsetD subst_sends_fv_to_img)
qed (metis (no_types) UnI1 UnI2 subsetCE no_var_subterm subst_sends_dom_to_img
           subst_to_var_is_var trm_subst_ident' vars_iff_subterm_or_eq)

lemma Unifier_rm_var: "⟦Unifier ϑ s t; v ∉ fv s ∪ fv t⟧ ⟹ Unifier (rm_var v ϑ) s t"
by (auto simp add: repl_invariance)

lemma Unifier_ground_rm_vars:
  assumes "ground (subst_range s)" "Unifier (rm_vars X s) t t'"
  shows "Unifier s t t'"
by (rule Unifier_support[OF assms(2) rm_vars_ground_supports[OF assms(1)]])

lemma Unifier_dom_restrict:
  assumes "Unifier s t t'" "fv t ∪ fv t' ⊆ S"
  shows "Unifier (rm_vars (UNIV - S) s) t t'"
proof -
  let ?s = "rm_vars (UNIV - S) s"
  show ?thesis using term_subst_eq_conv[of t s ?s] term_subst_eq_conv[of t' s ?s] assms by auto
qed
```

2.3.6 Well-formedness of Substitutions and Unifiers

```
inductive_set wf_subst_set::"('a,'b) subst set" where
  Empty[simp]: "Var ∈ wf_subst_set"
| Insert[simp]:
    "⟦ϑ ∈ wf_subst_set; v ∉ subst_domain ϑ;
      v ∉ range_vars ϑ; fv t ∩ (insert v (subst_domain ϑ)) = {}⟧
      ⟹ ϑ(v := t) ∈ wf_subst_set"

definition wf_subst::"('a,'b) subst ⇒ bool" where
  "wf_subst ϑ ≡ subst_domain ϑ ∩ range_vars ϑ = {} ∧ finite (subst_domain ϑ)"

definition wf_MGU::"('a,'b) subst ⇒ ('a,'b) term ⇒ ('a,'b) term ⇒ bool" where
  "wf_MGU ϑ s t ≡ wf_subst ϑ ∧ MGU ϑ s t ∧ subst_domain ϑ ∪ range_vars ϑ ⊆ fv s ∪ fv t"

lemma wf_subst_subst_idem: "wf_subst ϑ ⟹ subst_idem ϑ" using subst_idemI[of ϑ] unfolding
wf_subst_def by fast

lemma wf_subst_properties: "ϑ ∈ wf_subst_set = wf_subst ϑ"
proof
  show "wf_subst ϑ ⟹ ϑ ∈ wf_subst_set" unfolding wf_subst_def
  proof -
    assume "subst_domain ϑ ∩ range_vars ϑ = {} ∧ finite (subst_domain ϑ)"
    hence "finite (subst_domain ϑ)" "subst_domain ϑ ∩ range_vars ϑ = {}"
      by auto
    thus "ϑ ∈ wf_subst_set"
    proof (induction ϑ rule: fsubst_induct)
      case fvar thus ?case by simp
    next
      case (FUpdate δ v t)
      have "subst_domain δ ⊆ subst_domain (δ(v := t))" "range_vars δ ⊆ range_vars (δ(v := t))"
        using FUpdate.hyps(2,3) subst_img_update
        unfolding range_vars_alt_def by (fastforce simp add: subst_domain_def)+
      hence "subst_domain δ ∩ range_vars δ = {}" using FUpdate.prems(1) by blast
      hence "δ ∈ wf_subst_set" using FUpdate.IH by metis

      have *: "range_vars (δ(v := t)) = range_vars δ ∪ fv t"
        using FUpdate.hyps(2) subst_img_update[OF _ FUpdate.hyps(3)]
        by fastforce
```

```
      hence "fv t ∩ insert v (subst_domain δ) = {}"
        using FUpdate.prems subst_dom_update2[OF FUpdate.hyps(3)] by blast
      moreover have "subst_domain (δ(v := t)) = insert v (subst_domain δ)"
        by (meson FUpdate.hyps(3) subst_dom_update2)
      hence "v ∉ range_vars δ" using FUpdate.prems * by blast
      ultimately show ?case using Insert[OF ‹δ ∈ wf_subst_set› ‹v ∉ subst_domain δ›] by metis
    qed
  qed

  show "ϑ ∈ wf_subst_set ⟹ wf_subst ϑ" unfolding wf_subst_def
  proof (induction ϑ rule: wf_subst_set.induct)
    case Empty thus ?case by simp
  next
    case (Insert σ v t)
    hence 1: "subst_domain σ ∩ range_vars σ = {}" by simp
    hence 2: "subst_domain (σ(v := t)) ∩ range_vars σ = {}"
      using Insert.hyps(3) by (auto simp add: subst_domain_def)
    have 3: "fv t ∩ subst_domain (σ(v := t)) = {}"
      using Insert.hyps(4) by (auto simp add: subst_domain_def)
    have 4: "σ v = Var v" using ‹v ∉ subst_domain σ› by (simp add: subst_domain_def)

    from Insert.IH have "finite (subst_domain σ)" by simp
    hence 5: "finite (subst_domain (σ(v := t)))" using subst_dom_insert_finite[of σ] by simp

    have "subst_domain (σ(v := t)) ∩ range_vars (σ(v := t)) = {}"
    proof (cases "t = Var v")
      case True
      hence "range_vars (σ(v := t)) = range_vars σ"
        using 4 fun_upd_triv term.inject(1)
        unfolding range_vars_alt_def by (auto simp add: subst_domain_def)
      thus "subst_domain (σ(v := t)) ∩ range_vars (σ(v := t)) = {}"
        using 1 2 3 by auto
    next
      case False
      hence "range_vars (σ(v := t)) = fv t ∪ (range_vars σ)"
        using 4 subst_img_update[of σ v] by auto
      thus "subst_domain (σ(v := t)) ∩ range_vars (σ(v := t)) = {}" using 1 2 3 by blast
    qed
    thus ?case using 5 by blast
  qed
qed

lemma wf_subst_induct[consumes 1, case_names Empty Insert]:
  assumes "wf_subst δ" "P Var"
  and "⋀ϑ v t. ⟦wf_subst ϑ; P ϑ; v ∉ subst_domain ϑ; v ∉ range_vars ϑ;
              fv t ∩ insert v (subst_domain ϑ) = {}⟧
              ⟹ P (ϑ(v := t))"
  shows "P δ"
proof -
  from assms(1,3) wf_subst_properties have
    "δ ∈ wf_subst_set"
    "⋀ϑ v t. ⟦ϑ ∈ wf_subst_set; P ϑ; v ∉ subst_domain ϑ; v ∉ range_vars ϑ;
            fv t ∩ insert v (subst_domain ϑ) = {}⟧
            ⟹ P (ϑ(v := t))"
    by blast+
  thus "P δ" using wf_subst_set.induct assms(2) by blast
qed

lemma wf_subst_fsubst: "wf_subst δ ⟹ δ ∈ fsubst"
unfolding wf_subst_def using finite_dom_iff_fsubst by blast

lemma wf_subst_nil: "wf_subst Var" unfolding wf_subst_def by simp
```

```
lemma wf_MGU_nil: "MGU Var s t ⟹ wf_MGU Var s t"
using wf_subst_nil subst_domain_Var range_vars_Var
unfolding wf_MGU_def by fast

lemma wf_MGU_dom_bound: "wf_MGU ϑ s t ⟹ subst_domain ϑ ⊆ fv s ∪ fv t" unfolding wf_MGU_def by
blast

lemma wf_subst_single:
  assumes "v ∉ fv t" "σ v = t" "⋀w. v ≠ w ⟹ σ w = Var w"
  shows "wf_subst σ"
proof -
  have *: "subst_domain σ = {v}" by (metis subst_fv_dom_img_single(1)[OF assms])

  have "subst_domain σ ∩ range_vars σ = {}"
    using * assms subst_fv_dom_img_single(2)
    by (metis inf_bot_left insert_disjoint(1))
  moreover have "finite (subst_domain σ)" using * by simp
  ultimately show ?thesis by (metis wf_subst_def)
qed

lemma wf_subst_reduction:
  "wf_subst s ⟹ wf_subst (rm_var v s)"
proof -
  assume "wf_subst s"
  moreover have "subst_domain (rm_var v s) ⊆ subst_domain s" by (auto simp add: subst_domain_def)
  moreover have "range_vars (rm_var v s) ⊆ range_vars s"
    unfolding range_vars_alt_def by (auto simp add: subst_domain_def)
  ultimately have "subst_domain (rm_var v s) ∩ range_vars (rm_var v s) = {}"
    by (meson compl_le_compl_iff disjoint_eq_subset_Compl subset_trans wf_subst_def)
  moreover have "finite (subst_domain (rm_var v s))"
    using ‹subst_domain (rm_var v s) ⊆ subst_domain s› ‹wf_subst s› rev_finite_subset
    unfolding wf_subst_def by blast
  ultimately show "wf_subst (rm_var v s)" by (metis wf_subst_def)
qed

lemma wf_subst_compose:
  assumes "wf_subst ϑ1" "wf_subst ϑ2"
    and "subst_domain ϑ1 ∩ subst_domain ϑ2 = {}"
    and "subst_domain ϑ1 ∩ range_vars ϑ2 = {}"
  shows "wf_subst (ϑ1 ∘s ϑ2)"
using assms
proof (induction ϑ1 rule: wf_subst_induct)
  case Empty thus ?case unfolding wf_subst_def by simp
next
  case (Insert σ1 v t)
  have "t ≠ Var v" using Insert.hyps(4) by auto
  hence dom1v_unfold: "subst_domain (σ1(v := t)) = insert v (subst_domain σ1)"
    using subst_dom_update2 by metis
  hence doms_disj: "subst_domain σ1 ∩ subst_domain ϑ2 = {}"
    using Insert.prems(2) disjoint_insert(1) by blast
  moreover have dom_img_disj: "subst_domain σ1 ∩ range_vars ϑ2 = {}"
    using Insert.hyps(2) Insert.prems(3)
    by (fastforce simp add: subst_domain_def)
  ultimately have "wf_subst (σ1 ∘s ϑ2)" using Insert.IH[OF ‹wf_subst ϑ2›] by metis

  have dom_comp_is_union: "subst_domain (σ1 ∘s ϑ2) = subst_domain σ1 ∪ subst_domain ϑ2"
    using subst_dom_comp_eq[OF dom_img_disj] .

  have "v ∉ subst_domain ϑ2"
    using Insert.prems(2) ‹t ≠ Var v›
    by (fastforce simp add: subst_domain_def)
  hence "ϑ2 v = Var v" "σ1 v = Var v" using Insert.hyps(2) by (simp_all add: subst_domain_def)
  hence "(σ1 ∘s ϑ2) v = Var v" "(σ1(v := t) ∘s ϑ2) v = t · ϑ2" "((σ1 ∘s ϑ2)(v := t)) v = t"
```

```
    unfolding subst_compose_def by simp_all

  have fv_t2_bound: "fv (t · ϑ2) ⊆ fv t ∪ range_vars ϑ2" by (meson subst_sends_fv_to_img)

  have 1: "v ∉ subst_domain (σ1 ∘s ϑ2)"
    using ‹(σ1 ∘s ϑ2) v = Var v›
    by (auto simp add: subst_domain_def)

  have "insert v (subst_domain σ1) ∩ range_vars ϑ2 = {}"
    using Insert.prems(3) dom1v_unfold by blast
  hence "v ∉ range_vars σ1 ∪ range_vars ϑ2" using Insert.hyps(3) by blast
  hence 2: "v ∉ range_vars (σ1 ∘s ϑ2)" by (meson set_rev_mp subst_img_comp_subset)

  have "subst_domain ϑ2 ∩ range_vars ϑ2 = {}"
    using ‹wf_subst ϑ2› unfolding wf_subst_def by simp
  hence "fv (t · ϑ2) ∩ subst_domain ϑ2 = {}"
    using subst_dom_elim unfolding range_vars_alt_def by simp
  moreover have "v ∉ range_vars ϑ2" using Insert.prems(3) dom1v_unfold by blast
  hence "v ∉ fv t ∪ range_vars ϑ2" using Insert.hyps(4) by blast
  hence "v ∉ fv (t · ϑ2)" using ‹fv (t · ϑ2) ⊆ fv t ∪ range_vars ϑ2› by blast
  moreover have "fv (t · ϑ2) ∩ subst_domain σ1 = {}"
    using dom_img_disj fv_t2_bound ‹fv t ∩ insert v (subst_domain σ1) = {}› by blast
  ultimately have 3: "fv (t · ϑ2) ∩ insert v (subst_domain (σ1 ∘s ϑ2)) = {}"
    using dom_comp_is_union by blast

  have "σ1(v := t) ∘s ϑ2 = (σ1 ∘s ϑ2)(v := t · ϑ2)" using subst_comp_upd1[of σ1 v t ϑ2] .
  moreover have "wf_subst ((σ1 ∘s ϑ2)(v := t · ϑ2))"
    using "wf_subst_set.Insert"[OF _ 1 2 3] ‹wf_subst (σ1 ∘s ϑ2)› wf_subst_properties by metis
  ultimately show ?case by presburger
qed

lemma wf_subst_append:
  fixes ϑ1 ϑ2::"('f,'v) subst"
  assumes "wf_subst ϑ1" "wf_subst ϑ2"
    and "subst_domain ϑ1 ∩ subst_domain ϑ2 = {}"
    and "subst_domain ϑ1 ∩ range_vars ϑ2 = {}"
    and "range_vars ϑ1 ∩ subst_domain ϑ2 = {}"
  shows "wf_subst (λv. if ϑ1 v = Var v then ϑ2 v else ϑ1 v)"
using assms
proof (induction ϑ1 rule: wf_subst_induct)
  case Empty thus ?case unfolding wf_subst_def by simp
next
  case (Insert σ1 v t)
  let ?if = "λw. if σ1 w = Var w then ϑ2 w else σ1 w"
  let ?if_upd = "λw. if (σ1(v := t)) w = Var w then ϑ2 w else (σ1(v := t)) w"

  from Insert.hyps(4) have "?if_upd = ?if(v := t)" by fastforce

  have dom_insert: "subst_domain (σ1(v := t)) = insert v (subst_domain σ1)"
    using Insert.hyps(4) by (auto simp add: subst_domain_def)

  have "σ1 v = Var v" "t ≠ Var v" using Insert.hyps(2,4) by auto
  hence img_insert: "range_vars (σ1(v := t)) = range_vars σ1 ∪ fv t"
    using subst_img_update by metis

  from Insert.prems(2) dom_insert have "subst_domain σ1 ∩ subst_domain ϑ2 = {}"
    by (auto simp add: subst_domain_def)
  moreover have "subst_domain σ1 ∩ range_vars ϑ2 = {}"
    using Insert.prems(3) dom_insert
    by (simp add: subst_domain_def)
  moreover have "range_vars σ1 ∩ subst_domain ϑ2 = {}"
    using Insert.prems(4) img_insert
    by blast
```

```
  ultimately have "wf_subst ?if" using Insert.IH[OF Insert.prems(1)] by metis

  have dom_union: "subst_domain ?if = subst_domain σ1 ∪ subst_domain ϑ2"
    by (auto simp add: subst_domain_def)
  hence "v ∉ subst_domain ?if"
    using Insert.hyps(2) Insert.prems(2) dom_insert
    by (auto simp add: subst_domain_def)
  moreover have "v ∉ range_vars ?if"
    using Insert.prems(3) Insert.hyps(3) dom_insert
    unfolding range_vars_alt_def by (auto simp add: subst_domain_def)
  moreover have "fv t ∩ insert v (subst_domain ?if) = {}"
    using Insert.hyps(4) Insert.prems(4) img_insert
    unfolding range_vars_alt_def by (fastforce simp add: subst_domain_def)
  ultimately show ?case
    using wf_subst_set.Insert ‹wf_subst ?if› ‹?if_upd = ?if(v := t)› wf_subst_properties
    by (metis (no_types, lifting))
qed

lemma wf_subst_elim_append:
  assumes "wf_subst ϑ" "subst_elim ϑ v" "v ∉ fv t"
  shows "subst_elim (ϑ(w := t)) v"
using assms
proof (induction ϑ rule: wf_subst_induct)
  case (Insert ϑ v' t')
  hence "⋀q. v ∉ fv (Var q · ϑ(v' := t'))" using subst_elimD by blast
  hence "⋀q. v ∉ fv (Var q · ϑ(v' := t', w := t))" using ‹v ∉ fv t› by simp
  thus ?case by (metis subst_elimI' subst_apply_term.simps(1))
qed (simp add: subst_elim_def)

lemma wf_subst_elim_dom:
  assumes "wf_subst ϑ"
  shows "∀ v ∈ subst_domain ϑ. subst_elim ϑ v"
using assms
proof (induction ϑ rule: wf_subst_induct)
  case (Insert ϑ w t)
  have dom_insert: "subst_domain (ϑ(w := t)) ⊆ insert w (subst_domain ϑ)"
    by (auto simp add: subst_domain_def)
  hence "∀ v ∈ subst_domain ϑ. subst_elim (ϑ(w := t)) v" using Insert.IH Insert.hyps(2,4)
    by (metis Insert.hyps(1) IntI disjoint_insert(2) empty_iff wf_subst_elim_append)
  moreover have "w ∉ fv t" using Insert.hyps(4) by simp
  hence "⋀q. w ∉ fv (Var q · ϑ(w := t))"
    by (metis fv_simps(1) fv_in_subst_img Insert.hyps(3) contra_subsetD
              fun_upd_def singletonD subst_apply_term.simps(1))
  hence "subst_elim (ϑ(w := t)) w" by (metis subst_elimI')
  ultimately show ?case using dom_insert by blast
qed simp

lemma wf_subst_support_iff_mgt: "wf_subst ϑ ⟹ ϑ supports δ ⟷ ϑ ≼∘ δ"
using subst_support_def subst_support_if_mgt_subst_idem wf_subst_subst_idem by blast
```

2.3.7 Interpretations

```
abbreviation interpretation_subst::"('a,'b) subst ⇒ bool" where
  "interpretation_subst ϑ ≡ subst_domain ϑ = UNIV ∧ ground (subst_range ϑ)"

lemma interpretation_substI:
  "(⋀v. fv (ϑ v) = {}) ⟹ interpretation_subst ϑ"
proof -
  assume "⋀v. fv (ϑ v) = {}"
  moreover { fix v assume "fv (ϑ v) = {}" hence "v ∈ subst_domain ϑ" by auto }
  ultimately show ?thesis by auto
qed
```

```
lemma interpretation_grounds[simp]:
  "interpretation_subst ϑ ⟹ fv (t · ϑ) = {}"
using subst_fv_dom_ground_if_ground_img[of t ϑ] by blast

lemma interpretation_grounds_all:
  "interpretation_subst ϑ ⟹ (⋀v. fv (ϑ v) = {})"
by (metis range_vars_alt_def UNIV_I fv_in_subst_img subset_empty subst_dom_vars_in_subst)

lemma interpretation_grounds_all':
  "interpretation_subst ϑ ⟹ ground (M ·set ϑ)"
using subst_fv_dom_ground_if_ground_img[of _ ϑ]
by simp

lemma interpretation_comp:
  assumes "interpretation_subst ϑ"
  shows "interpretation_subst (σ ∘s ϑ)" "interpretation_subst (ϑ ∘s σ)"
proof -
  have ϑ_fv: "fv (ϑ v) = {}" for v using interpretation_grounds_all[OF assms] by simp
  hence ϑ_fv': "fv (t · ϑ) = {}" for t
    by (metis all_not_in_conv subst_elimD subst_elimI' subst_apply_term.simps(1))

  from assms have "(σ ∘s ϑ) v ≠ Var v" for v
    unfolding subst_compose_def by (metis fv_simps(1) ϑ_fv' insert_not_empty)
  hence "subst_domain (σ ∘s ϑ) = UNIV" by (simp add: subst_domain_def)
  moreover have "fv ((σ ∘s ϑ) v) = {}" for v unfolding subst_compose_def using ϑ_fv' by simp
  hence "ground (subst_range (σ ∘s ϑ))" by simp
  ultimately show "interpretation_subst (σ ∘s ϑ)" ..

  from assms have "(ϑ ∘s σ) v ≠ Var v" for v
    unfolding subst_compose_def by (metis fv_simps(1) ϑ_fv insert_not_empty subst_to_var_is_var)
  hence "subst_domain (ϑ ∘s σ) = UNIV" by (simp add: subst_domain_def)
  moreover have "fv ((ϑ ∘s σ) v) = {}" for v
    unfolding subst_compose_def by (simp add: ϑ_fv trm_subst_ident)
  hence "ground (subst_range (ϑ ∘s σ))" by simp
  ultimately show "interpretation_subst (ϑ ∘s σ)" ..
qed

lemma interpretation_subst_exists:
  "∃ℐ::('f,'v) subst. interpretation_subst ℐ"
proof -
  obtain c::"'f" where "c ∈ UNIV" by simp
  then obtain ℐ::"('f,'v) subst" where "⋀v. ℐ v = Fun c []" by simp
  hence "subst_domain ℐ = UNIV" "ground (subst_range ℐ)"
    by (simp_all add: subst_domain_def)
  thus ?thesis by auto
qed

lemma interpretation_subst_exists':
  "∃ϑ::('f,'v) subst. subst_domain ϑ = X ∧ ground (subst_range ϑ)"
proof -
  obtain ℐ::"('f,'v) subst" where ℐ: "subst_domain ℐ = UNIV" "ground (subst_range ℐ)"
    using interpretation_subst_exists by moura
  let ?ϑ = "rm_vars (UNIV - X) ℐ"
  have 1: "subst_domain ?ϑ = X" using ℐ by (auto simp add: subst_domain_def)
  hence 2: "ground (subst_range ?ϑ)" using ℐ by force
  show ?thesis using 1 2 by blast
qed

lemma interpretation_subst_idem:
  "interpretation_subst ϑ ⟹ subst_idem ϑ"
unfolding subst_idem_def
using interpretation_grounds_all[of ϑ] trm_subst_ident subst_eq_if_eq_vars
by fastforce
```

```
lemma subst_idem_comp_upd_eq:
  assumes "v ∉ subst_domain ℐ" "subst_idem ϑ"
  shows "ℐ ∘s ϑ = ℐ(v := ϑ v) ∘s ϑ"
proof -
  from assms(1) have "(ℐ ∘s ϑ) v = ϑ v" unfolding subst_compose_def by auto
  moreover have "⋀w. w ≠ v ⟹ (ℐ ∘s ϑ) w = (ℐ(v := ϑ v) ∘s ϑ) w" unfolding subst_compose_def by
auto
  moreover have "(ℐ(v := ϑ v) ∘s ϑ) v = ϑ v" using assms(2) unfolding subst_idem_def
subst_compose_def
    by (metis fun_upd_same)
  ultimately show ?thesis by (metis fun_upd_same fun_upd_triv subst_comp_upd1)
qed

lemma interpretation_dom_img_disjoint:
  "interpretation_subst ℐ ⟹ subst_domain ℐ ∩ range_vars ℐ = {}"
unfolding range_vars_alt_def by auto
```

2.3.8 Basic Properties of MGUs

```
lemma MGU_is_mgu_singleton: "MGU ϑ t u = is_mgu ϑ {(t,u)}"
unfolding is_mgu_def unifiers_def by auto

lemma Unifier_in_unifiers_singleton: "Unifier ϑ s t ⟷ ϑ ∈ unifiers {(s,t)}"
unfolding unifiers_def by auto

lemma subst_list_singleton_fv_subset:
  "(⋃x ∈ set (subst_list (subst v t) E). fv (fst x) ∪ fv (snd x))
    ⊆ fv t ∪ (⋃x ∈ set E. fv (fst x) ∪ fv (snd x))"
proof (induction E)
  case (Cons x E)
  let ?fvs = "λL. ⋃x ∈ set L. fv (fst x) ∪ fv (snd x)"
  let ?fvx = "fv (fst x) ∪ fv (snd x)"
  let ?fvxsubst = "fv (fst x · Var(v := t)) ∪ fv (snd x · Var(v := t))"
  have "?fvs (subst_list (subst v t) (x#E)) = ?fvxsubst ∪ ?fvs (subst_list (subst v t) E)"
    unfolding subst_list_def subst_def by auto
  hence "?fvs (subst_list (subst v t) (x#E)) ⊆ ?fvxsubst ∪ fv t ∪ ?fvs E"
    using Cons.IH by blast
  moreover have "?fvs (x#E) = ?fvx ∪ ?fvs E" by auto
  moreover have "?fvxsubst ⊆ ?fvx ∪ fv t" using subst_fv_bound_singleton[of _ v t] by blast
  ultimately show ?case unfolding range_vars_alt_def by auto
qed (simp add: subst_list_def)

lemma subst_of_dom_subset: "subst_domain (subst_of L) ⊆ set (map fst L)"
proof (induction L rule: List.rev_induct)
  case (snoc x L)
  then obtain v t where x: "x = (v,t)" by (metis surj_pair)
  hence "subst_of (L@[x]) = Var(v := t) ∘s subst_of L"
    unfolding subst_of_def subst_def by (induct L) auto
  hence "subst_domain (subst_of (L@[x])) ⊆ insert v (subst_domain (subst_of L))"
    using x subst_domain_compose[of "Var(v := t)" "subst_of L"]
    by (auto simp add: subst_domain_def)
  thus ?case using snoc.IH x by auto
qed simp

lemma wf_MGU_is_imgu_singleton: "wf_MGU ϑ s t ⟹ is_imgu ϑ {(s,t)}"
proof -
  assume 1: "wf_MGU ϑ s t"

  have 2: "subst_idem ϑ" by (metis wf_subst_subst_idem 1 wf_MGU_def)

  have 3: "∀ϑ' ∈ unifiers {(s,t)}. ϑ ≼∘ ϑ'" "ϑ ∈ unifiers {(s,t)}"
    by (metis 1 Unifier_in_unifiers_singleton wf_MGU_def)+
```

```
  have "∀τ ∈ unifiers {(s,t)}. τ = ϑ ∘s τ" by (metis 2 3 subst_idem_def subst_compose_assoc)
  thus "is_imgu ϑ {(s,t)}" by (metis is_imgu_def ‹ϑ ∈ unifiers {(s,t)}›)
qed

lemma mgu_subst_range_vars:
  assumes "mgu s t = Some σ" shows "range_vars σ ⊆ vars_term s ∪ vars_term t"
proof -
  obtain xs where *: "Unification.unify [(s, t)] [] = Some xs" and [simp]: "subst_of xs = σ"
    using assms by (simp split: option.splits)
  from unify_Some_UNIF [OF *] obtain ss
    where "compose ss = σ" and "UNIF ss {#(s, t)#} {#}" by auto
  with UNIF_range_vars_subset [of ss "{#(s, t)#}" "{#}"]
    show ?thesis by (metis vars_mset_singleton fst_conv snd_conv)
qed

lemma mgu_subst_domain_range_vars_disjoint:
  assumes "mgu s t = Some σ" shows "subst_domain σ ∩ range_vars σ = {}"
proof -
  have "is_imgu σ {(s, t)}" using assms mgu_sound by simp
  hence "σ = σ ∘s σ" unfolding is_imgu_def by blast
  thus ?thesis by (metis subst_idemp_iff)
qed

lemma mgu_same_empty: "mgu (t::('a,'b) term) t = Some Var"
proof -
  { fix E::"('a,'b) equation list" and U::"('b × ('a,'b) term) list"
    assume "∀(s,t) ∈ set E. s = t"
    hence "Unification.unify E U = Some U"
    proof (induction E U rule: Unification.unify.induct)
      case (2 f S g T E U)
      hence *: "f = g" "S = T" by auto
      moreover have "∀(s,t) ∈ set (zip T T). s = t" by (induct T) auto
      hence "∀(s,t) ∈ set (zip T T@E). s = t" using "2.prems"(1) by auto
      moreover have "zip_option S T = Some (zip S T)" using ‹S = T› by auto
      hence **: "decompose (Fun f S) (Fun g T) = Some (zip S T)"
        using ‹f = g› unfolding decompose_def by auto
      ultimately have "Unification.unify (zip S T@E) U = Some U" using "2.IH" * by auto
      thus ?case using ** by auto
    qed auto
  }
  hence "Unification.unify [(t,t)] [] = Some []" by auto
  thus ?thesis by auto
qed

lemma mgu_var: assumes "x ∉ fv t" shows "mgu (Var x) t = Some (Var(x := t))"
proof -
  have "unify [(Var x,t)] [] = Some [(x,t)]" using assms by (auto simp add: subst_list_def)
  moreover have "subst_of [(x,t)] = Var(x := t)" unfolding subst_of_def subst_def by simp
  ultimately show ?thesis by simp
qed

lemma mgu_gives_wellformed_subst:
  assumes "mgu s t = Some ϑ" shows "wf_subst ϑ"
using mgu_finite_subst_domain[OF assms] mgu_subst_domain_range_vars_disjoint[OF assms]
unfolding wf_subst_def
by auto

lemma mgu_gives_wellformed_MGU:
  assumes "mgu s t = Some ϑ" shows "wf_MGU ϑ s t"
using mgu_subst_domain[OF assms] mgu_sound[OF assms] mgu_subst_range_vars [OF assms]
      MGU_is_mgu_singleton[of s ϑ t] is_imgu_imp_is_mgu[of ϑ "{(s,t)}"]
      mgu_gives_wellformed_subst[OF assms]
```

```
unfolding wf_MGU_def by blast

lemma mgu_vars_bounded[dest?]:
  "mgu M N = Some σ ⟹ subst_domain σ ∪ range_vars σ ⊆ fv M ∪ fv N"
using mgu_gives_wellformed_MGU unfolding wf_MGU_def by blast

lemma mgu_gives_subst_idem: "mgu s t = Some ϑ ⟹ subst_idem ϑ"
using mgu_sound[of s t ϑ] unfolding is_imgu_def subst_idem_def by auto

lemma mgu_always_unifies: "Unifier ϑ M N ⟹ ∃δ. mgu M N = Some δ"
using mgu_complete Unifier_in_unifiers_singleton by blast

lemma mgu_gives_MGU: "mgu s t = Some ϑ ⟹ MGU ϑ s t"
using mgu_sound[of s t ϑ, THEN is_imgu_imp_is_mgu] MGU_is_mgu_singleton by metis

lemma mgu_eliminates[dest?]:
  assumes "mgu M N = Some σ"
  shows "(∃v ∈ fv M ∪ fv N. subst_elim σ v) ∨ σ = Var"
  (is "?P M N σ")
proof (cases "σ = Var")
  case False
  then obtain v where v: "v ∈ subst_domain σ" by auto
  hence "v ∈ fv M ∪ fv N" using mgu_vars_bounded[OF assms] by blast
  thus ?thesis using wf_subst_elim_dom[OF mgu_gives_wellformed_subst[OF assms]] v by blast
qed simp

lemma mgu_eliminates_dom:
  assumes "mgu x y = Some ϑ" "v ∈ subst_domain ϑ"
  shows "subst_elim ϑ v"
using mgu_gives_wellformed_subst[OF assms(1)]
unfolding wf_MGU_def wf_subst_def subst_elim_def
by (metis disjoint_iff_not_equal subst_dom_elim assms(2))

lemma unify_list_distinct:
  assumes "Unification.unify E B = Some U" "distinct (map fst B)"
  and "(⋃x ∈ set E. fv (fst x) ∪ fv (snd x)) ∩ set (map fst B) = {}"
  shows "distinct (map fst U)"
using assms
proof (induction E B arbitrary: U rule: Unification.unify.induct)
  case 1 thus ?case by simp
next
  case (2 f X g Y E B U)
  let ?fvs = "λL. ⋃x ∈ set L. fv (fst x) ∪ fv (snd x)"
  from "2.prems"(1) obtain E' where *: "decompose (Fun f X) (Fun g Y) = Some E'"
    and [simp]: "f = g" "length X = length Y" "E' = zip X Y"
    and **: "Unification.unify (E'@E) B = Some U"
    by (auto split: option.splits)
  hence "⋀t t'. (t,t') ∈ set E' ⟹ fv t ⊆ fv (Fun f X) ∧ fv t' ⊆ fv (Fun g Y)"
    by (metis zip_arg_subterm subtermeq_vars_subset)
  hence "?fvs E' ⊆ fv (Fun f X) ∪ fv (Fun g Y)" by fastforce
  moreover have "fv (Fun f X) ∩ set (map fst B) = {}" "fv (Fun g Y) ∩ set (map fst B) = {}"
    using "2.prems"(3) by auto
  ultimately have "?fvs E' ∩ set (map fst B) = {}" by blast
  moreover have "?fvs E ∩ set (map fst B) = {}" using "2.prems"(3) by auto
  ultimately have "?fvs (E'@E) ∩ set (map fst B) = {}" by auto
  thus ?case using "2.IH"[OF * ** "2.prems"(2)] by metis
next
  case (3 v t E B)
  let ?fvs = "λL. ⋃x ∈ set L. fv (fst x) ∪ fv (snd x)"
  let ?E' = "subst_list (subst v t) E"
  from "3.prems"(3) have "v ∉ set (map fst B)" "fv t ∩ set (map fst B) = {}" by force+
  hence *: "distinct (map fst ((v, t)#B))" using "3.prems"(2) by auto
```

```
  show ?case
  proof (cases "t = Var v")
    case True thus ?thesis using "3.prems" "3.IH"(1) by auto
  next
    case False
    hence "v ∉ fv t" using "3.prems"(1) by auto
    hence "Unification.unify (subst_list (subst v t) E) ((v, t)#B) = Some U"
      using ‹t ≠ Var v› "3.prems"(1) by auto
    moreover have "?fvs ?E' ∩ set (map fst ((v, t)#B)) = {}"
    proof -
      have "v ∉ ?fvs ?E'"
        unfolding subst_list_def subst_def
        by (simp add: ‹v ∉ fv t› subst_remove_var)
      moreover have "?fvs ?E' ⊆ fv t ∪ ?fvs E" by (metis subst_list_singleton_fv_subset)
      hence "?fvs ?E' ∩ set (map fst B) = {}" using "3.prems"(3) by auto
      ultimately show ?thesis by auto
    qed
    ultimately show ?thesis using "3.IH"(2)[OF ‹t ≠ Var v› ‹v ∉ fv t› _ *] by metis
  qed
next
  case (4 f X v E B U)
  let ?fvs = "λL. ⋃x ∈ set L. fv (fst x) ∪ fv (snd x)"
  let ?E' = "subst_list (subst v (Fun f X)) E"
  have *: "?fvs E ∩ set (map fst B) = {}" using "4.prems"(3) by auto
  from "4.prems"(1) have "v ∉ fv (Fun f X)" by force
  from "4.prems"(3) have **: "v ∉ set (map fst B)" "fv (Fun f X) ∩ set (map fst B) = {}" by force+
  hence ***: "distinct (map fst ((v, Fun f X)#B))" using "4.prems"(2) by auto
  from "4.prems"(3) have ****: "?fvs ?E' ∩ set (map fst ((v, Fun f X)#B)) = {}"
  proof -
    have "v ∉ ?fvs ?E'"
      unfolding subst_list_def subst_def
      using ‹v ∉ fv (Fun f X)› subst_remove_var[of v "Fun f X"] by simp
    moreover have "?fvs ?E' ⊆ fv (Fun f X) ∪ ?fvs E" by (metis subst_list_singleton_fv_subset)
    hence "?fvs ?E' ∩ set (map fst B) = {}" using * ** by blast
    ultimately show ?thesis by auto
  qed
  have "Unification.unify (subst_list (subst v (Fun f X)) E) ((v, Fun f X) # B) = Some U"
    using ‹v ∉ fv (Fun f X)› "4.prems"(1) by auto
  thus ?case using "4.IH"[OF ‹v ∉ fv (Fun f X)› _ *** ****] by metis
qed

lemma mgu_None_is_subst_neq:
  fixes s t::"('a,'b) term" and δ::"('a,'b) subst"
  assumes "mgu s t = None"
  shows "s · δ ≠ t · δ"
using assms mgu_always_unifies by force

lemma mgu_None_if_neq_ground:
  assumes "t ≠ t'" "fv t = {}" "fv t' = {}"
  shows "mgu t t' = None"
proof (rule ccontr)
  assume "mgu t t' ≠ None"
  then obtain δ where δ: "mgu t t' = Some δ" by auto
  hence "t · δ = t" "t' · δ = t'" using assms subst_ground_ident by auto
  thus False using assms(1) MGU_is_Unifier[OF mgu_gives_MGU[OF δ]] by auto
qed

lemma mgu_None_commutes:
  "mgu s t = None ⟹ mgu t s = None"
using mgu_complete[of s t]
      Unifier_in_unifiers_singleton[of s _ t]
      Unifier_sym[of t _ s]
      Unifier_in_unifiers_singleton[of t _ s]
```

```
      mgu_sound[of t s]
unfolding is_imgu_def
by fastforce

lemma mgu_img_subterm_subst:
  fixes δ::"('f,'v) subst" and s t u::"('f,'v) term"
  assumes "mgu s t = Some δ" "u ∈ subterms_set (subst_range δ) - range Var"
  shows "u ∈ ((subterms s ∪ subterms t) - range Var) ·set δ"
proof -
  define subterms_tuples::"('f,'v) equation list ⇒ ('f,'v) terms" where subtt_def:
    "subterms_tuples ≡ λE. subterms_set (fst ` set E) ∪ subterms_set (snd ` set E)"
  define subterms_img::"('f,'v) subst ⇒ ('f,'v) terms" where subti_def:
    "subterms_img ≡ λd. subterms_set (subst_range d)"

  define d where "d ≡ λv t. subst v t::('f,'v) subst"
  define V where "V ≡ range Var::('f,'v) terms"
  define R where "R ≡ λd::('f,'v) subst. ((subterms s ∪ subterms t) - V) ·set d"
  define M where "M ≡ λE d. subterms_tuples E ∪ subterms_img d"
  define Q where "Q ≡ (λE d. M E d - V ⊆ R d - V)"
  define Q' where "Q' ≡ (λE d d'. (M E d - V) ·set d' ⊆ (R d - V) ·set (d'::('f,'v) subst))"

  have Q_subst: "Q (subst_list (subst v t') E) (subst_of ((v, t')#B))"
    when v_fv: "v ∉ fv t'" and Q_assm: "Q ((Var v, t')#E) (subst_of B)"
    for v t' E B
  proof -
    define E' where "E' ≡ subst_list (subst v t') E"
    define B' where "B' ≡ subst_of ((v, t')#B)"

    have E': "E' = subst_list (d v t') E"
        and B': "B' = subst_of B ∘s d v t'"
      using subst_of_simps(3)[of "(v, t')"]
      unfolding subst_def E'_def B'_def d_def by simp_all

    have vt_img_subt: "subterms_set (subst_range (d v t')) = subterms t'"
         and vt_dom: "subst_domain (d v t') = {v}"
      using v_fv by (auto simp add: subst_domain_def d_def subst_def)

    have *: "subterms u1 ⊆ subterms_set (fst ` set E)" "subterms u2 ⊆ subterms_set (snd ` set E)"
      when "(u1,u2) ∈ set E" for u1 u2
      using that by auto

    have **: "subterms_set (d v t' ` (fv u ∩ subst_domain (d v t'))) ⊆ subterms t'"
      for u::"('f,'v) term"
      using vt_dom unfolding d_def by force

    have 1: "subterms_tuples E' - V ⊆ (subterms t' - V) ∪ (subterms_tuples E - V ·set d v t')"
      (is "?A ⊆ ?B")
    proof
      fix u assume "u ∈ ?A"
      then obtain u1 u2 where u12:
          "(u1,u2) ∈ set E"
          "u ∈ (subterms (u1 · (d v t')) - V) ∪ (subterms (u2 · (d v t')) - V)"
        unfolding subtt_def subst_list_def E'_def d_def by moura
      hence "u ∈ (subterms t' - V) ∪ (((subterms_tuples E) ·set d v t') - V)"
        using subterms_subst[of u1 "d v t'"] subterms_subst[of u2 "d v t'"]
              *[OF u12(1)] **[of u1] **[of u2]
        unfolding subtt_def subst_list_def by auto
      moreover have
          "(subterms_tuples E ·set d v t') - V ⊆
           (subterms_tuples E - V ·set d v t') ∪ {t'}"
        unfolding subst_def subtt_def V_def d_def by force
      ultimately show "u ∈ ?B" using u12 v_fv by auto
    qed
```

```
    have 2: "subterms_img B' - V ⊆
             (subterms t' - V) ∪ (subterms_img (subst_of B) - V ·set d v t')"
      using B' vt_img_subt subst_img_comp_subset'''[of "subst_of B" "d v t'"]
      unfolding subti_def subst_def V_def by argo

    have 3: "subterms_tuples ((Var v, t')#E) - V = (subterms t' - V) ∪ (subterms_tuples E - V)"
      by (auto simp add: subst_def subtt_def V_def)

    have "fvset (subterms t' - V) ∩ subst_domain (d v t') = {}"
      using v_fv vt_dom fv_subterms[of t'] by fastforce
    hence 4: "subterms t' - V ·set d v t' = subterms t' - V"
      using set_subst_ident[of "subterms t' - range Var" "d v t'"] by (simp add: V_def)

    have "M E' B' - V ⊆ M ((Var v, t')#E) (subst_of B) - V ·set d v t'"
      using 1 2 3 4 unfolding M_def by blast
    moreover have "Q' ((Var v, t')#E) (subst_of B) (d v t')"
      using Q_assm unfolding Q_def Q'_def by auto
    moreover have "R (subst_of B) ·set d v t' = R (subst_of ((v,t')#B))"
      unfolding R_def d_def by auto
    ultimately have
      "M (subst_list (d v t') E) (subst_of ((v, t')#B)) - V ⊆ R (subst_of ((v, t')#B)) - V"
      unfolding Q'_def E'_def B'_def d_def by blast
    thus ?thesis unfolding Q_def M_def R_def d_def by blast
  qed

  have "u ∈ subterms s ∪ subterms t - V ·set subst_of U"
    when assms':
      "unify E B = Some U"
      "u ∈ subtermsset (subst_range (subst_of U)) - V"
      "Q E (subst_of B)"
    for E B U and T::"('f,'v) term list"
    using assms'
  proof (induction E B arbitrary: U rule: Unification.unify.induct)
    case (1 B) thus ?case by (auto simp add: Q_def M_def R_def subti_def)
  next
    case (2 g X h Y E B U)
    from "2.prems"(1) obtain E' where E':
        "decompose (Fun g X) (Fun h Y) = Some E'"
        "g = h" "length X = length Y" "E' = zip X Y"
        "Unification.unify (E'@E) B = Some U"
      by (auto split: option.splits)
    moreover have "subterms_tuples (E'@E) ⊆ subterms_tuples ((Fun g X, Fun h Y)#E)"
    proof
      fix u assume "u ∈ subterms_tuples (E'@E)"
      then obtain u1 u2 where u12: "(u1,u2) ∈ set (E'@E)" "u ∈ subterms u1 ∪ subterms u2"
        unfolding subtt_def by fastforce
      thus "u ∈ subterms_tuples ((Fun g X, Fun h Y)#E)"
      proof (cases "(u1,u2) ∈ set E'")
        case True
        hence "subterms u1 ⊆ subterms (Fun g X)" "subterms u2 ⊆ subterms (Fun h Y)"
          using E'(4) subterms_subset params_subterms subsetCE
          by (metis set_zip_leftD, metis set_zip_rightD)
        thus ?thesis using u12 unfolding subtt_def by auto
      next
        case False thus ?thesis using u12 unfolding subtt_def by fastforce
      qed
     qed
    hence "Q (E'@E) (subst_of B)" using "2.prems"(3) unfolding Q_def M_def by blast
    ultimately show ?case using "2.IH"[of E' U] "2.prems" by meson
  next
    case (3 v t' E B)
    show ?case
```

```
    proof (cases "t' = Var v")
      case True thus ?thesis
        using "3.prems" "3.IH"(1) unfolding Q_def M_def V_def subtt_def by auto
    next
      case False
      hence 1: "v ∉ fv t'" using "3.prems"(1) by auto
      hence "unify (subst_list (subst v t') E) ((v, t')#B) = Some U"
        using False "3.prems"(1) by auto
      thus ?thesis
        using Q_subst[OF 1 "3.prems"(3)]
              "3.IH"(2)[OF False 1 _ "3.prems"(2)]
        by metis
    qed
  next
    case (4 g X v E B U)
    have 1: "v ∉ fv (Fun g X)" using "4.prems"(1) not_None_eq by fastforce
    hence 2: "unify (subst_list (subst v (Fun g X)) E) ((v, Fun g X)#B) = Some U"
      using "4.prems"(1) by auto

    have 3: "Q ((Var v, Fun g X)#E) (subst_of B)"
      using "4.prems"(3) unfolding Q_def M_def subtt_def by auto

    show ?case
      using Q_subst[OF 1 3] "4.IH"[OF 1 2 "4.prems"(2)]
      by metis
  qed
  moreover obtain D where "unify [(s, t)] [] = Some D" "δ = subst_of D"
    using assms(1) by (auto split: option.splits)
  moreover have "Q [(s,t)] (subst_of [])"
    unfolding Q_def M_def R_def subtt_def subti_def
    by force
  ultimately show ?thesis using assms(2) unfolding V_def by auto
qed

lemma mgu_img_consts:
  fixes δ::"('f,'v) subst" and s t::"('f,'v) term" and c::'f and z::'v
  assumes "mgu s t = Some δ" "Fun c [] ∈ subterms_set (subst_range δ)"
  shows "Fun c [] ∈ subterms s ∪ subterms t"
proof -
  obtain u where "u ∈ (subterms s ∪ subterms t) - range Var" "u · δ = Fun c []"
    using mgu_img_subterm_subst[OF assms(1), of "Fun c []"] assms(2) by force
  thus ?thesis by (cases u) auto
qed

lemma mgu_img_consts':
  fixes δ::"('f,'v) subst" and s t::"('f,'v) term" and c::'f and z::'v
  assumes "mgu s t = Some δ" "δ z = Fun c []"
  shows "Fun c [] ⊑ s ∨ Fun c [] ⊑ t"
using mgu_img_consts[OF assms(1)] assms(2)
by (metis Un_iff in_subterms_Union subst_imgI term.distinct(1))

lemma mgu_img_composed_var_term:
  fixes δ::"('f,'v) subst" and s t::"('f,'v) term" and f::'f and Z::"'v list"
  assumes "mgu s t = Some δ" "Fun f (map Var Z) ∈ subterms_set (subst_range δ)"
  shows "∃Z'. map δ Z' = map Var Z ∧ Fun f (map Var Z') ∈ subterms s ∪ subterms t"
proof -
  obtain u where u: "u ∈ (subterms s ∪ subterms t) - range Var" "u · δ = Fun f (map Var Z)"
    using mgu_img_subterm_subst[OF assms(1), of "Fun f (map Var Z)"] assms(2) by fastforce
  then obtain T where T: "u = Fun f T" "map (λt. t · δ) T = map Var Z" by (cases u) auto
  have "∀t ∈ set T. ∃x. t = Var x" using T(2) by (induct T arbitrary: Z) auto
  then obtain Z' where Z': "map Var Z' = T" by (metis ex_map_conv)
  hence "map δ Z' = map Var Z" using T(2) by (induct Z' arbitrary: T Z) auto
  thus ?thesis using u(1) T(1) Z' by auto
```

qed

2.3.9 Lemmata: The "Inequality Lemmata"

Subterm injectivity (a stronger injectivity property)

```
definition subterm_inj_on where
  "subterm_inj_on f A ≡ ∀x∈A. ∀y∈A. (∃v. v ⊑ f x ∧ v ⊑ f y) ⟶ x = y"

lemma subterm_inj_on_imp_inj_on: "subterm_inj_on f A ⟹ inj_on f A"
unfolding subterm_inj_on_def inj_on_def by fastforce

lemma subst_inj_on_is_bij_betw:
  "inj_on ϑ (subst_domain ϑ) = bij_betw ϑ (subst_domain ϑ) (subst_range ϑ)"
unfolding inj_on_def bij_betw_def by auto

lemma subterm_inj_on_alt_def:
    "subterm_inj_on f A ⟷
     (inj_on f A ∧ (∀s ∈ f`A. ∀u ∈ f`A. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u))"
    (is "?A ⟷ ?B")
unfolding subterm_inj_on_def inj_on_def by fastforce

lemma subterm_inj_on_alt_def':
    "subterm_inj_on ϑ (subst_domain ϑ) ⟷
     (inj_on ϑ (subst_domain ϑ) ∧
      (∀s ∈ subst_range ϑ. ∀u ∈ subst_range ϑ. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u))"
    (is "?A ⟷ ?B")
by (metis subterm_inj_on_alt_def subst_range.simps)

lemma subterm_inj_on_subset:
  assumes "subterm_inj_on f A"
    and "B ⊆ A"
  shows "subterm_inj_on f B"
proof -
  have "inj_on f A" "∀s∈f ` A. ∀u∈f ` A. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u"
    using subterm_inj_on_alt_def[of f A] assms(1) by auto
  moreover have "f ` B ⊆ f ` A" using assms(2) by auto
  ultimately have "inj_on f B" "∀s∈f ` B. ∀u∈f ` B. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u"
    using inj_on_subset[of f A] assms(2) by blast+
  thus ?thesis by (metis subterm_inj_on_alt_def)
qed

lemma inj_subst_unif_consts:
  fixes ℐ ϑ σ::"('f,'v) subst" and s t::"('f,'v) term"
  assumes ϑ: "subterm_inj_on ϑ (subst_domain ϑ)" "∀x ∈ (fv s ∪ fv t) - X. ∃c. ϑ x = Fun c []"
            "subterms_set (subst_range ϑ) ∩ (subterms s ∪ subterms t) = {}" "ground (subst_range ϑ)"
            "subst_domain ϑ ∩ X = {}"
  and ℐ: "ground (subst_range ℐ)" "subst_domain ℐ = subst_domain ϑ"
  and unif: "Unifier σ (s · ϑ) (t · ϑ)"
  shows "∃δ. Unifier δ (s · ℐ) (t · ℐ)"
proof -
  let ?xs = "subst_domain ϑ"
  let ?ys = "(fv s ∪ fv t) - ?xs"

  have "∃δ::('f,'v) subst. s · δ = t · δ" by (metis subst_subst_compose unif)
  then obtain δ::"('f,'v) subst" where δ: "mgu s t = Some δ"
    using mgu_always_unifies by moura
  have 1: "∃σ::('f,'v) subst. s · ϑ · σ = t · ϑ · σ" by (metis unif)
  have 2: "⋀γ::('f,'v) subst. s · ϑ · γ = t · ϑ · γ ⟹ δ ≼∘ ϑ ∘s γ" using mgu_gives_MGU[OF δ] by
simp
  have 3: "⋀(z::'v) (c::'f). δ z = Fun c [] ⟹ Fun c [] ⊑ s ∨ Fun c [] ⊑ t"
    by (rule mgu_img_consts'[OF δ])
  have 4: "subst_domain δ ∩ range_vars δ = {}"
```

```
    by (metis mgu_gives_wellformed_subst[OF δ] wf_subst_def)
  have 5: "subst_domain δ ∪ range_vars δ ⊆ fv s ∪ fv t"
    by (metis mgu_gives_wellformed_MGU[OF δ] wf_MGU_def)

  { fix x and γ::"('f,'v) subst" assume "x ∈ subst_domain ϑ"
    hence "(ϑ ∘s γ) x = ϑ x"
      using ϑ(4) ident_comp_subst_trm_if_disj[of γ ϑ]
      unfolding range_vars_alt_def by fast
  }
  then obtain τ::"('f,'v) subst" where τ: "∀x ∈ subst_domain ϑ. ϑ x = (δ ∘s τ) x" using 1 2 by
moura

  have *: "⋀x. x ∈ subst_domain δ ∩ subst_domain ϑ ⟹ ∃y ∈ ?ys. δ x = Var y"
  proof -
    fix x assume "x ∈ subst_domain δ ∩ ?xs"
    hence x: "x ∈ subst_domain δ" "x ∈ subst_domain ϑ" by auto
    then obtain c where c: "ϑ x = Fun c []" using ϑ(2,5) 5 by moura
    hence *: "(δ ∘s τ) x = Fun c []" using τ x by fastforce
    hence **: "x ∈ subst_domain (δ ∘s τ)" "Fun c [] ∈ subst_range (δ ∘s τ)"
      by (auto simp add: subst_domain_def)
    have "δ x = Fun c [] ∨ (∃z. δ x = Var z ∧ τ z = Fun c [])"
      by (rule subst_img_comp_subset_const'[OF *])
    moreover have "δ x ≠ Fun c []"
    proof (rule ccontr)
      assume "¬δ x ≠ Fun c []"
      hence "Fun c [] ⊑ s ∨ Fun c [] ⊑ t" using 3 by metis
      moreover have "∀u ∈ subst_range ϑ. u ∉ subterms s ∪ subterms t"
        using ϑ(3) by force
      hence "Fun c [] ∉ subterms s ∪ subterms t"
        by (metis c ‹ground (subst_range ϑ)›x(2) ground_subst_dom_iff_img)
      ultimately show False by auto
    qed
    moreover have "∀x' ∈ subst_domain ϑ. δ x ≠ Var x'"
    proof (rule ccontr)
      assume "¬(∀x' ∈ subst_domain ϑ. δ x ≠ Var x')"
      then obtain x' where x': "x' ∈ subst_domain ϑ" "δ x = Var x'" by moura
      hence "τ x' = Fun c []" "(δ ∘s τ) x = Fun c []" using * unfolding subst_compose_def by auto
      moreover have "x ≠ x'"
        using x(1) x'(2) 4
        by (auto simp add: subst_domain_def)
      moreover have "x' ∉ subst_domain δ"
        using x'(2) mgu_eliminates_dom[OF δ]
        by (metis (no_types) subst_elim_def subst_apply_term.simps(1) vars_iff_subterm_or_eq)
      moreover have "(δ ∘s τ) x = ϑ x" "(δ ∘s τ) x' = ϑ x'" using τ x(2) x'(1) by auto
      ultimately show False
        using subterm_inj_on_imp_inj_on[OF ϑ(1)] *
        by (simp add: inj_on_def subst_compose_def x'(2) subst_domain_def)
    qed
    ultimately show "∃y ∈ ?ys. δ x = Var y"
      by (metis 5 x(2) subtermeqI' vars_iff_subtermeq DiffI Un_iff subst_fv_imgI sup.orderE)
  qed

  have **: "inj_on δ (subst_domain δ ∩ ?xs)"
  proof (intro inj_onI)
    fix x y assume *:
      "x ∈ subst_domain δ ∩ subst_domain ϑ" "y ∈ subst_domain δ ∩ subst_domain ϑ" "δ x = δ y"
    hence "(δ ∘s τ) x = (δ ∘s τ) y" unfolding subst_compose_def by auto
    hence "ϑ x = ϑ y" using τ * by auto
    thus "x = y" using inj_onD[OF subterm_inj_on_imp_inj_on[OF ϑ(1)]] *(1,2) by simp
  qed

  define α where "α = (λy'. if Var y' ∈ δ ` (subst_domain δ ∩ ?xs)
                          then Var ((inv_into (subst_domain δ ∩ ?xs) δ) (Var y'))
```

```
                              else Var y'::('f,'v) term)"
  have a1: "Unifier (δ ∘s α) s t" using mgu_gives_MGU[OF δ] by auto

  define δ' where "δ' = δ ∘s α"
  have d1: "subst_domain δ' ⊆ ?ys"
  proof
    fix z assume z: "z ∈ subst_domain δ'"
    have "z ∈ ?xs ⟹ z ∉ subst_domain δ'"
    proof (cases "z ∈ subst_domain δ")
      case True
      moreover assume "z ∈ ?xs"
      ultimately have z_in: "z ∈ subst_domain δ ∩ ?xs" by simp
      then obtain y where y: "δ z = Var y" "y ∈ ?ys" using * by moura
      hence "α y = Var ((inv_into (subst_domain δ ∩ ?xs) δ) (Var y))"
        using α_def z_in by simp
      hence "α y = Var z" by (metis y(1) z_in ** inv_into_f_eq)
      hence "δ' z = Var z" using δ'_def y(1) subst_compose_def[of δ α] by simp
      thus ?thesis by (simp add: subst_domain_def)
    next
      case False
      hence "δ z = Var z" by (simp add: subst_domain_def)
      moreover assume "z ∈ ?xs"
      hence "α z = Var z" using α_def * by force
      ultimately show ?thesis
        using δ'_def subst_compose_def[of δ α]
        by (simp add: subst_domain_def)
    qed
    moreover have "subst_domain α ⊆ range_vars δ"
      unfolding δ'_def α_def range_vars_alt_def
      by (auto simp add: subst_domain_def)
    hence "subst_domain δ' ⊆ subst_domain δ ∪ range_vars δ"
      using subst_domain_compose[of δ α] unfolding δ'_def by blast
    ultimately show "z ∈ ?ys" using 5 z by auto
  qed
  have d2: "Unifier (δ' ∘s 𝓘) s t" using a1 δ'_def by auto
  have d3: "𝓘 ∘s δ' ∘s 𝓘 = δ' ∘s 𝓘"
  proof -
    { fix z::'v assume z: "z ∈ ?xs"
      then obtain u where u: "𝓘 z = u" "fv u = {}" using 𝓘 by auto
      hence "(𝓘 ∘s δ' ∘s 𝓘) z = u" by (simp add: subst_compose subst_ground_ident)
      moreover have "z ∉ subst_domain δ'" using d1 z by auto
      hence "δ' z = Var z" by (simp add: subst_domain_def)
      hence "(δ' ∘s 𝓘) z = u" using u(1) by (simp add: subst_compose)
      ultimately have "(𝓘 ∘s δ' ∘s 𝓘) z = (δ' ∘s 𝓘) z" by metis
    } moreover {
      fix z::'v assume "z ∈ ?ys"
      hence "z ∉ subst_domain 𝓘" using 𝓘(2) by auto
      hence "(𝓘 ∘s δ' ∘s 𝓘) z = (δ' ∘s 𝓘) z" by (simp add: subst_compose subst_domain_def)
    } moreover {
      fix z::'v assume "z ∉ ?xs" "z ∉ ?ys"
      hence "𝓘 z = Var z" "δ' z = Var z" using 𝓘(2) d1 by blast+
      hence "(𝓘 ∘s δ' ∘s 𝓘) z = (δ' ∘s 𝓘) z" by (simp add: subst_compose)
    } ultimately show ?thesis by auto
  qed

  from d2 d3 have "Unifier (δ' ∘s 𝓘) (s · 𝓘) (t · 𝓘)" by (metis subst_subst_compose)
  thus ?thesis by metis
qed

lemma inj_subst_unif_comp_terms:
  fixes 𝓘 ϑ σ::"('f,'v) subst" and s t::"('f,'v) term"
  assumes ϑ: "subterm_inj_on ϑ (subst_domain ϑ)" "ground (subst_range ϑ)"
             "subterms_set (subst_range ϑ) ∩ (subterms s ∪ subterms t) = {}"
```

```
            "(fv s ∪ fv t) - subst_domain ϑ ⊆ X"
  and tfr: "∀f U. Fun f U ∈ subterms s ∪ subterms t ⟶ U = [] ∨ (∃u ∈ set U. u ∉ Var ` X)"
  and ℐ: "ground (subst_range ℐ)" "subst_domain ℐ = subst_domain ϑ"
  and unif: "Unifier σ (s · ϑ) (t · ϑ)"
  shows "∃δ. Unifier δ (s · ℐ) (t · ℐ)"
proof -
  let ?xs = "subst_domain ϑ"
  let ?ys = "(fv s ∪ fv t) - ?xs"

  have "ground (subst_range ϑ)" using ϑ(2) by auto

  have "∃δ::('f,'v) subst. s · δ = t · δ" by (metis subst_subst_compose unif)
  then obtain δ::"('f,'v) subst" where δ: "mgu s t = Some δ"
    using mgu_always_unifies by moura
  have 1: "∃σ::('f,'v) subst. s · ϑ · σ = t · ϑ · σ" by (metis unif)
  have 2: "⋀γ::('f,'v) subst. s · ϑ · γ = t · ϑ · γ ⟹ δ ≼∘ ϑ ∘_s γ" using mgu_gives_MGU[OF δ] by simp
  have 3: "⋀(z::'v) (c::'f). Fun c [] ⊑ δ z ⟹ Fun c [] ⊑ s ∨ Fun c [] ⊑ t"
    using mgu_img_consts[OF δ] by force
  have 4: "subst_domain δ ∩ range_vars δ = {}"
    using mgu_gives_wellformed_subst[OF δ]
    by (metis wf_subst_def)
  have 5: "subst_domain δ ∪ range_vars δ ⊆ fv s ∪ fv t"
    using mgu_gives_wellformed_MGU[OF δ]
    by (metis wf_MGU_def)

  { fix x and γ::"('f,'v) subst" assume "x ∈ subst_domain ϑ"
    hence "(ϑ ∘_s γ) x = ϑ x"
      using ‹ground (subst_range ϑ)› ident_comp_subst_trm_if_disj[of γ ϑ x]
      unfolding range_vars_alt_def by blast
  }
  then obtain τ::"('f,'v) subst" where τ: "∀x ∈ subst_domain ϑ. ϑ x = (δ ∘_s τ) x" using 1 2 by moura

  have ***: "⋀x. x ∈ subst_domain δ ∩ subst_domain ϑ ⟹ fv (δ x) ⊆ ?ys"
  proof -
    fix x assume "x ∈ subst_domain δ ∩ ?xs"
    hence x: "x ∈ subst_domain δ" "x ∈ subst_domain ϑ" by auto
    moreover have "¬(∃x' ∈ ?xs. x' ∈ fv (δ x))"
    proof (rule ccontr)
      assume "¬¬(∃x' ∈ ?xs. x' ∈ fv (δ x))"
      then obtain x' where x': "x' ∈ fv (δ x)" "x' ∈ ?xs" by metis
      have "x ≠ x'" "x' ∉ subst_domain δ" "δ x' = Var x'"
        using 4 x(1) x'(1) unfolding range_vars_alt_def by auto
      hence "(δ ∘_s τ) x' ⊑ (δ ∘_s τ) x" "τ x' = (δ ∘_s τ) x'"
        using τ x(2) x'(2)
        by (metis subst_compose subst_mono vars_iff_subtermeq x'(1),
            metis subst_apply_term.simps(1) subst_compose_def)
      hence "ϑ x' ⊑ ϑ x" using τ x(2) x'(2) by auto
      thus False
        using ϑ(1) x'(2) x(2) ‹x ≠ x'›
        unfolding subterm_inj_on_def
        by (meson subtermeqI')
    qed
    ultimately show "fv (δ x) ⊆ ?ys"
      using 5 subst_dom_vars_in_subst[of x δ] subst_fv_imgI[of δ x]
      by blast
  qed

  have **: "inj_on δ (subst_domain δ ∩ ?xs)"
  proof (intro inj_onI)
    fix x y assume *:
      "x ∈ subst_domain δ ∩ subst_domain ϑ" "y ∈ subst_domain δ ∩ subst_domain ϑ" "δ x = δ y"
```

```
    hence "(δ ∘s τ) x = (δ ∘s τ) y" unfolding subst_compose_def by auto
    hence "ϑ x = ϑ y" using τ * by auto
    thus "x = y" using inj_onD[OF subterm_inj_on_imp_inj_on[OF ϑ(1)]] *(1,2) by simp
  qed

  have *: "⋀x. x ∈ subst_domain δ ∩ subst_domain ϑ ⟹ ∃y ∈ ?ys. δ x = Var y"
  proof (rule ccontr)
    fix xi assume xi_assms: "xi ∈ subst_domain δ ∩ subst_domain ϑ" "¬(∃y ∈ ?ys. δ xi = Var y)"
    hence xi_ϑ: "xi ∈ subst_domain ϑ" and δ_xi_comp: "¬(∃y. δ xi = Var y)"
      using ***[of xi] 5 by auto
    then obtain f T where f: "δ xi = Fun f T" by (cases "δ xi") moura

    have "∃g Y'. Y' ≠ [] ∧ Fun g (map Var Y') ⊑ δ xi ∧ set Y' ⊆ ?ys"
    proof -
      have "∀c. Fun c [] ⊑ δ xi ⟶ Fun c [] ⊑ ϑ xi"
        using τ xi_ϑ by (metis const_subterm_subst subst_compose)
      hence 1: "∀c. ¬(Fun c [] ⊑ δ xi)"
        using 3[of _ xi] xi_ϑ ϑ(3)
        by auto

      have "¬(∃x. δ xi = Var x)" using f by auto
      hence "∃g S. Fun g S ⊑ δ xi ∧ (∀s ∈ set S. (∃c. s = Fun c []) ∨ (∃x. s = Var x))"
        using nonvar_term_has_composed_shallow_term[of "δ xi"] by auto
      then obtain g S where gS: "Fun g S ⊑ δ xi" "∀s ∈ set S. (∃c. s = Fun c []) ∨ (∃x. s = Var
x)"
        by moura

      have "∀s ∈ set S. ∃x. s = Var x"
        using 1 term.order_trans gS
        by (metis (no_types, lifting) UN_I term.order_refl subsetCE subterms.simps(2) sup_ge2)
      then obtain S' where 2: "map Var S' = S" by (metis ex_map_conv)

      have "S ≠ []" using 1 term.order_trans[OF _ gS(1)] by fastforce
      hence 3: "S' ≠ []" "Fun g (map Var S') ⊑ δ xi" using gS(1) 2 by auto

      have "set S' ⊆ fv (Fun g (map Var S'))" by simp
      hence 4: "set S' ⊆ fv (δ xi)" using 3(2) fv_subterms by force

      show ?thesis using ***[OF xi_assms(1)] 2 3 4 by auto
    qed
    then obtain g Y' where g: "Y' ≠ []" "Fun g (map Var Y') ⊑ δ xi" "set Y' ⊆ ?ys" by moura
    then obtain X where X: "map δ X = map Var Y'" "Fun g (map Var X) ∈ subterms s ∪ subterms t"
      using mgu_img_composed_var_term[OF δ, of g Y'] by force
    hence "∃(u::('f,'v) term) ∈ set (map Var X). u ∉ Var ` ?ys"
      using ϑ(4) tfr g(1) by fastforce
    then obtain j where j: "j < length X" "X ! j ∉ ?ys"
      by (metis image_iff[of _ Var "fv s ∪ fv t - subst_domain ϑ"] nth_map[of _ X Var]
                in_set_conv_nth[of _ "map Var X"] length_map[of Var X])

    define yj' where yj': "yj' ≡ Y' ! j"
    define xj where xj: "xj ≡ X ! j"

    have "xj ∈ fv s ∪ fv t"
      using j X(1) g(3) 5 xj yj'
      by (metis length_map nth_map term.simps(1) in_set_conv_nth le_supE subsetCE subst_domI)
    hence xj_ϑ: "xj ∈ subst_domain ϑ" using j unfolding xj by simp

    have len: "length X = length Y'" by (rule map_eq_imp_length_eq[OF X(1)])

    have "Var yj' ⊑ δ xi"
      using term.order_trans[OF _ g(2)] j(1) len unfolding yj' by auto
    hence "τ yj' ⊑ ϑ xi"
      using τ xi_ϑ by (metis subst_apply_term.simps(1) subst_compose_def subst_mono)
```

```
  moreover have δ_xj_var: "Var yj' = δ xj"
    using X(1) len j(1) nth_map
    unfolding xj yj' by metis
  hence "τ yj' = ϑ xj" using τ xj_ϑ by (metis subst_apply_term.simps(1) subst_compose_def)
  moreover have "xi ≠ xj" using δ_xi_comp δ_xj_var by auto
  ultimately show False using ϑ(1) xi_ϑ xj_ϑ unfolding subterm_inj_on_def by blast
qed

define α where "α = (λy'. if Var y' ∈ δ ` (subst_domain δ ∩ ?xs)
                          then Var ((inv_into (subst_domain δ ∩ ?xs) δ) (Var y'))
                          else Var y'::('f,'v) term)"
have a1: "Unifier (δ ∘s α) s t" using mgu_gives_MGU[OF δ] by auto

define δ' where "δ' = δ ∘s α"
have d1: "subst_domain δ' ⊆ ?ys"
proof
  fix z assume z: "z ∈ subst_domain δ'"
  have "z ∈ ?xs ⟹ z ∉ subst_domain δ'"
  proof (cases "z ∈ subst_domain δ")
    case True
    moreover assume "z ∈ ?xs"
    ultimately have z_in: "z ∈ subst_domain δ ∩ ?xs" by simp
    then obtain y where y: "δ z = Var y" "y ∈ ?ys" using * by moura
    hence "α y = Var ((inv_into (subst_domain δ ∩ ?xs) δ) (Var y))"
      using α_def z_in by simp
    hence "α y = Var z" by (metis y(1) z_in ** inv_into_f_eq)
    hence "δ' z = Var z" using δ'_def y(1) subst_compose_def[of δ α] by simp
    thus ?thesis by (simp add: subst_domain_def)
  next
    case False
    hence "δ z = Var z" by (simp add: subst_domain_def)
    moreover assume "z ∈ ?xs"
    hence "α z = Var z" using α_def * by force
    ultimately show ?thesis using δ'_def subst_compose_def[of δ α] by (simp add: subst_domain_def)
  qed
  moreover have "subst_domain α ⊆ range_vars δ"
    unfolding δ'_def α_def range_vars_alt_def subst_domain_def
    by auto
  hence "subst_domain δ' ⊆ subst_domain δ ∪ range_vars δ"
    using subst_domain_compose[of δ α]
    unfolding δ'_def by blast
  ultimately show "z ∈ ?ys" using 5 z by blast
qed
have d2: "Unifier (δ' ∘s ℐ) s t" using a1 δ'_def by auto
have d3: "ℐ ∘s δ' ∘s ℐ = δ' ∘s ℐ"
proof -
  { fix z::'v assume z: "z ∈ ?xs"
    then obtain u where u: "ℐ z = u" "fv u = {}" using ℐ by auto
    hence "(ℐ ∘s δ' ∘s ℐ) z = u" by (simp add: subst_compose subst_ground_ident)
    moreover have "z ∉ subst_domain δ'" using d1 z by auto
    hence "δ' z = Var z" by (simp add: subst_domain_def)
    hence "(δ' ∘s ℐ) z = u" using u(1) by (simp add: subst_compose)
    ultimately have "(ℐ ∘s δ' ∘s ℐ) z = (δ' ∘s ℐ) z" by metis
  } moreover {
    fix z::'v assume "z ∈ ?ys"
    hence "z ∉ subst_domain ℐ" using ℐ(2) by auto
    hence "(ℐ ∘s δ' ∘s ℐ) z = (δ' ∘s ℐ) z" by (simp add: subst_compose subst_domain_def)
  } moreover {
    fix z::'v assume "z ∉ ?xs" "z ∉ ?ys"
    hence "ℐ z = Var z" "δ' z = Var z" using ℐ(2) d1 by blast+
    hence "(ℐ ∘s δ' ∘s ℐ) z = (δ' ∘s ℐ) z" by (simp add: subst_compose)
  } ultimately show ?thesis by auto
qed
```

```
  from d2 d3 have "Unifier (δ' ∘s ℐ) (s · ℐ) (t · ℐ)" by (metis subst_subst_compose)
  thus ?thesis by metis
qed

context
begin
private lemma sat_ineq_subterm_inj_subst_aux:
  fixes ℐ::"('f,'v) subst"
  assumes "Unifier σ (s · ℐ) (t · ℐ)" "ground (subst_range ℐ)"
          "(fv s ∪ fv t) - X ⊆ subst_domain ℐ" "subst_domain ℐ ∩ X = {}"
  shows "∃δ::('f,'v) subst. subst_domain δ = X ∧ ground (subst_range δ) ∧ s · δ · ℐ = t · δ · ℐ"
proof -
  have "∃σ. Unifier σ (s · ℐ) (t · ℐ) ∧ interpretation_subst σ"
  proof -
    obtain ℐ'::"('f,'v) subst" where *: "interpretation_subst ℐ'"
      using interpretation_subst_exists by metis
    hence "Unifier (σ ∘s ℐ') (s · ℐ) (t · ℐ)" using assms(1) by simp
    thus ?thesis using * interpretation_comp by blast
  qed
  then obtain σ' where σ': "Unifier σ' (s · ℐ) (t · ℐ)" "interpretation_subst σ'" by moura

  define σ'' where "σ'' = rm_vars (UNIV - X) σ'"

  have *: "fv (s · ℐ) ⊆ X" "fv (t · ℐ) ⊆ X"
    using assms(2,3) subst_fv_unfold_ground_img[of ℐ]
    unfolding range_vars_alt_def
    by (simp_all add: Diff_subset_conv Un_commute)
  hence **: "subst_domain σ'' = X" "ground (subst_range σ'')"
    using rm_vars_img_subset[of "UNIV - X" σ'] rm_vars_dom[of "UNIV - X" σ'] σ'(2)
    unfolding σ''_def by auto
  hence "⋀t. t · ℐ · σ'' = t · σ'' · ℐ"
    using subst_eq_if_disjoint_vars_ground[OF _ _ assms(2)] assms(4) by blast
  moreover have "Unifier σ'' (s · ℐ) (t · ℐ)"
    using Unifier_dom_restrict[OF σ'(1)] σ''_def * by blast
  ultimately show ?thesis using ** by auto
qed
```

The "inequality lemma": This lemma gives sufficient syntactic conditions for finding substitutions ϑ under which terms s and t are not unifiable.

This is useful later when establishing the typing results since we there want to find well-typed solutions to inequality constraints / "negative checks" constraints, and this lemma gives conditions for protocols under which such constraints are well-typed satisfiable if satisfiable.

```
lemma sat_ineq_subterm_inj_subst:
  fixes ϑ ℐ δ::"('f,'v) subst"
  assumes ϑ: "subterm_inj_on ϑ (subst_domain ϑ)"
             "ground (subst_range ϑ)"
             "subst_domain ϑ ∩ X = {}"
             "subterms_set (subst_range ϑ) ∩ (subterms s ∪ subterms t) = {}"
             "(fv s ∪ fv t) - subst_domain ϑ ⊆ X"
  and tfr: "(∀x ∈ (fv s ∪ fv t) - X. ∃c. ϑ x = Fun c []) ∨
            (∀f U. Fun f U ∈ subterms s ∪ subterms t ⟶ U = [] ∨ (∃u ∈ set U. u ∉ Var ` X))"
  and ℐ: "∀δ::('f,'v) subst. subst_domain δ = X ∧ ground (subst_range δ) ⟶ s · δ · ℐ ≠ t · δ · ℐ"
         "(fv s ∪ fv t) - X ⊆ subst_domain ℐ" "subst_domain ℐ ∩ X = {}" "ground (subst_range ℐ)"
         "subst_domain ℐ = subst_domain ϑ"
  and δ: "subst_domain δ = X" "ground (subst_range δ)"
  shows "s · δ · ϑ ≠ t · δ · ϑ"
proof -
  have "∀σ. ¬Unifier σ (s · ℐ) (t · ℐ)"
    by (metis ℐ(1) sat_ineq_subterm_inj_subst_aux[OF _ ℐ(4,2,3)])
  hence "¬Unifier δ (s · ϑ) (t · ϑ)"
    using inj_subst_unif_consts[OF ϑ(1) _ ϑ(4,2,3) ℐ(4,5)]
          inj_subst_unif_comp_terms[OF ϑ(1,2,4,5) _ ℐ(4,5)]
```

```
        tfr
    by metis
  moreover have "subst_domain δ ∩ subst_domain ϑ = {}" using ϑ(2,3) δ(1) by auto
  ultimately show ?thesis using δ subst_eq_if_disjoint_vars_ground[OF _ ϑ(2) δ(2)] by metis
qed
end

lemma ineq_subterm_inj_cond_subst:
  assumes "X ∩ range_vars ϑ = {}"
  and "∀f T. Fun f T ∈ subterms_set S ⟶ T = [] ∨ (∃u ∈ set T. u ∉ Var`X)"
  shows "∀f T. Fun f T ∈ subterms_set (S ·_set ϑ) ⟶ T = [] ∨ (∃u ∈ set T. u ∉ Var`X)"
proof (intro allI impI)
  let ?M = "λS. subterms_set S ·_set ϑ"
  let ?N = "λS. subterms_set (ϑ ` (fv_set S ∩ subst_domain ϑ))"

  fix f T assume "Fun f T ∈ subterms_set (S ·_set ϑ)"
  hence 1: "Fun f T ∈ ?M S ∨ Fun f T ∈ ?N S"
    using subterms_subst[of _ ϑ] by auto

  have 2: "Fun f T ∈ subterms_set (subst_range ϑ) ⟹ ∀u ∈ set T. u ∉ Var`X"
    using fv_subset_subterms[of "Fun f T" "subst_range ϑ"] assms(1)
    unfolding range_vars_alt_def by force

  have 3: "∀x ∈ subst_domain ϑ. ϑ x ∉ Var`X"
  proof
    fix x assume "x ∈ subst_domain ϑ"
    hence "fv (ϑ x) ⊆ range_vars ϑ"
      using subst_dom_vars_in_subst subst_fv_imgI
      unfolding range_vars_alt_def by auto
    thus "ϑ x ∉ Var`X" using assms(1) by auto
  qed

  show "T = [] ∨ (∃s ∈ set T. s ∉ Var`X)" using 1
  proof
    assume "Fun f T ∈ ?M S"
    then obtain u where u: "u ∈ subterms_set S" "u · ϑ = Fun f T" by fastforce
    show ?thesis
    proof (cases u)
      case (Var x)
      hence "Fun f T ∈ subst_range ϑ" using u(2) by (simp add: subst_domain_def)
      hence "∀u ∈ set T. u ∉ Var`X" using 2 by force
      thus ?thesis by auto
    next
      case (Fun g S)
      hence "S = [] ∨ (∃u ∈ set S. u ∉ Var`X)" using assms(2) u(1) by metis
      thus ?thesis
      proof
        assume "S = []" thus ?thesis using u(2) Fun by simp
      next
        assume "∃u ∈ set S. u ∉ Var`X"
        then obtain u' where u': "u' ∈ set S" "u' ∉ Var`X" by moura
        hence "u' · ϑ ∈ set T" using u(2) Fun by auto
        thus ?thesis using u'(2) 3 by (cases u') force+
      qed
    qed
  next
    assume "Fun f T ∈ ?N S"
    thus ?thesis using 2 by force
  qed
qed
```

2.3.10 Lemmata: Sufficient Conditions for Term Matching

Injective substitutions from variables to variables are invertible

```
definition subst_var_inv where
  "subst_var_inv δ X ≡ (λx. if Var x ∈ δ ` X then Var ((inv_into X δ) (Var x)) else Var x)"

lemma inj_var_ran_subst_is_invertible:
  assumes δ_inj_on_t: "inj_on δ (fv t)"
    and δ_var_on_t: "δ ` fv t ⊆ range Var"
  shows "t = t · δ ∘s subst_var_inv δ (fv t)"
proof -
  have "δ x · subst_var_inv δ (fv t) = Var x" when x: "x ∈ fv t" for x
  proof -
    obtain y where y: "δ x = Var y" using x δ_var_on_t by auto
    hence "Var y ∈ δ ` (fv t)" using x by simp
    thus ?thesis using y inv_into_f_eq[OF δ_inj_on_t x y] unfolding subst_var_inv_def by simp
  qed
  thus ?thesis by (simp add: subst_compose_def trm_subst_ident'')
qed
```

Sufficient conditions for matching unifiable terms

```
lemma inj_var_ran_unifiable_has_subst_match:
  assumes "t · δ = s · δ" "inj_on δ (fv t)" "δ ` fv t ⊆ range Var"
  shows "t = s · δ ∘s subst_var_inv δ (fv t)"
using assms inj_var_ran_subst_is_invertible by fastforce

end
```

2.4 Dolev-Yao Intruder Model (Intruder_Deduction)

```
theory Intruder_Deduction
imports Messages More_Unification
begin
```

2.4.1 Syntax for the Intruder Deduction Relations

```
consts INTRUDER_SYNTH::"('f,'v) terms ⇒ ('f,'v) term ⇒ bool" (infix "⊢c" 50)
consts INTRUDER_DEDUCT::"('f,'v) terms ⇒ ('f,'v) term ⇒ bool" (infix "⊢" 50)
```

2.4.2 Intruder Model Locale

The intruder model is parameterized over arbitrary function symbols (e.g, cryptographic operators) and variables. It requires three functions: - *arity* that assigns an arity to each function symbol. - *public* that partitions the function symbols into those that will be available to the intruder and those that will not. - *Ana*, the analysis interface, that defines how messages can be decomposed (e.g., decryption).

```
locale intruder_model =
  fixes arity :: "'fun ⇒ nat"
    and public :: "'fun ⇒ bool"
    and Ana :: "('fun,'var) term ⇒ (('fun,'var) term list × ('fun,'var) term list)"
  assumes Ana_keys_fv: "⋀t K R. Ana t = (K,R) ⟹ fvset (set K) ⊆ fv t"
    and Ana_keys_wf: "⋀t k K R f T.
      Ana t = (K,R) ⟹ (⋀g S. Fun g S ⊑ t ⟹ length S = arity g)
                    ⟹ k ∈ set K ⟹ Fun f T ⊑ k ⟹ length T = arity f"
    and Ana_var[simp]: "⋀x. Ana (Var x) = ([],[])"
    and Ana_fun_subterm: "⋀f T K R. Ana (Fun f T) = (K,R) ⟹ set R ⊆ set T"
    and Ana_subst: "⋀t δ K R. ⟦Ana t = (K,R); K ≠ [] ∨ R ≠ []⟧ ⟹ Ana (t · δ) = (K ·list δ,R ·list
δ)"
begin

lemma Ana_subterm: assumes "Ana t = (K,T)" shows "set T ⊂ subterms t"
using assms
```

```
by (cases t)
  (simp add: psubsetI,
   metis Ana_fun_subterm Fun_gt_params UN_I term.order_refl
         params_subterms psubsetI subset_antisym subset_trans)

lemma Ana_subterm': "s ∈ set (snd (Ana t)) ⟹ s ⊑ t"
using Ana_subterm by (cases "Ana t") auto

lemma Ana_vars: assumes "Ana t = (K,M)" shows "fv_set (set K) ⊆ fv t" "fv_set (set M) ⊆ fv t"
by (rule Ana_keys_fv[OF assms]) (use Ana_subterm[OF assms] subtermeq_vars_subset in auto)

abbreviation 𝒱 where "𝒱 ≡ UNIV::'var set"
abbreviation Σn ("Σ^") where "Σ^n ≡ {f::'fun. arity f = n}"
abbreviation Σnpub ("Σ_pub^") where "Σ_pub^n ≡ {f. public f} ∩ Σ^n"
abbreviation Σnpriv ("Σ_priv^") where "Σ_priv^n ≡ {f. ¬public f} ∩ Σ^n"
abbreviation Σ_pub where "Σ_pub ≡ (⋃n. Σ_pub^n)"
abbreviation Σ_priv where "Σ_priv ≡ (⋃n. Σ_priv^n)"
abbreviation Σ where "Σ ≡ (⋃n. Σ^n)"
abbreviation 𝒞 where "𝒞 ≡ Σ^0"
abbreviation 𝒞_pub where "𝒞_pub ≡ {f. public f} ∩ 𝒞"
abbreviation 𝒞_priv where "𝒞_priv ≡ {f. ¬public f} ∩ 𝒞"
abbreviation Σ_f where "Σ_f ≡ Σ - 𝒞"
abbreviation Σ_fpub where "Σ_fpub ≡ Σ_f ∩ Σ_pub"
abbreviation Σ_fpriv where "Σ_fpriv ≡ Σ_f ∩ Σ_priv"

lemma disjoint_fun_syms: "Σ_f ∩ 𝒞 = {}" by auto
lemma id_union_univ: "Σ_f ∪ 𝒞 = UNIV" "Σ = UNIV" by auto
lemma const_arity_eq_zero[dest]: "c ∈ 𝒞 ⟹ arity c = 0" by simp
lemma const_pub_arity_eq_zero[dest]: "c ∈ 𝒞_pub ⟹ arity c = 0 ∧ public c" by simp
lemma const_priv_arity_eq_zero[dest]: "c ∈ 𝒞_priv ⟹ arity c = 0 ∧ ¬public c" by simp
lemma fun_arity_gt_zero[dest]: "f ∈ Σ_f ⟹ arity f > 0" by fastforce
lemma pub_fun_public[dest]: "f ∈ Σ_fpub ⟹ public f" by fastforce
lemma pub_fun_arity_gt_zero[dest]: "f ∈ Σ_fpub ⟹ arity f > 0" by fastforce

lemma Σ_f_unfold: "Σ_f = {f::'fun. arity f > 0}" by auto
lemma 𝒞_unfold: "𝒞 = {f::'fun. arity f = 0}" by auto
lemma 𝒞pub_unfold: "𝒞_pub = {f::'fun. arity f = 0 ∧ public f}" by auto
lemma 𝒞priv_unfold: "𝒞_priv = {f::'fun. arity f = 0 ∧ ¬public f}" by auto
lemma Σnpub_unfold: "(Σ_pub^n) = {f::'fun. arity f = n ∧ public f}" by auto
lemma Σnpriv_unfold: "(Σ_priv^n) = {f::'fun. arity f = n ∧ ¬public f}" by auto
lemma Σfpub_unfold: "Σ_fpub = {f::'fun. arity f > 0 ∧ public f}" by auto
lemma Σfpriv_unfold: "Σ_fpriv = {f::'fun. arity f > 0 ∧ ¬public f}" by auto
lemma Σn_m_eq: "⟦(Σ^n) ≠ {}; (Σ^n) = (Σ^m)⟧ ⟹ n = m" by auto
```

2.4.3 Term Well-formedness

```
definition "wf_trm t ≡ ∀f T. Fun f T ⊑ t ⟶ length T = arity f"

abbreviation "wf_trms T ≡ ∀t ∈ T. wf_trm t"

lemma Ana_keys_wf': "Ana t = (K,T) ⟹ wf_trm t ⟹ k ∈ set K ⟹ wf_trm k"
using Ana_keys_wf unfolding wf_trm_def by metis

lemma wf_trm_Var[simp]: "wf_trm (Var x)" unfolding wf_trm_def by simp

lemma wf_trm_subst_range_Var[simp]: "wf_trms (subst_range Var)" by simp

lemma wf_trm_subst_range_iff: "(∀x. wf_trm (ϑ x)) ⟷ wf_trms (subst_range ϑ)"
by force

lemma wf_trm_subst_rangeD: "wf_trms (subst_range ϑ) ⟹ wf_trm (ϑ x)"
by (metis wf_trm_subst_range_iff)
```

```
lemma wf_trm_subst_rangeI[intro]:
  "(⋀x. wf_trm (δ x)) ⟹ wf_trms (subst_range δ)"
by (metis wf_trm_subst_range_iff)

lemma wf_trmI[intro]:
  assumes "⋀t. t ∈ set T ⟹ wf_trm t" "length T = arity f"
  shows "wf_trm (Fun f T)"
using assms unfolding wf_trm_def by auto

lemma wf_trm_subterm: "⟦wf_trm t; s ⊏ t⟧ ⟹ wf_trm s"
unfolding wf_trm_def by (induct t) auto

lemma wf_trm_subtermeq:
  assumes "wf_trm t" "s ⊑ t"
  shows "wf_trm s"
proof (cases "s = t")
  case False thus "wf_trm s" using assms(2) wf_trm_subterm[OF assms(1)] by simp
qed (metis assms(1))

lemma wf_trm_param:
  assumes "wf_trm (Fun f T)" "t ∈ set T"
  shows "wf_trm t"
by (meson assms subtermeqI'' wf_trm_subtermeq)

lemma wf_trm_param_idx:
  assumes "wf_trm (Fun f T)"
    and "i < length T"
  shows "wf_trm (T ! i)"
using wf_trm_param[OF assms(1), of "T ! i"] assms(2)
by fastforce

lemma wf_trm_subst:
  assumes "wf_trms (subst_range δ)"
  shows "wf_trm t = wf_trm (t · δ)"
proof
  show "wf_trm t ⟹ wf_trm (t · δ)"
  proof (induction t)
    case (Fun f T)
    hence "⋀t. t ∈ set T ⟹ wf_trm t"
      by (meson wf_trm_def Fun_param_is_subterm term.order_trans)
    hence "⋀t. t ∈ set T ⟹ wf_trm (t · δ)" using Fun.IH by auto
    moreover have "length (map (λt. t · δ) T) = arity f"
      using Fun.prems unfolding wf_trm_def by auto
    ultimately show ?case by fastforce
  qed (simp add: wf_trm_subst_rangeD[OF assms])

  show "wf_trm (t · δ) ⟹ wf_trm t"
  proof (induction t)
    case (Fun f T)
    hence "wf_trm t" when "t ∈ set (map (λs. s · δ) T)" for t
      by (metis that wf_trm_def Fun_param_is_subterm term.order_trans subst_apply_term.simps(2))
    hence "wf_trm t" when "t ∈ set T" for t using that Fun.IH by auto
    moreover have "length (map (λt. t · δ) T) = arity f"
      using Fun.prems unfolding wf_trm_def by auto
    ultimately show ?case by fastforce
  qed (simp add: assms)
qed

lemma wf_trm_subst_singleton:
  assumes "wf_trm t" "wf_trm t'" shows "wf_trm (t · Var(v := t'))"
proof -
  have "wf_trm ((Var(v := t')) w)" for w using assms(2) unfolding wf_trm_def by simp
  thus ?thesis using assms(1) wf_trm_subst[of "Var(v := t')" t, OF wf_trm_subst_rangeI] by simp
```

```
qed

lemma wf_trm_subst_rm_vars:
  assumes "wf_trm (t · δ)"
  shows "wf_trm (t · rm_vars X δ)"
using assms
proof (induction t)
  case (Fun f T)
  have "wf_trm (t · δ)" when "t ∈ set T" for t
    using that wf_trm_param[of f "map (λt. t · δ) T"] Fun.prems
    by auto
  hence "wf_trm (t · rm_vars X δ)" when "t ∈ set T" for t using that Fun.IH by simp
  moreover have "length T = arity f" using Fun.prems unfolding wf_trm_def by auto
  ultimately show ?case unfolding wf_trm_def by auto
qed simp

lemma wf_trm_subst_rm_vars': "wf_trm (δ v) ⟹ wf_trm (rm_vars X δ v)"
by auto

lemma wf_trms_subst:
  assumes "wf_trms (subst_range δ)" "wf_trms M"
  shows "wf_trms (M ·_set δ)"
by (metis (no_types, lifting) assms imageE wf_trm_subst)

lemma wf_trms_subst_rm_vars:
  assumes "wf_trms (M ·_set δ)"
  shows "wf_trms (M ·_set rm_vars X δ)"
using assms wf_trm_subst_rm_vars by blast

lemma wf_trms_subst_rm_vars':
  assumes "wf_trms (subst_range δ)"
  shows "wf_trms (subst_range (rm_vars X δ))"
using assms by force

lemma wf_trms_subst_compose:
  assumes "wf_trms (subst_range ϑ)" "wf_trms (subst_range δ)"
  shows "wf_trms (subst_range (ϑ ∘_s δ))"
using assms subst_img_comp_subset' wf_trm_subst by blast

lemma wf_trm_subst_compose:
  fixes δ::"('fun, 'v) subst"
  assumes "wf_trm (ϑ x)" "⋀x. wf_trm (δ x)"
  shows "wf_trm ((ϑ ∘_s δ) x)"
using wf_trm_subst[of δ "ϑ x", OF wf_trm_subst_rangeI[OF assms(2)]] assms(1)
      subst_subst_compose[of "Var x" ϑ δ]
      subst_apply_term.simps(1)[of x ϑ]
      subst_apply_term.simps(1)[of x "ϑ ∘_s δ"]
by argo

lemma wf_trms_Var_range:
  assumes "subst_range δ ⊆ range Var"
  shows "wf_trms (subst_range δ)"
using assms by fastforce

lemma wf_trms_subst_compose_Var_range:
  assumes "wf_trms (subst_range ϑ)"
    and "subst_range δ ⊆ range Var"
  shows "wf_trms (subst_range (δ ∘_s ϑ))"
    and "wf_trms (subst_range (ϑ ∘_s δ))"
using assms wf_trms_subst_compose wf_trms_Var_range by metis+

lemma wf_trm_subst_inv: "wf_trm (t · δ) ⟹ wf_trm t"
unfolding wf_trm_def by (induct t) auto
```

```
lemma wf_trms_subst_inv: "wf_trms (M ·set δ) ⟹ wf_trms M"
using wf_trm_subst_inv by fast

lemma wf_trm_subterms: "wf_trm t ⟹ wf_trms (subterms t)"
using wf_trm_subterm by blast

lemma wf_trms_subterms: "wf_trms M ⟹ wf_trms (subterms_set M)"
using wf_trm_subterms by blast

lemma wf_trm_arity: "wf_trm (Fun f T) ⟹ length T = arity f"
unfolding wf_trm_def by blast

lemma wf_trm_subterm_arity: "wf_trm t ⟹ Fun f T ⊑ t ⟹ length T = arity f"
unfolding wf_trm_def by blast

lemma unify_list_wf_trm:
  assumes "Unification.unify E B = Some U" "∀(s,t) ∈ set E. wf_trm s ∧ wf_trm t"
  and "∀(v,t) ∈ set B. wf_trm t"
  shows "∀(v,t) ∈ set U. wf_trm t"
using assms
proof (induction E B arbitrary: U rule: Unification.unify.induct)
  case (1 B U) thus ?case by auto
next
  case (2 f T g S E B U)
  have wf_fun: "wf_trm (Fun f T)" "wf_trm (Fun g S)" using "2.prems"(2) by auto
  from "2.prems"(1) obtain E' where *: "decompose (Fun f T) (Fun g S) = Some E'"
    and [simp]: "f = g" "length T = length S" "E' = zip T S"
    and **: "Unification.unify (E'@E) B = Some U"
    by (auto split: option.splits)
  hence "t ⊏ Fun f T" "t' ⊏ Fun g S" when "(t,t') ∈ set E'" for t t'
    using that by (metis zip_arg_subterm(1), metis zip_arg_subterm(2))
  hence "wf_trm t" "wf_trm t'" when "(t,t') ∈ set E'" for t t'
    using wf_trm_subterm wf_fun ‹f = g› that by blast+
  thus ?case using "2.IH"[OF * ** _ "2.prems"(3)] "2.prems"(2) by fastforce
next
  case (3 v t E B)
  hence *: "∀(w,x) ∈ set ((v, t) # B). wf_trm x"
      and **: "∀(s,t) ∈ set E. wf_trm s ∧ wf_trm t" "wf_trm t"
    by auto

  show ?case
  proof (cases "t = Var v")
    case True thus ?thesis using "3.prems" "3.IH"(1) by auto
  next
    case False
    hence "v ∉ fv t" using "3.prems"(1) by auto
    hence "Unification.unify (subst_list (subst v t) E) ((v, t)#B) = Some U"
      using ‹t ≠ Var v› "3.prems"(1) by auto
    moreover have "∀(s, t) ∈ set (subst_list (subst v t) E). wf_trm s ∧ wf_trm t"
      using wf_trm_subst_singleton[OF _ ‹wf_trm t›] "3.prems"(2)
      unfolding subst_list_def subst_def by auto
    ultimately show ?thesis using "3.IH"(2)[OF ‹t ≠ Var v› ‹v ∉ fv t› _ _ *] by metis
  qed
next
  case (4 f T v E B U)
  hence *: "∀(w,x) ∈ set ((v, Fun f T) # B). wf_trm x"
      and **: "∀(s,t) ∈ set E. wf_trm s ∧ wf_trm t" "wf_trm (Fun f T)"
    by auto

  have "v ∉ fv (Fun f T)" using "4.prems"(1) by force
  hence "Unification.unify (subst_list (subst v (Fun f T)) E) ((v, Fun f T)#B) = Some U"
    using "4.prems"(1) by auto
```

```
  moreover have "∀(s, t) ∈ set (subst_list (subst v (Fun f T)) E). wf_trm s ∧ wf_trm t"
    using wf_trm_subst_singleton[OF _ ‹wf_trm (Fun f T)›] "4.prems"(2)
    unfolding subst_list_def subst_def by auto
  ultimately show ?case using "4.IH"[OF ‹v ∉ fv (Fun f T)› _ _ *] by metis
qed

lemma mgu_wf_trm:
  assumes "mgu s t = Some σ" "wf_trm s" "wf_trm t"
  shows "wf_trm (σ v)"
proof -
  from assms obtain σ' where "subst_of σ' = σ" "∀(v,t) ∈ set σ'. wf_trm t"
    using unify_list_wf_trm[of "[(s,t)]" "[]"] by (auto split: option.splits)
  thus ?thesis
  proof (induction σ' arbitrary: σ v rule: List.rev_induct)
    case (snoc x σ' σ v)
    define ϑ where "ϑ = subst_of σ'"
    hence "wf_trm (ϑ v)" for v using snoc.prems(2) snoc.IH[of ϑ] by fastforce
    moreover obtain w t where x: "x = (w,t)" by (metis surj_pair)
    hence σ: "σ = Var(w := t) ∘s ϑ" using snoc.prems(1) by (simp add: subst_def ϑ_def)
    moreover have "wf_trm t" using snoc.prems(2) x by auto
    ultimately show ?case using wf_trm_subst[of _ t] unfolding subst_compose_def by auto
  qed (simp add: wf_trm_def)
qed

lemma mgu_wf_trms:
  assumes "mgu s t = Some σ" "wf_trm s" "wf_trm t"
  shows "wf_trms (subst_range σ)"
using mgu_wf_trm[OF assms] by simp
```

2.4.4 Definitions: Intruder Deduction Relations

A standard Dolev-Yao intruder.

```
inductive intruder_deduct::"('fun,'var) terms ⇒ ('fun,'var) term ⇒ bool"
where
  Axiom[simp]:    "t ∈ M ⟹ intruder_deduct M t"
| Compose[simp]: "⟦length T = arity f; public f; ⋀t. t ∈ set T ⟹ intruder_deduct M t⟧
                  ⟹ intruder_deduct M (Fun f T)"
| Decompose:     "⟦intruder_deduct M t; Ana t = (K, T); ⋀k. k ∈ set K ⟹ intruder_deduct M k;
                   t_i ∈ set T⟧
                  ⟹ intruder_deduct M t_i"
```

A variant of the intruder relation which limits the intruder to composition only.

```
inductive intruder_synth::"('fun,'var) terms ⇒ ('fun,'var) term ⇒ bool"
where
  AxiomC[simp]:    "t ∈ M ⟹ intruder_synth M t"
| ComposeC[simp]: "⟦length T = arity f; public f; ⋀t. t ∈ set T ⟹ intruder_synth M t⟧
                   ⟹ intruder_synth M (Fun f T)"

adhoc_overloading INTRUDER_DEDUCT intruder_deduct
adhoc_overloading INTRUDER_SYNTH intruder_synth

lemma intruder_deduct_induct[consumes 1, case_names Axiom Compose Decompose]:
  assumes "M ⊢ t" "⋀t. t ∈ M ⟹ P M t"
          "⋀T f. ⟦length T = arity f; public f;
                  ⋀t. t ∈ set T ⟹ M ⊢ t;
                  ⋀t. t ∈ set T ⟹ P M t⟧ ⟹ P M (Fun f T)"
          "⋀t K T t_i. ⟦M ⊢ t; P M t; Ana t = (K, T); ⋀k. k ∈ set K ⟹ M ⊢ k;
                        ⋀k. k ∈ set K ⟹ P M k; t_i ∈ set T⟧ ⟹ P M t_i"
  shows "P M t"
using assms by (induct rule: intruder_deduct.induct) blast+

lemma intruder_synth_induct[consumes 1, case_names AxiomC ComposeC]:
```

```
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "M ⊢c t" "⋀t. t ∈ M ⟹ P M t"
          "⋀T f. ⟦length T = arity f; public f;
                  ⋀t. t ∈ set T ⟹ M ⊢c t;
                  ⋀t. t ∈ set T ⟹ P M t⟧ ⟹ P M (Fun f T)"
  shows "P M t"
using assms by (induct rule: intruder_synth.induct) auto
```

2.4.5 Definitions: Analyzed Knowledge and Public Ground Well-formed Terms (PGWTs)

```
definition analyzed::"('fun,'var) terms ⇒ bool" where
  "analyzed M ≡ ∀t. M ⊢ t ⟷ M ⊢c t"

definition analyzed_in where
  "analyzed_in t M ≡ ∀K R. (Ana t = (K,R) ∧ (∀k ∈ set K. M ⊢c k)) ⟶ (∀r ∈ set R. M ⊢c r)"

definition decomp_closure::"('fun,'var) terms ⇒ ('fun,'var) terms ⇒ bool" where
  "decomp_closure M M' ≡ ∀t. M ⊢ t ∧ (∃t' ∈ M. t ⊑ t') ⟷ t ∈ M'"

inductive public_ground_wf_term::"('fun,'var) term ⇒ bool" where
  PGWT[simp]: "⟦public f; arity f = length T;
              ⋀t. t ∈ set T ⟹ public_ground_wf_term t⟧
                ⟹ public_ground_wf_term (Fun f T)"

abbreviation "public_ground_wf_terms ≡ {t. public_ground_wf_term t}"

lemma public_const_deduct:
  assumes "c ∈ C_pub"
  shows "M ⊢ Fun c []" "M ⊢c Fun c []"
proof -
  have "arity c = 0" "public c" using const_arity_eq_zero ‹c ∈ C_pub› by auto
  thus "M ⊢ Fun c []" "M ⊢c Fun c []"
    using intruder_synth.ComposeC[OF _ ‹public c›, of "[]"]
          intruder_deduct.Compose[OF _ ‹public c›, of "[]"]
    by auto
qed

lemma public_const_deduct'[simp]:
  assumes "arity c = 0" "public c"
  shows "M ⊢ Fun c []" "M ⊢c Fun c []"
using intruder_deduct.Compose[of "[]" c] intruder_synth.ComposeC[of "[]" c] assms by simp_all

lemma private_fun_deduct_in_ik:
  assumes t: "M ⊢ t" "Fun f T ∈ subterms t"
    and f: "¬public f"
  shows "Fun f T ∈ subterms_set M"
using t
proof (induction t rule: intruder_deduct.induct)
  case Decompose thus ?case by (meson Ana_subterm psubsetD term.order_trans)
qed (auto simp add: f in_subterms_Union)

lemma private_fun_deduct_in_ik':
  assumes t: "M ⊢ Fun f T"
    and f: "¬public f"
    and M: "Fun f T ∈ subterms_set M ⟹ Fun f T ∈ M"
  shows "Fun f T ∈ M"
by (rule M[OF private_fun_deduct_in_ik[OF t term.order_refl f]])

lemma pgwt_public: "⟦public_ground_wf_term t; Fun f T ⊑ t⟧ ⟹ public f"
by (induct t rule: public_ground_wf_term.induct) auto

lemma pgwt_ground: "public_ground_wf_term t ⟹ fv t = {}"
by (induct t rule: public_ground_wf_term.induct) auto
```

```
lemma pgwt_fun: "public_ground_wf_term t ⟹ ∃f T. t = Fun f T"
using pgwt_ground[of t] by (cases t) auto

lemma pgwt_arity: "⟦public_ground_wf_term t; Fun f T ⊑ t⟧ ⟹ arity f = length T"
by (induct t rule: public_ground_wf_term.induct) auto

lemma pgwt_wellformed: "public_ground_wf_term t ⟹ wf_trm t"
by (induct t rule: public_ground_wf_term.induct) auto

lemma pgwt_deducible: "public_ground_wf_term t ⟹ M ⊢c t"
by (induct t rule: public_ground_wf_term.induct) auto

lemma pgwt_is_empty_synth: "public_ground_wf_term t ⟷ {} ⊢c t"
proof -
  { fix M::"('fun,'var) term set" assume "M ⊢c t" "M = {}" hence "public_ground_wf_term t"
      by (induct t rule: intruder_synth.induct) auto
  }
  thus ?thesis using pgwt_deducible by auto
qed

lemma ideduct_synth_subst_apply:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "{} ⊢c t" "⋀v. M ⊢c ϑ v"
  shows "M ⊢c t · ϑ"
proof -
  { fix M'::"('fun,'var) term set" assume "M' ⊢c t" "M' = {}" hence "M ⊢c t · ϑ"
    proof (induction t rule: intruder_synth.induct)
      case (ComposeC T f M')
      hence "length (map (λt. t · ϑ) T) = arity f" "⋀x. x ∈ set (map (λt. t · ϑ) T) ⟹ M ⊢c x"
        by auto
      thus ?case using intruder_synth.ComposeC[of "map (λt. t · ϑ) T" f M] ‹public f› by fastforce
    qed simp
  }
  thus ?thesis using assms by metis
qed
```

2.4.6 Lemmata: Monotonicity, deduction private constants, etc.

```
context
begin
lemma ideduct_mono:
  "⟦M ⊢ t; M ⊆ M'⟧ ⟹ M' ⊢ t"
proof (induction rule: intruder_deduct.induct)
  case (Decompose M t K T ti)
  have "∀k. k ∈ set K ⟶ M' ⊢ k" using Decompose.IH ‹M ⊆ M'› by simp
  moreover have "M' ⊢ t" using Decompose.IH ‹M ⊆ M'› by simp
  ultimately show ?case using Decompose.hyps intruder_deduct.Decompose by blast
qed auto

lemma ideduct_synth_mono:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  shows "⟦M ⊢c t; M ⊆ M'⟧ ⟹ M' ⊢c t"
by (induct rule: intruder_synth.induct) auto

lemma ideduct_reduce:
  "⟦M ∪ M' ⊢ t; ⋀t'. t' ∈ M' ⟹ M ⊢ t'⟧ ⟹ M ⊢ t"
proof (induction rule: intruder_deduct_induct)
  case Decompose thus ?case using intruder_deduct.Decompose by blast
qed auto

lemma ideduct_synth_reduce:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
```

```
  shows "⟦M ∪ M' ⊢c t; ⋀t'. t' ∈ M' ⟹ M ⊢c t'⟧ ⟹ M ⊢c t"
by (induct rule: intruder_synth_induct) auto

lemma ideduct_mono_eq:
  assumes "∀t. M ⊢ t ⟷ M' ⊢ t" shows "M ∪ N ⊢ t ⟷ M' ∪ N ⊢ t"
proof
  show "M ∪ N ⊢ t ⟹ M' ∪ N ⊢ t"
  proof (induction t rule: intruder_deduct_induct)
    case (Axiom t) thus ?case
    proof (cases "t ∈ M")
      case True
      hence "M ⊢ t" using intruder_deduct.Axiom by metis
      thus ?thesis using assms ideduct_mono[of M' t "M' ∪ N"] by simp
    qed auto
  next
    case (Compose T f) thus ?case using intruder_deduct.Compose by auto
  next
    case (Decompose t K T ti) thus ?case using intruder_deduct.Decompose[of "M' ∪ N" t K T] by auto
  qed

  show "M' ∪ N ⊢ t ⟹ M ∪ N ⊢ t"
  proof (induction t rule: intruder_deduct_induct)
    case (Axiom t) thus ?case
    proof (cases "t ∈ M'")
      case True
      hence "M' ⊢ t" using intruder_deduct.Axiom by metis
      thus ?thesis using assms ideduct_mono[of M t "M ∪ N"] by simp
    qed auto
  next
    case (Compose T f) thus ?case using intruder_deduct.Compose by auto
  next
    case (Decompose t K T ti) thus ?case using intruder_deduct.Decompose[of "M ∪ N" t K T] by auto
  qed
qed

lemma deduct_synth_subterm:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "M ⊢c t" "s ∈ subterms t" "∀m ∈ M. ∀s ∈ subterms m. M ⊢c s"
  shows "M ⊢c s"
using assms by (induct t rule: intruder_synth.induct) auto

lemma deduct_if_synth[intro, dest]: "M ⊢c t ⟹ M ⊢ t"
by (induct rule: intruder_synth.induct) auto

private lemma ideduct_ik_eq: assumes "∀t ∈ M. M' ⊢ t" shows "M' ⊢ t ⟷ M' ∪ M ⊢ t"
by (meson assms ideduct_mono ideduct_reduce sup_ge1)

private lemma synth_if_deduct_empty: "{} ⊢ t ⟹ {} ⊢c t"
proof (induction t rule: intruder_deduct_induct)
  case (Decompose t K M m)
  then obtain f T where "t = Fun f T" "m ∈ set T"
    using Ana_fun_subterm Ana_var by (cases t) fastforce+
  with Decompose.IH(1) show ?case by (induction rule: intruder_synth_induct) auto
qed auto

private lemma ideduct_deduct_synth_mono_eq:
  assumes "∀t. M ⊢ t ⟷ M' ⊢c t" "M ⊆ M'"
  and "∀t. M' ∪ N ⊢ t ⟷ M' ∪ N ∪ D ⊢c t"
  shows "M ∪ N ⊢ t ⟷ M' ∪ N ∪ D ⊢c t"
proof -
  have "∀m ∈ M'. M ⊢ m" using assms(1) by auto
  hence "∀t. M ⊢ t ⟷ M' ⊢ t" by (metis assms(1,2) deduct_if_synth ideduct_reduce sup.absorb2)
  hence "∀t. M' ∪ N ⊢ t ⟷ M ∪ N ⊢ t" by (meson ideduct_mono_eq)
```

```
  thus ?thesis by (meson assms(3))
qed

lemma ideduct_subst: "M ⊢ t ⟹ M ·set δ ⊢ t · δ"
proof (induction t rule: intruder_deduct_induct)
  case (Compose T f)
  hence "length (map (λt. t · δ) T) = arity f" "⋀t. t ∈ set T ⟹ M ·set δ ⊢ t · δ" by auto
  thus ?case using intruder_deduct.Compose[OF _ Compose.hyps(2), of "map (λt. t · δ) T"] by auto
next
  case (Decompose t K M' m')
  hence "Ana (t · δ) = (K ·list δ, M' ·list δ)"
        "⋀k. k ∈ set (K ·list δ) ⟹ M ·set δ ⊢ k"
        "m' · δ ∈ set (M' ·list δ)"
    using Ana_subst[OF Decompose.hyps(2)] by fastforce+
  thus ?case using intruder_deduct.Decompose[OF Decompose.IH(1)] by metis
qed simp

lemma ideduct_synth_subst:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term" and δ::"('fun,'var) subst"
  shows "M ⊢c t ⟹ M ·set δ ⊢c t · δ"
proof (induction t rule: intruder_synth_induct)
  case (ComposeC T f)
  hence "length (map (λt. t · δ) T) = arity f" "⋀t. t ∈ set T ⟹ M ·set δ ⊢c t · δ" by auto
  thus ?case using intruder_synth.ComposeC[OF _ ComposeC.hyps(2), of "map (λt. t · δ) T"] by auto
qed simp

lemma ideduct_vars:
  assumes "M ⊢ t"
  shows "fv t ⊆ fvset M"
using assms
proof (induction t rule: intruder_deduct_induct)
  case (Decompose t K T ti) thus ?case
    using Ana_vars(2) fv_subset by blast
qed auto

lemma ideduct_synth_vars:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "M ⊢c t"
  shows "fv t ⊆ fvset M"
using assms by (induct t rule: intruder_synth_induct) auto

lemma ideduct_synth_priv_fun_in_ik:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "M ⊢c t" "f ∈ funs_term t" "¬public f"
  shows "f ∈ ⋃(funs_term ` M)"
using assms by (induct t rule: intruder_synth_induct) auto

lemma ideduct_synth_priv_const_in_ik:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "M ⊢c Fun c []" "¬public c"
  shows "Fun c [] ∈ M"
using intruder_synth.cases[OF assms(1)] assms(2) by fast

lemma ideduct_synth_ik_replace:
  fixes M::"('fun,'var) terms" and t::"('fun,'var) term"
  assumes "∀t ∈ M. N ⊢c t"
    and "M ⊢c t"
  shows "N ⊢c t"
using assms(2,1) by (induct t rule: intruder_synth.induct) auto
end
```

2.4.7 Lemmata: Analyzed Intruder Knowledge Closure

```
lemma deducts_eq_if_analyzed: "analyzed M ⟹ M ⊢ t ⟷ M ⊢c t"
unfolding analyzed_def by auto

lemma closure_is_superset: "decomp_closure M M' ⟹ M ⊆ M'"
unfolding decomp_closure_def by force

lemma deduct_if_closure_deduct: "⟦M' ⊢ t; decomp_closure M M'⟧ ⟹ M ⊢ t"
proof (induction t rule: intruder_deduct.induct)
  case (Decompose M' t K T ti)
  thus ?case using intruder_deduct.Decompose[OF _ ‹Ana t = (K,T)› _ ‹ti ∈ set T›] by simp
qed (auto simp add: decomp_closure_def)

lemma deduct_if_closure_synth: "⟦decomp_closure M M'; M' ⊢c t⟧ ⟹ M ⊢ t"
using deduct_if_closure_deduct by blast

lemma decomp_closure_subterms_composable:
  assumes "decomp_closure M M'"
  and "M' ⊢c t'" "M' ⊢ t" "t ⊑ t'"
  shows "M' ⊢c t"
using ‹M' ⊢c t'› assms
proof (induction t' rule: intruder_synth.induct)
  case (AxiomC t' M')
  have "M ⊢ t" using ‹M' ⊢ t› deduct_if_closure_deduct AxiomC.prems(1) by blast
  moreover
  { have "∃s ∈ M. t' ⊑ s" using ‹t' ∈ M'› AxiomC.prems(1) unfolding decomp_closure_def by blast
    hence "∃s ∈ M. t ⊑ s" using ‹t ⊑ t'› term.order_trans by auto
  }
  ultimately have "t ∈ M'" using AxiomC.prems(1) unfolding decomp_closure_def by blast
  thus ?case by simp
next
  case (ComposeC T f M')
  let ?t' = "Fun f T"
  { assume "t = ?t'" have "M' ⊢c t" using ‹M' ⊢c ?t'› ‹t = ?t'› by simp }
  moreover
  { assume "t ≠ ?t'"
    have "∃x ∈ set T. t ⊑ x" using ‹t ⊑ ?t'› ‹t ≠ ?t'› by simp
    hence "M' ⊢c t" using ComposeC.IH ComposeC.prems(1,3) ComposeC.hyps(3) by blast
  }
  ultimately show ?case using cases_simp[of "t = ?t'" "M' ⊢c t"] by simp
qed

lemma decomp_closure_analyzed:
  assumes "decomp_closure M M'"
  shows "analyzed M'"
proof -
  { fix t assume "M' ⊢ t" have "M' ⊢c t" using ‹M' ⊢ t› assms
    proof (induction t rule: intruder_deduct.induct)
      case (Decompose M' t K T ti)
      hence "M' ⊢ ti" using Decompose.hyps intruder_deduct.Decompose by blast
      moreover have "ti ⊑ t"
        using Decompose.hyps(4) Ana_subterm[OF Decompose.hyps(2)] by blast
      moreover have "M' ⊢c t" using Decompose.IH(1) Decompose.prems by blast
      ultimately show "M' ⊢c ti" using decomp_closure_subterms_composable Decompose.prems by blast
    qed auto
  }
  moreover have "∀t. M ⊢c t ⟶ M ⊢ t" by auto
  ultimately show ?thesis by (auto simp add: decomp_closure_def analyzed_def)
qed

lemma analyzed_if_all_analyzed_in:
  assumes M: "∀t ∈ M. analyzed_in t M"
```

```
  shows "analyzed M"
proof (unfold analyzed_def, intro allI iffI)
  fix t
  assume t: "M ⊢ t"
  thus "M ⊢c t"
  proof (induction t rule: intruder_deduct_induct)
    case (Decompose t K T ti)
    { assume "t ∈ M"
      hence ?case
        using M Decompose.IH(2) Decompose.hyps(2,4)
        unfolding analyzed_in_def by fastforce
    } moreover {
      fix f S assume "t = Fun f S" "⋀s. s ∈ set S ⟹ M ⊢c s"
      hence ?case using Ana_fun_subterm[of f S] Decompose.hyps(2,4) by blast
    } ultimately show ?case using intruder_synth.cases[OF Decompose.IH(1), of ?case] by blast
  qed simp_all
qed auto

lemma analyzed_is_all_analyzed_in:
  "(∀t ∈ M. analyzed_in t M) ⟷ analyzed M"
proof
  show "analyzed M ⟹ ∀t ∈ M. analyzed_in t M"
    unfolding analyzed_in_def analyzed_def
    by (auto intro: intruder_deduct.Decompose[OF intruder_deduct.Axiom])
qed (rule analyzed_if_all_analyzed_in)

lemma ik_has_synth_ik_closure:
  fixes M :: "('fun,'var) terms"
  shows "∃M'. (∀t. M ⊢ t ⟷ M' ⊢c t) ∧ decomp_closure M M' ∧ (finite M ⟶ finite M')"
proof -
  let ?M' = "{t. M ⊢ t ∧ (∃t' ∈ M. t ⊑ t')}"

  have M'_closes: "decomp_closure M ?M'" unfolding decomp_closure_def by simp
  hence "M ⊆ ?M'" using closure_is_superset by simp

  have "∀t. ?M' ⊢c t ⟶ M ⊢ t" using deduct_if_closure_synth[OF M'_closes] by blast
  moreover have "∀t. M ⊢ t ⟶ ?M' ⊢ t" using ideduct_mono[OF _ ‹M ⊆ ?M'›] by simp
  moreover have "analyzed ?M'" using decomp_closure_analyzed[OF M'_closes] .
  ultimately have "∀t. M ⊢ t ⟷ ?M' ⊢c t" unfolding analyzed_def by blast
  moreover have "finite M ⟶ finite ?M'" by auto
  ultimately show ?thesis using M'_closes by blast
qed
```

2.4.8 Intruder Variants: Numbered and Composition-Restricted Intruder Deduction Relations

A variant of the intruder relation which restricts composition to only those terms that satisfy a given predicate Q.

```
inductive intruder_deduct_restricted::
  "('fun,'var) terms ⇒ (('fun,'var) term ⇒ bool) ⇒ ('fun,'var) term ⇒ bool"
  ("⟨_;_⟩ ⊢r _" 50)
where
  AxiomR[simp]:    "t ∈ M ⟹ ⟨M; Q⟩ ⊢r t"
| ComposeR[simp]: "⟦length T = arity f; public f; ⋀t. t ∈ set T ⟹ ⟨M; Q⟩ ⊢r t; Q (Fun f T)⟧
                    ⟹ ⟨M; Q⟩ ⊢r Fun f T"
| DecomposeR:      "⟦⟨M; Q⟩ ⊢r t; Ana t = (K, T); ⋀k. k ∈ set K ⟹ ⟨M; Q⟩ ⊢r k; ti ∈ set T⟧
                    ⟹ ⟨M; Q⟩ ⊢r ti"
```

A variant of the intruder relation equipped with a number representing the heigth of the derivation tree (i.e., $\langle M;\ k\rangle \vdash_n t$ iff k is the maximum number of applications of the compose an decompose rules in any path of the derivation tree for $M \vdash t$).

```
inductive intruder_deduct_num::
```

```
  "('fun,'var) terms ⇒ nat ⇒ ('fun,'var) term ⇒ bool"
  ("⟨_; _⟩ ⊢n _" 50)
where
  AxiomN[simp]:   "t ∈ M ⟹ ⟨M; 0⟩ ⊢n t"
| ComposeN[simp]: "⟦length T = arity f; public f; ⋀t. t ∈ set T ⟹ ⟨M; steps t⟩ ⊢n t⟧
                     ⟹ ⟨M; Suc (Max (insert 0 (steps ` set T)))⟩ ⊢n Fun f T"
| DecomposeN:     "⟦⟨M; n⟩ ⊢n t; Ana t = (K, T); ⋀k. k ∈ set K ⟹ ⟨M; steps k⟩ ⊢n k; ti ∈ set T⟧
                     ⟹ ⟨M; Suc (Max (insert n (steps ` set K)))⟩ ⊢n ti"

lemma intruder_deduct_restricted_induct[consumes 1, case_names AxiomR ComposeR DecomposeR]:
  assumes "⟨M; Q⟩ ⊢r t" "⋀t. t ∈ M ⟹ P M Q t"
          "⋀T f. ⟦length T = arity f; public f;
                  ⋀t. t ∈ set T ⟹ ⟨M; Q⟩ ⊢r t;
                  ⋀t. t ∈ set T ⟹ P M Q t; Q (Fun f T)
                  ⟧ ⟹ P M Q (Fun f T)"
          "⋀t K T ti. ⟦⟨M; Q⟩ ⊢r t; P M Q t; Ana t = (K, T); ⋀k. k ∈ set K ⟹ ⟨M; Q⟩ ⊢r k;
                      ⋀k. k ∈ set K ⟹ P M Q k; ti ∈ set T⟧ ⟹ P M Q ti"
  shows "P M Q t"
using assms by (induct t rule: intruder_deduct_restricted.induct) blast+

lemma intruder_deduct_num_induct[consumes 1, case_names AxiomN ComposeN DecomposeN]:
  assumes "⟨M; n⟩ ⊢n t" "⋀t. t ∈ M ⟹ P M 0 t"
          "⋀T f steps.
              ⟦length T = arity f; public f;
               ⋀t. t ∈ set T ⟹ ⟨M; steps t⟩ ⊢n t;
               ⋀t. t ∈ set T ⟹ P M (steps t) t⟧
              ⟹ P M (Suc (Max (insert 0 (steps ` set T)))) (Fun f T)"
          "⋀t K T ti steps n.
              ⟦⟨M; n⟩ ⊢n t; P M n t; Ana t = (K, T);
               ⋀k. k ∈ set K ⟹ ⟨M; steps k⟩ ⊢n k;
               ti ∈ set T; ⋀k. k ∈ set K ⟹ P M (steps k) k⟧
              ⟹ P M (Suc (Max (insert n (steps ` set K)))) ti"
  shows "P M n t"
using assms by (induct rule: intruder_deduct_num.induct) blast+

lemma ideduct_restricted_mono:
  "⟦⟨M; P⟩ ⊢r t; M ⊆ M'⟧ ⟹ ⟨M'; P⟩ ⊢r t"
proof (induction rule: intruder_deduct_restricted_induct)
  case (DecomposeR t K T ti)
  have "∀k. k ∈ set K ⟶ ⟨M'; P⟩ ⊢r k" using DecomposeR.IH ‹M ⊆ M'› by simp
  moreover have "⟨M'; P⟩ ⊢r t" using DecomposeR.IH ‹M ⊆ M'› by simp
  ultimately show ?case
    using DecomposeR
          intruder_deduct_restricted.DecomposeR[OF _ DecomposeR.hyps(2) _ DecomposeR.hyps(4)]
    by blast
qed auto
```

2.4.9 Lemmata: Intruder Deduction Equivalences

```
lemma deduct_if_restricted_deduct: "⟨M;P⟩ ⊢r m ⟹ M ⊢ m"
proof (induction m rule: intruder_deduct_restricted_induct)
  case (DecomposeR t K T ti) thus ?case using intruder_deduct.Decompose by blast
qed simp_all

lemma restricted_deduct_if_restricted_ik:
  assumes "⟨M;P⟩ ⊢r m" "∀m ∈ M. P m"
  and P: "∀t t'. P t ⟶ t' ⊑ t ⟶ P t'"
  shows "P m"
using assms(1)
proof (induction m rule: intruder_deduct_restricted_induct)
  case (DecomposeR t K T ti)
  obtain f S where "t = Fun f S" using Ana_var ‹ti ∈ set T› ‹Ana t = (K, T)› by (cases t) auto
  thus ?case using DecomposeR assms(2) P Ana_subterm by blast
```

```
qed (simp_all add: assms(2))

lemma deduct_restricted_if_synth:
  assumes P: "P m" "∀t t'. P t ⟶ t' ⊑ t ⟶ P t'"
  and m: "M ⊢c m"
  shows "⟨M; P⟩ ⊢r m"
using m P(1)
proof (induction m rule: intruder_synth_induct)
  case (ComposeC T f)
  hence "⟨M; P⟩ ⊢r t" when t: "t ∈ set T" for t
    using t P(2) subtermeqI''[of _ T f]
    by fastforce
  thus ?case
    using intruder_deduct_restricted.ComposeR[OF ComposeC.hyps(1,2)] ComposeC.prems(1)
    by metis
qed simp

lemma deduct_zero_in_ik:
  assumes "⟨M; 0⟩ ⊢n t" shows "t ∈ M"
proof -
  { fix k assume "⟨M; k⟩ ⊢n t" hence "k > 0 ∨ t ∈ M" by (induct t) auto
  } thus ?thesis using assms by auto
qed

lemma deduct_if_deduct_num: "⟨M; k⟩ ⊢n t ⟹ M ⊢ t"
by (induct t rule: intruder_deduct_num.induct)
   (metis intruder_deduct.Axiom,
    metis intruder_deduct.Compose,
    metis intruder_deduct.Decompose)

lemma deduct_num_if_deduct: "M ⊢ t ⟹ ∃k. ⟨M; k⟩ ⊢n t"
proof (induction t rule: intruder_deduct_induct)
  case (Compose T f)
  then obtain steps where *: "∀t ∈ set T. ⟨M; steps t⟩ ⊢n t" by moura
  then obtain n where "∀t ∈ set T. steps t ≤ n"
    using finite_nat_set_iff_bounded_le[of "steps ` set T"]
    by auto
  thus ?case using ComposeN[OF Compose.hyps(1,2), of M steps] * by force
next
  case (Decompose t K T ti)
  hence "⋀u. u ∈ insert t (set K) ⟹ ∃k. ⟨M; k⟩ ⊢n u" by auto
  then obtain steps where *: "⟨M; steps t⟩ ⊢n t" "∀t ∈ set K. ⟨M; steps t⟩ ⊢n t" by moura
  then obtain n where "steps t ≤ n" "∀t ∈ set K. steps t ≤ n"
    using finite_nat_set_iff_bounded_le[of "steps ` insert t (set K)"]
    by auto
  thus ?case using DecomposeN[OF _ Decompose.hyps(2) _ Decompose.hyps(4), of M _ steps] * by force
qed (metis AxiomN)

lemma deduct_normalize:
  assumes M: "∀m ∈ M. ∀f T. Fun f T ⊑ m ⟶ P f T"
  and t: "⟨M; k⟩ ⊢n t" "Fun f T ⊑ t" "¬P f T"
  shows "∃l ≤ k. (⟨M; l⟩ ⊢n Fun f T) ∧ (∀t ∈ set T. ∃j < l. ⟨M; j⟩ ⊢n t)"
using t
proof (induction t rule: intruder_deduct_num_induct)
  case (AxiomN t) thus ?case using M by auto
next
  case (ComposeN T' f' steps) thus ?case
  proof (cases "Fun f' T' = Fun f T")
    case True
    hence "⟨M; Suc (Max (insert 0 (steps ` set T')))⟩ ⊢n Fun f T" "T = T'"
      using intruder_deduct_num.ComposeN[OF ComposeN.hyps] by auto
    moreover have "⋀t. t ∈ set T ⟹ ⟨M; steps t⟩ ⊢n t"
      using True ComposeN.hyps(3) by auto
```

```
    moreover have "⋀t. t ∈ set T ⟹ steps t < Suc (Max (insert 0 (steps ` set T)))"
      using Max_less_iff[of "insert 0 (steps ` set T)" "Suc (Max (insert 0 (steps ` set T)))"]
      by auto
    ultimately show ?thesis by auto
  next
    case False
    then obtain t' where t': "t' ∈ set T'" "Fun f T ⊑ t'" using ComposeN by auto
    hence "∃l ≤ steps t'. (⟨M; l⟩ ⊢n Fun f T) ∧ (∀t ∈ set T. ∃j < l. ⟨M; j⟩ ⊢n t)"
      using ComposeN.IH[OF _ _ ComposeN.prems(2)] by auto
    moreover have "steps t' < Suc (Max (insert 0 (steps ` set T')))"
      using Max_less_iff[of "insert 0 (steps ` set T')" "Suc (Max (insert 0 (steps ` set T')))"]
      using t'(1) by auto
    ultimately show ?thesis using ComposeN.hyps(3)[OF t'(1)]
      by (meson Suc_le_eq le_Suc_eq le_trans)
  qed
next
  case (DecomposeN t K T' ti steps n)
  hence *: "Fun f T ⊑ t"
    using term.order_trans[of "Fun f T" ti t] Ana_subterm[of t K T']
    by blast
  have "∃l ≤ n. (⟨M; l⟩ ⊢n Fun f T) ∧ (∀t' ∈ set T. ∃j < l. ⟨M; j⟩ ⊢n t')"
    using DecomposeN.IH(1)[OF * DecomposeN.prems(2)] by auto
  moreover have "n < Suc (Max (insert n (steps ` set K)))"
      using Max_less_iff[of "insert n (steps ` set K)" "Suc (Max (insert n (steps ` set K)))"]
      by auto
  ultimately show ?case using DecomposeN.hyps(4) by (meson Suc_le_eq le_Suc_eq le_trans)
qed


lemma deduct_inv:
  assumes "⟨M; n⟩ ⊢n t"
  shows "t ∈ M ∨
         (∃f T. t = Fun f T ∧ public f ∧ length T = arity f ∧ (∀t ∈ set T. ∃l < n. ⟨M; l⟩ ⊢n t))
∨
         (∃m ∈ subterms_set M.
            (∃l < n. ⟨M; l⟩ ⊢n m) ∧ (∀k ∈ set (fst (Ana m)). ∃l < n. ⟨M; l⟩ ⊢n k) ∧
            t ∈ set (snd (Ana m)))"
    (is "?P t n ∨ ?Q t n ∨ ?R t n")
using assms
proof (induction n arbitrary: t rule: nat_less_induct)
  case (1 n t) thus ?case
  proof (cases n)
    case 0
    hence "t ∈ M" using deduct_zero_in_ik "1.prems"(1) by metis
    thus ?thesis by auto
  next
    case (Suc n')
    hence "⟨M; Suc n'⟩ ⊢n t"
          "∀m < Suc n'. ∀x. (⟨M; m⟩ ⊢n x) ⟶ ?P x m ∨ ?Q x m ∨ ?R x m"
      using "1.prems" "1.IH" by blast+
    hence "?P t (Suc n') ∨ ?Q t (Suc n') ∨ ?R t (Suc n')"
    proof (induction t rule: intruder_deduct_num_induct)
      case (AxiomN t) thus ?case by simp
    next
      case (ComposeN T f steps)
      have "⋀t. t ∈ set T ⟹ steps t < Suc (Max (insert 0 (steps ` set T)))"
          using Max_less_iff[of "insert 0 (steps ` set T)" "Suc (Max (insert 0 (steps ` set T)))"]
          by auto
      thus ?case using ComposeN.hyps by metis
    next
      case (DecomposeN t K T ti steps n)
      have 0: "n < Suc (Max (insert n (steps ` set K)))"
              "⋀k. k ∈ set K ⟹ steps k < Suc (Max (insert n (steps ` set K)))"
        using Max_less_iff[of "insert n (steps ` set K)" "Suc (Max (insert n (steps ` set K)))"]
```

```
        by auto

      have IH1: "?P t j ∨ ?Q t j ∨ ?R t j" when jt: "j < n" "⟨M; j⟩ ⊢n t" for j t
        using jt DecomposeN.prems(1) 0(1)
        by simp

      have IH2: "?P t n ∨ ?Q t n ∨ ?R t n"
        using DecomposeN.IH(1) IH1
        by simp

      have 1: "∀k ∈ set (fst (Ana t)). ∃l < Suc (Max (insert n (steps ` set K))). ⟨M; l⟩ ⊢n k"
        using DecomposeN.hyps(1,2,3) 0(2)
        by auto

      have 2: "ti ∈ set (snd (Ana t))"
        using DecomposeN.hyps(2,4)
        by fastforce

      have 3: "t ∈ subtermsset M" when "t ∈ set (snd (Ana m))" "m ⊑set M" for m
        using that(1) Ana_subterm[of m _ "snd (Ana m)"] in_subterms_subset_Union[OF that(2)]
        by (metis (no_types, lifting) prod.collapse psubsetD subsetCE subsetD)

      have 4: "?R ti (Suc (Max (insert n (steps ` set K))))" when "?R t n"
        using that 0(1) 1 2 3 DecomposeN.hyps(1)
        by (metis (no_types, lifting))

      have 5: "?R ti (Suc (Max (insert n (steps ` set K))))" when "?P t n"
        using that 0(1) 1 2 DecomposeN.hyps(1)
        by blast

      have 6: ?case when *: "?Q t n"
      proof -
        obtain g S where g:
            "t = Fun g S" "public g" "length S = arity g" "∀t ∈ set S. ∃l < n. ⟨M; l⟩ ⊢n t"
          using * by moura
        then obtain l where l: "l < n" "⟨M; l⟩ ⊢n ti"
          using 0(1) DecomposeN.hyps(2,4) Ana_fun_subterm[of g S K T] by blast

        have **: "l < Suc (Max (insert n (steps ` set K)))" using l(1) 0(1) by simp

        show ?thesis using IH1[OF l] less_trans[OF _ **] by fastforce
      qed

      show ?case using IH2 4 5 6 by argo
    qed
    thus ?thesis using Suc by fast
  qed
qed

lemma restricted_deduct_if_deduct:
  assumes M: "∀m ∈ M. ∀f T. Fun f T ⊑ m ⟶ P (Fun f T)"
  and P_subterm: "∀f T t. M ⊢ Fun f T ⟶ P (Fun f T) ⟶ t ∈ set T ⟶ P t"
  and P_Ana_key: "∀t K T k. M ⊢ t ⟶ P t ⟶ Ana t = (K, T) ⟶ M ⊢ k ⟶ k ∈ set K ⟶ P k"
  and m: "M ⊢ m" "P m"
  shows "⟨M; P⟩ ⊢r m"
proof -
  { fix k assume "⟨M; k⟩ ⊢n m"
    hence ?thesis using m(2)
    proof (induction k arbitrary: m rule: nat_less_induct)
      case (1 n m) thus ?case
      proof (cases n)
        case 0
        hence "m ∈ M" using deduct_zero_in_ik "1.prems"(1) by metis
```

```
      thus ?thesis by auto
    next
      case (Suc n')
      hence "⟨M; Suc n'⟩ ⊢n m"
            "∀m < Suc n'. ∀x. (⟨M; m⟩ ⊢n x) ⟶ P x ⟶ ⟨M;P⟩ ⊢r x"
        using "1.prems" "1.IH" by blast+
      thus ?thesis using "1.prems"(2)
      proof (induction m rule: intruder_deduct_num_induct)
        case (ComposeN T f steps)
        have *: "steps t < Suc (Max (insert 0 (steps ` set T)))" when "t ∈ set T" for t
          using Max_less_iff[of "insert 0 (steps ` set T)"] that
          by blast

        have **: "P t" when "t ∈ set T" for t
          using P_subterm ComposeN.prems(2) that
                Fun_param_is_subterm[OF that]
                intruder_deduct.Compose[OF ComposeN.hyps(1,2)]
                deduct_if_deduct_num[OF ComposeN.hyps(3)]
          by blast

        have "⟨M; P⟩ ⊢r t" when "t ∈ set T" for t
          using ComposeN.prems(1) ComposeN.hyps(3)[OF that] *[OF that] **[OF that]
          by blast
        thus ?case
          by (metis intruder_deduct_restricted.ComposeR[OF ComposeN.hyps(1,2)] ComposeN.prems(2))
      next
        case (DecomposeN t K T ti steps l)
        show ?case
        proof (cases "P t")
          case True
          hence "⋀k. k ∈ set K ⟹ P k"
            using P_Ana_key DecomposeN.hyps(1,2,3) deduct_if_deduct_num
            by blast
          moreover have
              "⋀k m x. k ∈ set K ⟹ m < steps k ⟹ ⟨M; m⟩ ⊢n x ⟹ P x ⟹ ⟨M;P⟩ ⊢r x"
          proof -
            fix k m x assume *: "k ∈ set K" "m < steps k" "⟨M; m⟩ ⊢n x" "P x"
            have "steps k ∈ insert l (steps ` set K)" using *(1) by simp
            hence "m < Suc (Max (insert l (steps ` set K)))"
              using less_trans[OF *(2), of "Suc (Max (insert l (steps ` set K)))"]
                    Max_less_iff[of "insert l (steps ` set K)"
                                    "Suc (Max (insert l (steps ` set K)))"]
              by auto
            thus "⟨M;P⟩ ⊢r x" using DecomposeN.prems(1) *(3,4) by simp
          qed
          ultimately have "⋀k. k ∈ set K ⟹ ⟨M; P⟩ ⊢r k"
            using DecomposeN.IH(2) by auto
          moreover have "⟨M; P⟩ ⊢r t"
            using True DecomposeN.prems(1) DecomposeN.hyps(1) le_imp_less_Suc
                  Max_less_iff[of "insert l (steps ` set K)" "Suc (Max (insert l (steps ` set K)))"]
            by blast
          ultimately show ?thesis
            using intruder_deduct_restricted.DecomposeR[OF _ DecomposeN.hyps(2)
                                                           _ DecomposeN.hyps(4)]
            by metis
        next
          case False
          obtain g S where gS: "t = Fun g S" using DecomposeN.hyps(2,4) by (cases t) moura+
          hence *: "Fun g S ⊑ t" "¬P (Fun g S)" using False by force+
          have "∃j<l. ⟨M; j⟩ ⊢n ti"
            using gS DecomposeN.hyps(2,4) Ana_fun_subterm[of g S K T]
                  deduct_normalize[of M "λf T. P (Fun f T)", OF M DecomposeN.hyps(1) *]
            by force
```

```
            hence "∃j<Suc (Max (insert 1 (steps ‘ set K))). ⟨M; j⟩ ⊢n ti"
              using Max_less_iff[of "insert 1 (steps ‘ set K)"
                                    "Suc (Max (insert 1 (steps ‘ set K)))"]
                    less_trans[of _ 1 "Suc (Max (insert 1 (steps ‘ set K)))"]
              by blast
            thus ?thesis using DecomposeN.prems(1,2) by meson
          qed
        qed auto
      qed
    qed
  } thus ?thesis using deduct_num_if_deduct m(1) by metis
qed

lemma restricted_deduct_if_deduct':
  assumes "∀m ∈ M. P m"
    and "∀t t'. P t ⟶ t' ⊑ t ⟶ P t'"
    and "∀t K T k. P t ⟶ Ana t = (K, T) ⟶ k ∈ set K ⟶ P k"
    and "M ⊢ m" "P m"
  shows "⟨M; P⟩ ⊢r m"
using restricted_deduct_if_deduct[of M P m] assms
by blast

lemma private_const_deduct:
  assumes c: "¬public c" "M ⊢ (Fun c []::('fun,'var) term)"
  shows "Fun c [] ∈ M ∨
         (∃m ∈ subtermsset M. M ⊢ m ∧ (∀k ∈ set (fst (Ana m)). M ⊢ m) ∧
                              Fun c [] ∈ set (snd (Ana m)))"
proof -
  obtain n where "⟨M; n⟩ ⊢n Fun c []"
    using c(2) deduct_num_if_deduct by moura
  hence "Fun c [] ∈ M ∨
         (∃m ∈ subtermsset M.
            (∃l < n. ⟨M; l⟩ ⊢n m) ∧
            (∀k ∈ set (fst (Ana m)). ∃l < n. ⟨M; l⟩ ⊢n k) ∧ Fun c [] ∈ set (snd (Ana m)))"
    using deduct_inv[of M n "Fun c []"] c(1) by fast
  thus ?thesis using deduct_if_deduct_num[of M] by blast
qed

lemma private_fun_deduct_in_ik'':
  assumes t: "M ⊢ Fun f T" "Fun c [] ∈ set T" "∀m ∈ subtermsset M. Fun f T ∉ set (snd (Ana m))"
    and c: "¬public c" "Fun c [] ∉ M" "∀m ∈ subtermsset M. Fun c [] ∉ set (snd (Ana m))"
  shows "Fun f T ∈ M"
proof -
  have *: "∄n. ⟨M; n⟩ ⊢n Fun c []"
    using private_const_deduct[OF c(1)] c(2,3) deduct_if_deduct_num
    by blast

  obtain n where n: "⟨M; n⟩ ⊢n Fun f T"
    using t(1) deduct_num_if_deduct
    by blast

  show ?thesis
    using deduct_inv[OF n] t(2,3) *
    by blast
qed

end
```

2.4.10 Executable Definitions for Code Generation

```
fun intruder_synth' where
  "intruder_synth' pu ar M (Var x) = (Var x ∈ M)"
| "intruder_synth' pu ar M (Fun f T) = (
```

```
    Fun f T ∈ M ∨ (pu f ∧ length T = ar f ∧ list_all (intruder_synth' pu ar M) T))"

definition "wf_trm' ar t ≡ (∀s ∈ subterms t. is_Fun s ⟶ ar (the_Fun s) = length (args s))"

definition "wf_trms' ar M ≡ (∀t ∈ M. wf_trm' ar t)"

definition "analyzed_in' An pu ar t M ≡ (case An t of
    (K,T) ⇒ (∀k ∈ set K. intruder_synth' pu ar M k) ⟶ (∀s ∈ set T. intruder_synth' pu ar M s))"

lemma (in intruder_model) intruder_synth'_induct[consumes 1, case_names Var Fun]:
  assumes "intruder_synth' public arity M t"
          "⋀x. intruder_synth' public arity M (Var x) ⟹ P (Var x)"
          "⋀f T. (⋀z. z ∈ set T ⟹ intruder_synth' public arity M z ⟹ P z) ⟹
                intruder_synth' public arity M (Fun f T) ⟹ P (Fun f T) "
  shows "P t"
using assms by (induct public arity M t rule: intruder_synth'.induct) auto

lemma (in intruder_model) wf_trm_code[code_unfold]:
  "wf_trm t = wf_trm' arity t"
unfolding wf_trm_def wf_trm'_def
by auto

lemma (in intruder_model) wf_trms_code[code_unfold]:
  "wf_trms M = wf_trms' arity M"
using wf_trm_code
unfolding wf_trms'_def
by auto

lemma (in intruder_model) intruder_synth_code[code_unfold]:
  "intruder_synth M t = intruder_synth' public arity M t"
  (is "?A ⟷ ?B")
proof
  show "?A ⟹ ?B"
  proof (induction t rule: intruder_synth_induct)
    case (AxiomC t) thus ?case by (cases t) auto
  qed (fastforce simp add: list_all_iff)

  show "?B ⟹ ?A"
  proof (induction t rule: intruder_synth'_induct)
    case (Fun f T) thus ?case
    proof (cases "Fun f T ∈ M")
      case False
      hence "public f" "length T = arity f" "list_all (intruder_synth' public arity M) T"
        using Fun.hyps by fastforce+
      thus ?thesis
        using Fun.IH intruder_synth.ComposeC[of T f M] Ball_set[of T]
        by blast
    qed simp
  qed simp
qed

lemma (in intruder_model) analyzed_in_code[code_unfold]:
  "analyzed_in t M = analyzed_in' Ana public arity t M"
using intruder_synth_code[of M]
unfolding analyzed_in_def analyzed_in'_def
by fastforce

end
```

3 The Typing Result for Non-Stateful Protocols

In this chapter, we formalize and prove a typing result for "stateless" security protocols. This work is described in more detail in [2] and [1, chapter 3].

3.1 Strands and Symbolic Intruder Constraints (Strands_and_Constraints)

```
theory Strands_and_Constraints
imports Messages More_Unification Intruder_Deduction
begin
```

3.1.1 Constraints, Strands and Related Definitions

```
datatype poscheckvariant = Assign ("assign") | Check ("check")
```

A strand (or constraint) step is either a message transmission (either a message being sent *Send* or being received *Receive*) or a check on messages (a positive check *Equality*—which can be either an "assignment" or just a check—or a negative check *Inequality*)

```
datatype (funs_stp: 'a, vars_stp: 'b) strand_step =
  Send       "('a,'b) term" ("send⟨_⟩_st" 80)
| Receive    "('a,'b) term" ("receive⟨_⟩_st" 80)
| Equality   poscheckvariant "('a,'b) term" "('a,'b) term" ("⟨_: _ ≐ _⟩_st" [80,80])
| Inequality (bvars_stp: "'b list") "(('a,'b) term × ('a,'b) term) list" ("∀_⟨∨≠: _⟩_st" [80,80])
where
  "bvars_stp (Send _) = []"
| "bvars_stp (Receive _) = []"
| "bvars_stp (Equality _ _ _) = []"
```

A strand is a finite sequence of strand steps (constraints and strands share the same datatype)

```
type_synonym ('a,'b) strand = "('a,'b) strand_step list"

type_synonym ('a,'b) strands = "('a,'b) strand set"

abbreviation "trms_pairs F ≡ ⋃(t,t') ∈ set F. {t,t'}"

fun trms_stp::"('a,'b) strand_step ⇒ ('a,'b) terms" where
  "trms_stp (Send t) = {t}"
| "trms_stp (Receive t) = {t}"
| "trms_stp (Equality _ t t') = {t,t'}"
| "trms_stp (Inequality _ F) = trms_pairs F"

lemma vars_stp_unfold[simp]: "vars_stp x = fv_set (trms_stp x) ∪ set (bvars_stp x)"
by (cases x) auto
```

The set of terms occurring in a strand

```
definition trms_st where "trms_st S ≡ ⋃(trms_stp ` set S)"

fun trms_list_stp::"('a,'b) strand_step ⇒ ('a,'b) term list" where
  "trms_list_stp (Send t) = [t]"
| "trms_list_stp (Receive t) = [t]"
| "trms_list_stp (Equality _ t t') = [t,t']"
| "trms_list_stp (Inequality _ F) = concat (map (λ(t,t'). [t,t']) F)"
```

The set of terms occurring in a strand (list variant)

```
definition trms_list_st where "trms_list_st S ≡ remdups (concat (map trms_list_stp S))"
```

The set of variables occurring in a sent message

```
definition fv_snd::"('a,'b) strand_step ⇒ 'b set" where
  "fv_snd x ≡ case x of Send t ⇒ fv t | _ ⇒ {}"
```

The set of variables occurring in a received message

```
definition fv_rcv::"('a,'b) strand_step ⇒ 'b set" where
  "fv_rcv x ≡ case x of Receive t ⇒ fv t | _ ⇒ {}"
```

The set of variables occurring in an equality constraint

```
definition fv_eq::"poscheckvariant ⇒ ('a,'b) strand_step ⇒ 'b set" where
  "fv_eq ac x ≡ case x of Equality ac' s t ⇒ if ac = ac' then fv s ∪ fv t else {} | _ ⇒ {}"
```

The set of variables occurring at the left-hand side of an equality constraint

```
definition fv_l_eq::"poscheckvariant ⇒ ('a,'b) strand_step ⇒ 'b set" where
  "fv_l_eq ac x ≡ case x of Equality ac' s t ⇒ if ac = ac' then fv s else {} | _ ⇒ {}"
```

The set of variables occurring at the right-hand side of an equality constraint

```
definition fv_r_eq::"poscheckvariant ⇒ ('a,'b) strand_step ⇒ 'b set" where
  "fv_r_eq ac x ≡ case x of Equality ac' s t ⇒ if ac = ac' then fv t else {} | _ ⇒ {}"
```

The free variables of inequality constraints

```
definition fv_ineq::"('a,'b) strand_step ⇒ 'b set" where
  "fv_ineq x ≡ case x of Inequality X F ⇒ fv_pairs F - set X | _ ⇒ {}"
```

```
fun fv_stp::"('a,'b) strand_step ⇒ 'b set" where
  "fv_stp (Send t) = fv t"
| "fv_stp (Receive t) = fv t"
| "fv_stp (Equality _ t t') = fv t ∪ fv t'"
| "fv_stp (Inequality X F) = (⋃(t,t') ∈ set F. fv t ∪ fv t') - set X"
```

The set of free variables of a strand

```
definition fv_st::"('a,'b) strand ⇒ 'b set" where
  "fv_st S ≡ ⋃(set (map fv_stp S))"
```

The set of bound variables of a strand

```
definition bvars_st::"('a,'b) strand ⇒ 'b set" where
  "bvars_st S ≡ ⋃(set (map (set ∘ bvars_stp) S))"
```

The set of all variables occurring in a strand

```
definition vars_st::"('a,'b) strand ⇒ 'b set" where
  "vars_st S ≡ ⋃(set (map vars_stp S))"
```

```
abbreviation wfrestrictedvars_stp::"('a,'b) strand_step ⇒ 'b set" where
  "wfrestrictedvars_stp x ≡
    case x of Inequality _ _ ⇒ {} | Equality Check _ _ ⇒ {} | _ ⇒ vars_stp x"
```

The variables of a strand whose occurrences might be restricted by well-formedness constraints

```
definition wfrestrictedvars_st::"('a,'b) strand ⇒ 'b set" where
  "wfrestrictedvars_st S ≡ ⋃(set (map wfrestrictedvars_stp S))"
```

```
abbreviation wfvarsoccs_stp where
  "wfvarsoccs_stp x ≡ case x of Send t ⇒ fv t | Equality Assign s t ⇒ fv s | _ ⇒ {}"
```

The variables of a strand that occur in sent messages or as variables in assignments

```
definition wfvarsoccs_st where
  "wfvarsoccs_st S ≡ ⋃(set (map wfvarsoccs_stp S))"
```

The variables occurring at the right-hand side of assignment steps

```
fun assignment_rhs_st where
  "assignment_rhs_st [] = {}"
| "assignment_rhs_st (Equality Assign t t'#S) = insert t' (assignment_rhs_st S)"
```

```
| "assignment_rhs_st (x#S) = assignment_rhs_st S"
```

The set function symbols occurring in a strand

```
definition funs_st::"('a,'b) strand ⇒ 'a set" where
  "funs_st S ≡ ⋃(set (map funs_stp S))"

fun subst_apply_strand_step::"('a,'b) strand_step ⇒ ('a,'b) subst ⇒ ('a,'b) strand_step"
  (infix "·_stp" 51) where
  "Send t ·_stp ϑ = Send (t · ϑ)"
| "Receive t ·_stp ϑ = Receive (t · ϑ)"
| "Equality a t t' ·_stp ϑ = Equality a (t · ϑ) (t' · ϑ)"
| "Inequality X F ·_stp ϑ = Inequality X (F ·_pairs rm_vars (set X) ϑ)"
```

Substitution application for strands

```
definition subst_apply_strand::"('a,'b) strand ⇒ ('a,'b) subst ⇒ ('a,'b) strand"
  (infix "·_st" 51) where
  "S ·_st ϑ ≡ map (λx. x ·_stp ϑ) S"
```

The semantics of inequality constraints

```
definition
  "ineq_model (ℐ::('a,'b) subst) X F ≡
     (∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶
            list_ex (λf. fst f · (δ ∘_s ℐ) ≠ snd f · (δ ∘_s ℐ)) F)"

fun simple_stp where
  "simple_stp (Receive t) = True"
| "simple_stp (Send (Var v)) = True"
| "simple_stp (Inequality X F) = (∃ℐ. ineq_model ℐ X F)"
| "simple_stp _ = False"
```

Simple constraints

```
definition simple where "simple S ≡ list_all simple_stp S"
```

The intruder knowledge of a constraint

```
fun ik_st::"('a,'b) strand ⇒ ('a,'b) terms" where
  "ik_st [] = {}"
| "ik_st (Receive t#S) = insert t (ik_st S)"
| "ik_st (_#S) = ik_st S"
```

Strand well-formedness

```
fun wf_st::"'b set ⇒ ('a,'b) strand ⇒ bool" where
  "wf_st V [] = True"
| "wf_st V (Receive t#S) = (fv t ⊆ V ∧ wf_st V S)"
| "wf_st V (Send t#S) = wf_st (V ∪ fv t) S"
| "wf_st V (Equality Assign s t#S) = (fv t ⊆ V ∧ wf_st (V ∪ fv s) S)"
| "wf_st V (Equality Check s t#S) = wf_st V S"
| "wf_st V (Inequality _ _#S) = wf_st V S"
```

Well-formedness of constraint states

```
definition wf_constr::"('a,'b) strand ⇒ ('a,'b) subst ⇒ bool" where
  "wf_constr S ϑ ≡ (wf_subst ϑ ∧ wf_st {} S ∧ subst_domain ϑ ∩ vars_st S = {} ∧
                 range_vars ϑ ∩ bvars_st S = {} ∧ fv_st S ∩ bvars_st S = {})"

declare trms_st_def[simp]
declare fv_snd_def[simp]
declare fv_rcv_def[simp]
declare fv_eq_def[simp]
declare fv_l_eq_def[simp]
declare fv_r_eq_def[simp]
declare fv_ineq_def[simp]
declare fv_st_def[simp]
declare vars_st_def[simp]
```

```
declare bvars_st_def[simp]
declare wfrestrictedvars_st_def[simp]
declare wfvarsoccs_st_def[simp]

lemmas wf_st_induct = wf_st.induct[case_names Nil ConsRcv ConsSnd ConsEq ConsEq2 ConsIneq]
lemmas ik_st_induct = ik_st.induct[case_names Nil ConsRcv ConsSnd ConsEq ConsIneq]
lemmas assignment_rhs_st_induct = assignment_rhs_st.induct[case_names Nil ConsEq2 ConsSnd ConsRcv
ConsEq ConsIneq]
```

Lexicographical measure on strands

```
definition size_st::"('a,'b) strand ⇒ nat" where
  "size_st S ≡ size_list (λx. Max (insert 0 (size ` trms_stp x))) S"

definition measure_st::"((('a, 'b) strand × ('a,'b) subst) × ('a, 'b) strand × ('a,'b) subst) set"
where
  "measure_st ≡ measures [λ(S,ϑ). card (fv_st S), λ(S,ϑ). size_st S]"

lemma measure_st_alt_def:
  "((s,x),(t,y)) ∈ measure_st =
      (card (fv_st s) < card (fv_st t) ∨ (card (fv_st s) = card (fv_st t) ∧ size_st s < size_st t))"
by (simp add: measure_st_def size_st_def)

lemma measure_st_trans: "trans measure_st"
by (simp add: trans_def measure_st_def size_st_def)
```

Some lemmata

```
lemma trms_list_st_is_trms_st: "trms_st S = set (trms_list_st S)"
unfolding trms_st_def trms_list_st_def
proof (induction S)
  case (Cons x S) thus ?case by (cases x) auto
qed simp

lemma subst_apply_strand_step_def:
  "s ·_stp ϑ = (case s of
    Send t ⇒ Send (t · ϑ)
  | Receive t ⇒ Receive (t · ϑ)
  | Equality a t t' ⇒ Equality a (t · ϑ) (t' · ϑ)
  | Inequality X F ⇒ Inequality X (F ·_pairs rm_vars (set X) ϑ))"
by (cases s) simp_all

lemma subst_apply_strand_nil[simp]: "[] ·_st δ = []"
unfolding subst_apply_strand_def by simp

lemma finite_funs_stp[simp]: "finite (funs_stp x)" by (cases x) auto
lemma finite_funs_st[simp]: "finite (funs_st S)" unfolding funs_st_def by simp
lemma finite_trms_pairs[simp]: "finite (trms_pairs x)" by (induct x) auto
lemma finite_trms_stp[simp]: "finite (trms_stp x)" by (cases x) auto
lemma finite_vars_stp[simp]: "finite (vars_stp x)" by auto
lemma finite_bvars_stp[simp]: "finite (set (bvars_stp x))" by rule
lemma finite_fv_snd[simp]: "finite (fv_snd x)" by (cases x) auto
lemma finite_fv_rcv[simp]: "finite (fv_rcv x)" by (cases x) auto
lemma finite_fv_stp[simp]: "finite (fv_stp x)" by (cases x) auto
lemma finite_vars_st[simp]: "finite (vars_st S)" by simp
lemma finite_bvars_st[simp]: "finite (bvars_st S)" by simp
lemma finite_fv_st[simp]: "finite (fv_st S)" by simp

lemma finite_wfrestrictedvars_stp[simp]: "finite (wfrestrictedvars_stp x)"
by (cases x) (auto split: poscheckvariant.splits)

lemma finite_wfrestrictedvars_st[simp]: "finite (wfrestrictedvars_st S)"
using finite_wfrestrictedvars_stp by auto
```

```
lemma finite_wfvarsoccs_stp[simp]: "finite (wfvarsoccs_stp x)"
by (cases x) (auto split: poscheckvariant.splits)

lemma finite_wfvarsoccs_st[simp]: "finite (wfvarsoccs_st S)"
using finite_wfvarsoccs_stp by auto

lemma finite_ik_st[simp]: "finite (ik_st S)"
by (induct S rule: ik_st.induct) simp_all

lemma finite_assignment_rhs_st[simp]: "finite (assignment_rhs_st S)"
by (induct S rule: assignment_rhs_st.induct) simp_all

lemma ik_st_is_rcv_set: "ik_st A = {t. Receive t ∈ set A}"
by (induct A rule: ik_st.induct) auto

lemma ik_stD[dest]: "t ∈ ik_st S ⟹ Receive t ∈ set S"
by (induct S rule: ik_st.induct) auto

lemma ik_stD'[dest]: "t ∈ ik_st S ⟹ t ∈ trms_st S"
by (induct S rule: ik_st.induct) auto

lemma ik_stD''[dest]: "t ∈ subterms_set (ik_st S) ⟹ t ∈ subterms_set (trms_st S)"
by (induct S rule: ik_st.induct) auto

lemma ik_st_subterm_exD:
  assumes "t ∈ ik_st S"
  shows "∃x ∈ set S. t ∈ subterms_set (trms_stp x)"
using assms ik_stD by force

lemma assignment_rhs_stD[dest]: "t ∈ assignment_rhs_st S ⟹ ∃t'. Equality Assign t' t ∈ set S"
by (induct S rule: assignment_rhs_st.induct) auto

lemma assignment_rhs_stD'[dest]: "t ∈ subterms_set (assignment_rhs_st S) ⟹ t ∈ subterms_set (trms_st S)"
by (induct S rule: assignment_rhs_st.induct) auto

lemma bvars_st_split: "bvars_st (S@S') = bvars_st S ∪ bvars_st S'"
unfolding bvars_st_def by auto

lemma bvars_st_singleton: "bvars_st [x] = set (bvars_stp x)"
unfolding bvars_st_def by auto

lemma strand_fv_bvars_disjointD:
  assumes "fv_st S ∩ bvars_st S = {}" "Inequality X F ∈ set S"
  shows "set X ⊆ bvars_st S" "fv_pairs F - set X ⊆ fv_st S"
using assms by (induct S) fastforce+

lemma strand_fv_bvars_disjoint_unfold:
  assumes "fv_st S ∩ bvars_st S = {}" "Inequality X F ∈ set S" "Inequality Y G ∈ set S"
  shows "set Y ∩ (fv_pairs F - set X) = {}"
proof -
  have "set X ⊆ bvars_st S" "set Y ⊆ bvars_st S"
       "fv_pairs F - set X ⊆ fv_st S" "fv_pairs G - set Y ⊆ fv_st S"
    using strand_fv_bvars_disjointD[OF assms(1)] assms(2,3) by auto
  thus ?thesis using assms(1) by fastforce
qed

lemma strand_subst_hom[iff]:
  "(S@S') ·_st ϑ = (S ·_st ϑ)@(S' ·_st ϑ)" "(x#S) ·_st ϑ = (x ·_stp ϑ)#(S ·_st ϑ)"
unfolding subst_apply_strand_def by auto

lemma strand_subst_comp: "range_vars δ ∩ bvars_st S = {} ⟹ S ·_st δ ∘_s ϑ = ((S ·_st δ) ·_st ϑ)"
proof (induction S)
```

```
  case (Cons x S)
  have *: "range_vars δ ∩ bvars_st S = {}" "range_vars δ ∩ (set (bvars_stp x)) = {}"
    using Cons bvars_st_split[of "[x]" S] append_Cons inf_sup_absorb
    by (metis (no_types, lifting) Int_iff Un_commute disjoint_iff_not_equal self_append_conv2,
        metis append_self_conv2 bvars_st_singleton inf_bot_right inf_left_commute)
  hence IH: "S ·st δ ∘s ϑ = (S ·st δ) ·st ϑ" using Cons.IH by auto
  have "(x#S ·st δ ∘s ϑ) = (x ·stp δ ∘s ϑ)#(S ·st δ ∘s ϑ)" by (metis strand_subst_hom(2))
  hence "... = (x ·stp δ ∘s ϑ)#((S ·st δ) ·st ϑ)" by (metis IH)
  hence "... = ((x ·stp δ) ·stp ϑ)#((S ·st δ) ·st ϑ)" using rm_vars_comp[OF *(2)]
  proof (induction x)
    case (Inequality X F) thus ?case
      by (induct F) (auto simp add: subst_apply_pairs_def subst_apply_strand_step_def)
  qed (simp_all add: subst_apply_strand_step_def)
  thus ?case using IH by auto
qed (simp add: subst_apply_strand_def)

lemma strand_substI[intro]:
  "subst_domain ϑ ∩ fv_st S = {} ⟹ S ·st ϑ = S"
  "subst_domain ϑ ∩ vars_st S = {} ⟹ S ·st ϑ = S"
proof -
  show "subst_domain ϑ ∩ vars_st S = {} ⟹ S ·st ϑ = S"
  proof (induction S)
    case (Cons x S)
    hence "S ·st ϑ = S" by auto
    moreover have "vars_stp x ∩ subst_domain ϑ = {}" using Cons.prems by auto
    hence "x ·stp ϑ = x"
    proof (induction x)
      case (Inequality X F) thus ?case
        by (induct F) (force simp add: subst_apply_pairs_def)+
    qed auto
    ultimately show ?case by simp
  qed (simp add: subst_apply_strand_def)

  show "subst_domain ϑ ∩ fv_st S = {} ⟹ S ·st ϑ = S"
  proof (induction S)
    case (Cons x S)
    hence "S ·st ϑ = S" by auto
    moreover have "fv_stp x ∩ subst_domain ϑ = {}"
      using Cons.prems by auto
    hence "x ·stp ϑ = x"
    proof (induction x)
      case (Inequality X F) thus ?case
        by (induct F) (force simp add: subst_apply_pairs_def)+
    qed auto
    ultimately show ?case by simp
  qed (simp add: subst_apply_strand_def)
qed

lemma strand_substI':
  "fv_st S = {} ⟹ S ·st ϑ = S"
  "vars_st S = {} ⟹ S ·st ϑ = S"
by (metis inf_bot_right strand_substI(1),
    metis inf_bot_right strand_substI(2))

lemma strand_subst_set: "(set (S ·st ϑ)) = ((λx. x ·stp ϑ) ` (set S))"
by (auto simp add: subst_apply_strand_def)

lemma strand_map_inv_set_snd_rcv_subst:
  assumes "finite (M::('a,'b) terms)"
  shows "set ((map Send (inv set M)) ·st ϑ) = Send ` (M ·set ϑ)" (is ?A)
        "set ((map Receive (inv set M)) ·st ϑ) = Receive ` (M ·set ϑ)" (is ?B)
proof -
  { fix f::"('a,'b) term ⇒ ('a,'b) strand_step" assume f: "f = Send ∨ f = Receive"
```

```
    from assms have "set ((map f (inv set M)) ·_st ϑ) = f ` (M ·_set ϑ)"
    proof (induction rule: finite_induct)
      case empty thus ?case unfolding inv_def by auto
    next
      case (insert m M)
      have "set (map f (inv set (insert m M)) ·_st ϑ) =
            insert (f m ·_stp ϑ) (set (map f (inv set M) ·_st ϑ))"
        by (simp add: insert.hyps(1) inv_set_fset subst_apply_strand_def)
      thus ?case using f insert.IH by auto
    qed
  }
  thus "?A" "?B" by auto
qed

lemma strand_ground_subst_vars_subset:
  assumes "ground (subst_range ϑ)" shows "vars_st (S ·_st ϑ) ⊆ vars_st S"
proof (induction S)
  case (Cons x S)
  have "vars_stp (x ·_stp ϑ) ⊆ vars_stp x" using ground_subst_fv_subset[OF assms]
  proof (cases x)
    case (Inequality X F)
    let ?ϑ = "rm_vars (set X) ϑ"
    have "fv_pairs (F ·_pairs ?ϑ) ⊆ fv_pairs F"
    proof (induction F)
      case (Cons f F)
      obtain t t' where f: "f = (t,t')" by (metis surj_pair)
      hence "fv_pairs (f#F ·_pairs ?ϑ) = fv (t · ?ϑ) ∪ fv (t' · ?ϑ) ∪ fv_pairs (F ·_pairs ?ϑ)"
            "fv_pairs (f#F) = fv t ∪ fv t' ∪ fv_pairs F"
        by (auto simp add: subst_apply_pairs_def)
      thus ?case
        using ground_subst_fv_subset[OF ground_subset[OF rm_vars_img_subset assms, of "set X"]]
              Cons.IH
        by (metis (no_types, lifting) Un_mono)
    qed (simp add: subst_apply_pairs_def)
    moreover have
        "vars_stp (x ·_stp ϑ) = fv_pairs (F ·_pairs rm_vars (set X) ϑ) ∪ set X"
        "vars_stp x = fv_pairs F ∪ set X"
      using Inequality
      by (auto simp add: subst_apply_pairs_def)
    ultimately show ?thesis by auto
  qed auto
  thus ?case using Cons.IH by auto
qed (simp add: subst_apply_strand_def)

lemma ik_union_subset: "⋃(P ` ik_st S) ⊆ (⋃x ∈ (set S). ⋃(P ` trms_stp x))"
by (induct S rule: ik_st.induct) auto

lemma ik_snd_empty[simp]: "ik_st (map Send X) = {}"
by (induct "map Send X" arbitrary: X rule: ik_st.induct) auto

lemma ik_snd_empty'[simp]: "ik_st [Send t] = {}" by simp

lemma ik_append[iff]: "ik_st (S@S') = ik_st S ∪ ik_st S'" by (induct S rule: ik_st.induct) auto

lemma ik_cons: "ik_st (x#S) = ik_st [x] ∪ ik_st S" using ik_append[of "[x]" S] by simp

lemma assignment_rhs_append[iff]: "assignment_rhs_st (S@S') = assignment_rhs_st S ∪ assignment_rhs_st
S'"
by (induct S rule: assignment_rhs_st.induct) auto

lemma eqs_rcv_map_empty: "assignment_rhs_st (map Receive M) = {}"
by auto
```

```
lemma ik_rcv_map: assumes "t ∈ set L" shows "t ∈ ik_st (map Receive L)"
proof -
  { fix L L'
    have "t ∈ ik_st [Receive t]" by auto
    hence "t ∈ ik_st (map Receive L@Receive t#map Receive L')" using ik_append by auto
    hence "t ∈ ik_st (map Receive (L@t#L'))" by auto
  }
  thus ?thesis using assms split_list_last by force
qed

lemma ik_subst: "ik_st (S ·_st δ) = ik_st S ·_set δ"
by (induct rule: ik_st_induct) auto

lemma ik_rcv_map': assumes "t ∈ ik_st (map Receive L)" shows "t ∈ set L"
using assms by force

lemma ik_append_subset[simp]: "ik_st S ⊆ ik_st (S@S')" "ik_st S' ⊆ ik_st (S@S')"
by (induct S rule: ik_st.induct) auto

lemma assignment_rhs_append_subset[simp]:
  "assignment_rhs_st S ⊆ assignment_rhs_st (S@S')"
  "assignment_rhs_st S' ⊆ assignment_rhs_st (S@S')"
by (induct S rule: assignment_rhs_st.induct) auto

lemma trms_st_cons: "trms_st (x#S) = trms_stp x ∪ trms_st S" by simp

lemma trm_strand_subst_cong:
  "t ∈ trms_st S ⟹ t · δ ∈ trms_st (S ·_st δ)
    ∨ (∃X F. Inequality X F ∈ set S ∧ t · rm_vars (set X) δ ∈ trms_st (S ·_st δ))"
    (is "t ∈ trms_st S ⟹ ?P t δ S")
  "t ∈ trms_st (S ·_st δ) ⟹ (∃t'. t = t' · δ ∧ t' ∈ trms_st S)
    ∨ (∃X F. Inequality X F ∈ set S ∧ (∃t' ∈ trms_pairs F. t = t' · rm_vars (set X) δ))"
    (is "t ∈ trms_st (S ·_st δ) ⟹ ?Q t δ S")
proof -
  show "t ∈ trms_st S ⟹ ?P t δ S"
  proof (induction S)
    case (Cons x S) show ?case
    proof (cases "t ∈ trms_st S")
      case True
      hence "?P t δ S" using Cons by simp
      thus ?thesis
        by (cases x)
           (metis (no_types, lifting) Un_iff list.set_intros(2) strand_subst_hom(2) trms_st_cons)+
    next
      case False
      hence "t ∈ trms_stp x" using Cons.prems by auto
      thus ?thesis
      proof (induction x)
        case (Inequality X F)
        hence "t · rm_vars (set X) δ ∈ trms_stp (Inequality X F ·_stp δ)"
          by (induct F) (auto simp add: subst_apply_pairs_def subst_apply_strand_step_def)
        thus ?case by fastforce
      qed (auto simp add: subst_apply_strand_step_def)
    qed
  qed simp

  show "t ∈ trms_st (S ·_st δ) ⟹ ?Q t δ S"
  proof (induction S)
    case (Cons x S) show ?case
    proof (cases "t ∈ trms_st (S ·_st δ)")
      case True
      hence "?Q t δ S" using Cons by simp
      thus ?thesis by (cases x) force+
```

```
    next
      case False
      hence "t ∈ trms_stp (x ·_stp δ)" using Cons.prems by auto
      thus ?thesis
      proof (induction x)
        case (Inequality X F)
        hence "t ∈ trms_stp (Inequality X F) ·_set rm_vars (set X) δ"
          by (induct F) (force simp add: subst_apply_pairs_def)+
        thus ?case by fastforce
      qed (auto simp add: subst_apply_strand_step_def)
    qed
  qed simp
qed
```

3.1.2 Lemmata: Free Variables of Strands

```
lemma fv_trm_snd_rcv[simp]: "fv_set (trms_stp (Send t)) = fv t" "fv_set (trms_stp (Receive t)) = fv t"
by simp_all

lemma in_strand_fv_subset: "x ∈ set S ⟹ vars_stp x ⊆ vars_st S" by fastforce
lemma in_strand_fv_subset_snd: "Send t ∈ set S ⟹ fv t ⊆ ⋃(set (map fv_snd S))" by auto
lemma in_strand_fv_subset_rcv: "Receive t ∈ set S ⟹ fv t ⊆ ⋃(set (map fv_rcv S))" by auto

lemma fv_sndE:
  assumes "v ∈ ⋃(set (map fv_snd S))"
  obtains t where "send⟨t⟩_st ∈ set S" "v ∈ fv t"
proof -
  have "∃t. send⟨t⟩_st ∈ set S ∧ v ∈ fv t"
    by (metis (no_types, lifting) assms UN_E empty_iff set_map strand_step.case_eq_if
              fv_snd_def strand_step.collapse(1))
  thus ?thesis by (metis that)
qed

lemma fv_rcvE:
  assumes "v ∈ ⋃(set (map fv_rcv S))"
  obtains t where "receive⟨t⟩_st ∈ set S" "v ∈ fv t"
proof -
  have "∃t. receive⟨t⟩_st ∈ set S ∧ v ∈ fv t"
    by (metis (no_types, lifting) assms UN_E empty_iff set_map strand_step.case_eq_if
              fv_rcv_def strand_step.collapse(2))
  thus ?thesis by (metis that)
qed

lemma vars_stpI[intro]: "x ∈ fv_stp s ⟹ x ∈ vars_stp s"
by (induct s rule: fv_stp.induct) auto

lemma vars_stI[intro]: "x ∈ fv_st S ⟹ x ∈ vars_st S" using vars_stpI by fastforce

lemma fv_st_subset_vars_st[simp]: "fv_st S ⊆ vars_st S" using vars_stI by force

lemma vars_st_is_fv_st_bvars_st: "vars_st S = fv_st S ∪ bvars_st S"
proof (induction S)
  case (Cons x S) thus ?case
  proof (induction x)
    case (Inequality X F) thus ?case by (induct F) auto
  qed auto
qed simp

lemma fv_stp_is_subterm_trms_stp: "x ∈ fv_stp a ⟹ Var x ∈ subterms_set (trms_stp a)"
using var_is_subterm by (cases a) force+

lemma fv_st_is_subterm_trms_st: "x ∈ fv_st A ⟹ Var x ∈ subterms_set (trms_st A)"
proof (induction A)
```

```
  case (Cons a A) thus ?case using fv_stp_is_subterm_trms_stp by (cases "x ∈ fv_st A") auto
qed simp

lemma vars_st_snd_map: "vars_st (map Send X) = fv (Fun f X)" by auto

lemma vars_st_rcv_map: "vars_st (map Receive X) = fv (Fun f X)" by auto

lemma vars_snd_rcv_union:
  "vars_stp x = fv_snd x ∪ fv_rcv x ∪ fv_eq assign x ∪ fv_eq check x ∪ fv_ineq x ∪ set (bvars_stp x)"
proof (cases x)
  case (Equality ac t t') thus ?thesis by (cases ac) auto
qed auto

lemma fv_snd_rcv_union:
  "fv_stp x = fv_snd x ∪ fv_rcv x ∪ fv_eq assign x ∪ fv_eq check x ∪ fv_ineq x"
proof (cases x)
  case (Equality ac t t') thus ?thesis by (cases ac) auto
qed auto

lemma fv_snd_rcv_empty[simp]: "fv_snd x = {} ∨ fv_rcv x = {}" by (cases x) simp_all

lemma vars_snd_rcv_strand[iff]:
  "vars_st (S::('a,'b) strand) =
    (⋃(set (map fv_snd S))) ∪ (⋃(set (map fv_rcv S))) ∪ (⋃(set (map (fv_eq assign) S)))
    ∪ (⋃(set (map (fv_eq check) S))) ∪ (⋃(set (map fv_ineq S))) ∪ bvars_st S"
unfolding bvars_st_def
proof (induction S)
  case (Cons x S)
  have "⋀s V. vars_stp (s::('a,'b) strand_step) ∪ V =
              fv_snd s ∪ fv_rcv s ∪ fv_eq assign s ∪ fv_eq check s ∪ fv_ineq s ∪ set (bvars_stp s) ∪ V"
    by (metis vars_snd_rcv_union)
  thus ?case using Cons.IH by (auto simp add: sup_assoc sup_left_commute)
qed simp

lemma fv_snd_rcv_strand[iff]:
  "fv_st (S::('a,'b) strand) =
    (⋃(set (map fv_snd S))) ∪ (⋃(set (map fv_rcv S))) ∪ (⋃(set (map (fv_eq assign) S)))
    ∪ (⋃(set (map (fv_eq check) S))) ∪ (⋃(set (map fv_ineq S)))"
unfolding bvars_st_def
proof (induction S)
  case (Cons x S)
  have "⋀s V. fv_stp (s::('a,'b) strand_step) ∪ V =
              fv_snd s ∪ fv_rcv s ∪ fv_eq assign s ∪ fv_eq check s ∪ fv_ineq s ∪ V"
    by (metis fv_snd_rcv_union)
  thus ?case using Cons.IH by (auto simp add: sup_assoc sup_left_commute)
qed simp

lemma vars_snd_rcv_strand2[iff]:
  "wfrestrictedvars_st (S::('a,'b) strand) =
    (⋃(set (map fv_snd S))) ∪ (⋃(set (map fv_rcv S))) ∪ (⋃(set (map (fv_eq assign) S)))"
by (induct S) (auto simp add: split: strand_step.split poscheckvariant.split)

lemma fv_snd_rcv_strand_subset[simp]:
  "⋃(set (map fv_snd S)) ⊆ fv_st S" "⋃(set (map fv_rcv S)) ⊆ fv_st S"
  "⋃(set (map (fv_eq ac) S)) ⊆ fv_st S" "⋃(set (map fv_ineq S)) ⊆ fv_st S"
  "wfvarsoccs_st S ⊆ fv_st S"
proof -
  show "⋃(set (map fv_snd S)) ⊆ fv_st S" "⋃(set (map fv_rcv S)) ⊆ fv_st S" "⋃(set (map fv_ineq S)) ⊆
fv_st S"
    using fv_snd_rcv_strand[of S] by auto

  show "⋃(set (map (fv_eq ac) S)) ⊆ fv_st S"
    by (induct S) (auto split: strand_step.split poscheckvariant.split)
```

```
  show "wfvarsoccs_st S ⊆ fv_st S"
    by (induct S) (auto split: strand_step.split poscheckvariant.split)
qed

lemma vars_snd_rcv_strand_subset2[simp]:
  "⋃(set (map fv_snd S)) ⊆ wfrestrictedvars_st S" "⋃(set (map fv_rcv S)) ⊆ wfrestrictedvars_st S"
  "⋃(set (map (fv_eq assign) S)) ⊆ wfrestrictedvars_st S" "wfvarsoccs_st S ⊆ wfrestrictedvars_st S"
by (induction S) (auto split: strand_step.split poscheckvariant.split)

lemma wfrestrictedvars_st_subset_vars_st: "wfrestrictedvars_st S ⊆ vars_st S"
by (induction S) (auto split: strand_step.split poscheckvariant.split)

lemma subst_sends_strand_step_fv_to_img: "fv_stp (x ·_stp δ) ⊆ fv_stp x ∪ range_vars δ"
using subst_sends_fv_to_img[of _ δ]
proof (cases x)
  case (Inequality X F)
  let ?ϑ = "rm_vars (set X) δ"
  have "fv_pairs (F ·_pairs ?ϑ) ⊆ fv_pairs F ∪ range_vars ?ϑ"
  proof (induction F)
    case (Cons f F) thus ?case
      using subst_sends_fv_to_img[of _ ?ϑ]
      by (auto simp add: subst_apply_pairs_def)
  qed (auto simp add: subst_apply_pairs_def)
  hence "fv_pairs (F ·_pairs ?ϑ) ⊆ fv_pairs F ∪ range_vars δ"
    using rm_vars_img_subset[of "set X" δ] fv_set_mono
    unfolding range_vars_alt_def by blast+
  thus ?thesis using Inequality by (auto simp add: subst_apply_strand_step_def)
qed (auto simp add: subst_apply_strand_step_def)

lemma subst_sends_strand_fv_to_img: "fv_st (S ·_st δ) ⊆ fv_st S ∪ range_vars δ"
proof (induction S)
  case (Cons x S)
  have *: "fv_st (x#S ·_st δ) = fv_stp (x ·_stp δ) ∪ fv_st (S ·_st δ)"
          "fv_st (x#S) ∪ range_vars δ = fv_stp x ∪ fv_st S ∪ range_vars δ"
    by auto
  thus ?case using Cons.IH subst_sends_strand_step_fv_to_img[of x δ] by auto
qed simp

lemma ineq_apply_subst:
  assumes "subst_domain δ ∩ set X = {}"
  shows "(Inequality X F) ·_stp δ = Inequality X (F ·_pairs δ)"
using rm_vars_apply'[OF assms] by (simp add: subst_apply_strand_step_def)

lemma fv_strand_step_subst:
  assumes "P = fv_stp ∨ P = fv_rcv ∨ P = fv_snd ∨ P = fv_eq ac ∨ P = fv_ineq"
  and "set (bvars_stp x) ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv_set (δ ‘ (P x)) = P (x ·_stp δ)"
proof (cases x)
  case (Send t)
  hence "vars_stp x = fv t" "fv_snd x = fv t" by auto
  thus ?thesis using assms Send subst_apply_fv_unfold[of _ δ] by auto
next
  case (Receive t)
  hence "vars_stp x = fv t" "fv_rcv x = fv t" by auto
  thus ?thesis using assms Receive subst_apply_fv_unfold[of _ δ] by auto
next
  case (Equality ac' t t') show ?thesis
  proof (cases "ac = ac'")
    case True
    hence "vars_stp x = fv t ∪ fv t'" "fv_eq ac x = fv t ∪ fv t'"
      using Equality
      by auto
```

```
    thus ?thesis
      using assms Equality subst_apply_fv_unfold[of _ δ] True
      by auto
  next
    case False
    hence "vars_stp x = fv t ∪ fv t'" "fv_eq ac x = {}"
      using Equality
      by auto
    thus ?thesis
      using assms Equality subst_apply_fv_unfold[of _ δ] False
      by auto
  qed
next
  case (Inequality X F)
  hence 1: "set X ∩ (subst_domain δ ∪ range_vars δ) = {}"
           "x ·_stp δ = Inequality X (F ·_pairs δ)"
           "rm_vars (set X) δ = δ"
    using assms ineq_apply_subst[of δ X F] rm_vars_apply'[of δ "set X"]
    unfolding range_vars_alt_def by force+

  have 2: "fv_ineq x = fv_pairs F - set X" using Inequality by auto
  hence "fv_set (δ ` fv_ineq x) = fv_set (δ ` fv_pairs F) - set X"
    using fv_set_subst_img_eq[OF 1(1), of "fv_pairs F"] by simp
  hence 3: "fv_set (δ ` fv_ineq x) = fv_pairs (F ·_pairs δ) - set X" by (metis fv_pairs_step_subst)

  have 4: "fv_ineq (x ·_stp δ) = fv_pairs (F ·_pairs δ) - set X" using 1(2) by auto

  show ?thesis
    using assms(1) Inequality subst_apply_fv_unfold[of _ δ] 1(2) 2 3 4
    unfolding fv_eq_def fv_rcv_def fv_snd_def
    by (metis (no_types) Sup_empty image_empty fv_pairs.simps fv_set.simps
              fv_stp.simps(4) strand_step.simps(20))
qed

lemma fv_strand_subst:
  assumes "P = fv_stp ∨ P = fv_rcv ∨ P = fv_snd ∨ P = fv_eq ac ∨ P = fv_ineq"
  and "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv_set (δ ` (⋃(set (map P S)))) = ⋃(set (map P (S ·_st δ)))"
using assms(2)
proof (induction S)
  case (Cons x S)
  hence *: "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
           "set (bvars_stp x) ∩ (subst_domain δ ∪ range_vars δ) = {}"
    unfolding bvars_st_def by force+
  hence **: "fv_set (δ ` P x) = P (x ·_stp δ)" using fv_strand_step_subst[OF assms(1), of x δ] by auto
  have "fv_set (δ ` (⋃(set (map P (x#S))))) = fv_set (δ ` P x) ∪ (⋃(set (map P ((S ·_st δ)))))"
    using Cons unfolding range_vars_alt_def bvars_st_def by force
  hence "fv_set (δ ` (⋃(set (map P (x#S))))) = P (x ·_stp δ) ∪ fv_set (δ ` (⋃(set (map P S))))"
    using ** by simp
  thus ?case using Cons.IH[OF *(1)] unfolding bvars_st_def by simp
qed simp

lemma fv_strand_subst2:
  assumes "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv_set (δ ` (wfrestrictedvars_st S)) = wfrestrictedvars_st (S ·_st δ)"
by (metis (no_types, lifting) assms fv_set.simps vars_snd_rcv_strand2 fv_strand_subst UN_Un image_Un)

lemma fv_strand_subst':
  assumes "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv_set (δ ` (fv_st S)) = fv_st (S ·_st δ)"
by (metis assms fv_strand_subst fv_st_def)

lemma fv_trms_pairs_is_fv_pairs:
```

```
  "fv_set (trms_pairs F) = fv_pairs F"
by auto

lemma fv_pairs_in_fv_trms_pairs: "x ∈ fv_pairs F ⟹ x ∈ fv_set (trms_pairs F)"
using fv_trms_pairs_is_fv_pairs[of F] by blast

lemma trms_st_append: "trms_st (A@B) = trms_st A ∪ trms_st B"
by auto

lemma trms_pairs_subst: "trms_pairs (a ·pairs ϑ) = trms_pairs a ·set ϑ"
by (auto simp add: subst_apply_pairs_def)

lemma trms_pairs_fv_subst_subset:
  "t ∈ trms_pairs F ⟹ fv (t · ϑ) ⊆ fv_pairs (F ·pairs ϑ)"
by (force simp add: subst_apply_pairs_def)

lemma trms_pairs_fv_subst_subset':
  fixes t::"('a,'b) term" and ϑ::"('a,'b) subst"
  assumes "t ∈ subterms_set (trms_pairs F)"
  shows "fv (t · ϑ) ⊆ fv_pairs (F ·pairs ϑ)"
proof -
  { fix x assume "x ∈ fv t"
    hence "x ∈ fv_pairs F"
      using fv_subset[OF assms] fv_subterms_set[of "trms_pairs F"] fv_trms_pairs_is_fv_pairs[of F]
      by blast
    hence "fv (ϑ x) ⊆ fv_pairs (F ·pairs ϑ)" using fv_pairs_subst_fv_subset by fast
  } thus ?thesis by (meson fv_subst_obtain_var subset_iff)
qed

lemma trms_pairs_funs_term_cases:
  assumes "t ∈ trms_pairs (F ·pairs ϑ)" "f ∈ funs_term t"
  shows "(∃u ∈ trms_pairs F. f ∈ funs_term u) ∨ (∃x ∈ fv_pairs F. f ∈ funs_term (ϑ x))"
using assms(1)
proof (induction F)
  case (Cons g F)
  obtain s u where g: "g = (s,u)" by (metis surj_pair)
  show ?case
  proof (cases "t ∈ trms_pairs (F ·pairs ϑ)")
    case False
    thus ?thesis
      using assms(2) Cons.prems g funs_term_subst[of _ ϑ]
      by (auto simp add: subst_apply_pairs_def)
  qed (use Cons.IH in fastforce)
qed simp

lemma trm_stp_subst:
  assumes "subst_domain ϑ ∩ set (bvars_stp a) = {}"
  shows "trms_stp (a ·stp ϑ) = trms_stp a ·set ϑ"
proof -
  have "rm_vars (set (bvars_stp a)) ϑ = ϑ" using assms by force
  thus ?thesis
    using assms
    by (auto simp add: subst_apply_pairs_def subst_apply_strand_step_def
             split: strand_step.splits)
qed

lemma trms_st_subst:
  assumes "subst_domain ϑ ∩ bvars_st A = {}"
  shows "trms_st (A ·st ϑ) = trms_st A ·set ϑ"
using assms
proof (induction A)
  case (Cons a A)
  have 1: "subst_domain ϑ ∩ bvars_st A = {}" "subst_domain ϑ ∩ set (bvars_stp a) = {}"
```

```
    using Cons.prems by auto
  hence IH: "trms_st A ·_set ϑ = trms_st (A ·_st ϑ)" using Cons.IH by simp

  have "trms_st (a#A) = trms_stp a ∪ trms_st A" by auto
  hence 2: "trms_st (a#A) ·_set ϑ = (trms_stp a ·_set ϑ) ∪ (trms_st A ·_set ϑ)" by (metis image_Un)

  have "trms_st (a#A ·_st ϑ) = (trms_stp (a ·_stp ϑ)) ∪ trms_st (A ·_st ϑ)"
    by (auto simp add: subst_apply_strand_def)
  hence 3: "trms_st (a#A ·_st ϑ) = (trms_stp a ·_set ϑ) ∪ trms_st (A ·_st ϑ)"
    using trm_stp_subst[OF 1(2)] by auto

  show ?case using IH 2 3 by metis
qed (simp add: subst_apply_strand_def)

lemma strand_map_set_subst:
  assumes δ: "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "⋃(set (map trms_stp (S ·_st δ))) = (⋃(set (map trms_stp S))) ·_set δ"
using assms
proof (induction S)
  case (Cons x S)
  hence "bvars_st [x] ∩ subst_domain δ = {}" "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
    unfolding bvars_st_def by force+
  hence *: "subst_domain δ ∩ set (bvars_stp x) = {}"
          "⋃(set (map trms_stp (S ·_st δ))) = ⋃(set (map trms_stp S)) ·_set δ"
    using Cons.IH(1) bvars_st_singleton[of x] by auto
  hence "trms_stp (x ·_stp δ) = (trms_stp x) ·_set δ"
  proof (cases x)
    case (Inequality X F)
    thus ?thesis
      using rm_vars_apply'[of δ "set X"] *
      by (metis (no_types, lifting) image_cong trm_stp_subst)
  qed simp_all
  thus ?case using * subst_all_insert by auto
qed simp

lemma subst_apply_fv_subset_strand_trm:
  assumes P: "P = fv_stp ∨ P = fv_rcv ∨ P = fv_snd ∨ P = fv_eq ac ∨ P = fv_ineq"
  and fv_sub: "fv t ⊆ ⋃(set (map P S)) ∪ V"
  and δ: "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv (t · δ) ⊆ ⋃(set (map P (S ·_st δ))) ∪ fv_set (δ ` V)"
using fv_strand_subst[OF P δ] subst_apply_fv_subset[OF fv_sub, of δ] by force

lemma subst_apply_fv_subset_strand_trm2:
  assumes fv_sub: "fv t ⊆ wfrestrictedvars_st S ∪ V"
  and δ: "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv (t · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ` V)"
using fv_strand_subst2[OF δ] subst_apply_fv_subset[OF fv_sub, of δ] by force

lemma subst_apply_fv_subset_strand:
  assumes P: "P = fv_stp ∨ P = fv_rcv ∨ P = fv_snd ∨ P = fv_eq ac ∨ P = fv_ineq"
  and P_subset: "P x ⊆ ⋃(set (map P S)) ∪ V"
  and δ: "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
        "set (bvars_stp x) ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "P (x ·_stp δ) ⊆ ⋃(set (map P (S ·_st δ))) ∪ fv_set (δ ` V)"
proof (cases x)
  case (Send t)
  hence *: "fv_stp x = fv t" "fv_stp (x ·_stp δ) = fv (t · δ)"
          "fv_rcv x = {}" "fv_rcv (x ·_stp δ) = {}"
          "fv_snd x = fv t" "fv_snd (x ·_stp δ) = fv (t · δ)"
          "fv_eq ac x = {}" "fv_eq ac (x ·_stp δ) = {}"
          "fv_ineq x = {}" "fv_ineq (x ·_stp δ) = {}"
    by auto
  hence **: "(P x = fv t ∧ P (x ·_stp δ) = fv (t · δ)) ∨ (P x = {} ∧ P (x ·_stp δ) = {})" by (metis P)
```

```
  moreover
  { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
  moreover
  { assume "P x = fv t" "P (x ·stp δ) = fv (t · δ)"
    hence "fv t ⊆ ⋃(set (map P S)) ∪ V" using P_subset by auto
    hence "fv (t · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
      unfolding varsst_def using P subst_apply_fv_subset_strand_trm assms by blast
    hence ?thesis using ‹P (x ·stp δ) = fv (t · δ)› by force
  }
  ultimately show ?thesis by metis
next
  case (Receive t)
  hence *: "fvstp x = fv t" "fvstp (x ·stp δ) = fv (t · δ)"
           "fvrcv x = fv t" "fvrcv (x ·stp δ) = fv (t · δ)"
           "fvsnd x = {}" "fvsnd (x ·stp δ) = {}"
           "fveq ac x = {}" "fveq ac (x ·stp δ) = {}"
           "fvineq x = {}" "fvineq (x ·stp δ) = {}"
    by auto
  hence **: "(P x = fv t ∧ P (x ·stp δ) = fv (t · δ)) ∨ (P x = {} ∧ P (x ·stp δ) = {})" by (metis P)
  moreover
  { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
  moreover
  { assume "P x = fv t" "P (x ·stp δ) = fv (t · δ)"
    hence "fv t ⊆ ⋃(set (map P S)) ∪ V" using P_subset by auto
    hence "fv (t · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
      unfolding varsst_def using P subst_apply_fv_subset_strand_trm assms by blast
    hence ?thesis using ‹P (x ·stp δ) = fv (t · δ)› by blast
  }
  ultimately show ?thesis by metis
next
  case (Equality ac' t t') show ?thesis
  proof (cases "ac' = ac")
    case True
    hence *: "fvstp x = fv t ∪ fv t'" "fvstp (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
             "fvrcv x = {}" "fvrcv (x ·stp δ) = {}"
             "fvsnd x = {}" "fvsnd (x ·stp δ) = {}"
             "fveq ac x = fv t ∪ fv t'" "fveq ac (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
             "fvineq x = {}" "fvineq (x ·stp δ) = {}"
      using Equality by auto
    hence **: "(P x = fv t ∪ fv t' ∧ P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ))
              ∨ (P x = {} ∧ P (x ·stp δ) = {})"
      by (metis P)
    moreover
    { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
    moreover
    { assume "P x = fv t ∪ fv t'" "P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
      hence "fv t ⊆ ⋃(set (map P S)) ∪ V" "fv t' ⊆ ⋃(set (map P S)) ∪ V" using P_subset by auto
      hence "fv (t · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
            "fv (t' · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
        unfolding varsst_def using P subst_apply_fv_subset_strand_trm assms by metis+
      hence ?thesis using ‹P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)› by blast
    }
    ultimately show ?thesis by metis
  next
    case False
    hence *: "fvstp x = fv t ∪ fv t'" "fvstp (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
             "fvrcv x = {}" "fvrcv (x ·stp δ) = {}"
             "fvsnd x = {}" "fvsnd (x ·stp δ) = {}"
             "fveq ac x = {}" "fveq ac (x ·stp δ) = {}"
             "fvineq x = {}" "fvineq (x ·stp δ) = {}"
      using Equality by auto
    hence **: "(P x = fv t ∪ fv t' ∧ P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ))
              ∨ (P x = {} ∧ P (x ·stp δ) = {})"
```

```
      by (metis P)
    moreover
    { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
    moreover
    { assume "P x = fv t ∪ fv t'" "P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
      hence "fv t ⊆ ⋃(set (map P S)) ∪ V" "fv t' ⊆ ⋃(set (map P S)) ∪ V" using P_subset by auto
      hence "fv (t · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
            "fv (t' · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
        unfolding varsst_def using P subst_apply_fv_subset_strand_trm assms by metis+
      hence ?thesis using ‹P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)› by blast
    }
    ultimately show ?thesis by metis
  qed
next
  case (Inequality X F)
  hence *: "fvstp x = fvpairs F - set X" "fvstp (x ·stp δ) = fvpairs (F ·pairs δ) - set X"
           "fvrcv x = {}" "fvrcv (x ·stp δ) = {}"
           "fvsnd x = {}" "fvsnd (x ·stp δ) = {}"
           "fveq ac x = {}" "fveq ac (x ·stp δ) = {}"
           "fvineq x = fvpairs F - set X"
           "fvineq (x ·stp δ) = fvpairs (F ·pairs δ) - set X"
    using δ(2) ineq_apply_subst[of δ X F] by force+
  hence **: "(P x = fvpairs F - set X ∧ P (x ·stp δ) = fvpairs (F ·pairs δ) - set X)
            ∨ (P x = {} ∧ P (x ·stp δ) = {})"
    by (metis P)
  moreover
  { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
  moreover
  { assume "P x = fvpairs F - set X" "P (x ·stp δ) = fvpairs (F ·pairs δ) - set X"
    hence "fvpairs F - set X ⊆ ⋃(set (map P S)) ∪ V"
      using P_subset by auto
    hence "fvpairs (F ·pairs δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` (V ∪ set X))"
    proof (induction F)
      case (Cons f G)
      hence IH: "fvpairs (G ·pairs δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` (V ∪ set X))"
        by (metis (no_types, lifting) Diff_subset_conv UN_insert le_sup_iff
                  list.simps(15) fvpairs.simps)
      obtain t t' where f: "f = (t,t')" by (metis surj_pair)
      hence "fv t ⊆ ⋃(set (map P S)) ∪ (V ∪ set X)" "fv t' ⊆ ⋃(set (map P S)) ∪ (V ∪ set X)"
        using Cons.prems by auto
      hence "fv (t · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` (V ∪ set X))"
            "fv (t' · δ) ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` (V ∪ set X))"
        using subst_apply_fv_subset_strand_trm[OF P _ assms(3)]
        by blast+
      thus ?case using f IH by (auto simp add: subst_apply_pairs_def)
    qed (simp add: subst_apply_pairs_def)
    moreover have "fvset (δ ` set X) = set X" using assms(4) Inequality by force
    ultimately have "fvpairs (F ·pairs δ) - set X ⊆ ⋃(set (map P (S ·st δ))) ∪ fvset (δ ` V)"
      by auto
    hence ?thesis using ‹P (x ·stp δ) = fvpairs (F ·pairs δ) - set X› by blast
  }
  ultimately show ?thesis by metis
qed

lemma subst_apply_fv_subset_strand2:
  assumes P: "P = fvstp ∨ P = fvrcv ∨ P = fvsnd ∨ P = fveq ac ∨ P = fvineq ∨ P = fv_req ac"
  and P_subset: "P x ⊆ wfrestrictedvarsst S ∪ V"
  and δ: "bvarsst S ∩ (subst_domain δ ∪ range_vars δ) = {}"
        "set (bvarsstp x) ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "P (x ·stp δ) ⊆ wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` V)"
proof (cases x)
  case (Send t)
  hence *: "fvstp x = fv t" "fvstp (x ·stp δ) = fv (t · δ)"
```

```
          "fv_rcv x = {}" "fv_rcv (x ·_stp δ) = {}"
          "fv_snd x = fv t" "fv_snd (x ·_stp δ) = fv (t · δ)"
          "fv_eq ac x = {}" "fv_eq ac (x ·_stp δ) = {}"
          "fv_ineq x = {}" "fv_ineq (x ·_stp δ) = {}"
          "fv_r_eq ac x = {}" "fv_r_eq ac (x ·_stp δ) = {}"
    by auto
  hence **: "(P x = fv t ∧ P (x ·_stp δ) = fv (t · δ)) ∨ (P x = {} ∧ P (x ·_stp δ) = {})" by (metis P)
  moreover
  { assume "P x = {}" "P (x ·_stp δ) = {}" hence ?thesis by simp }
  moreover
  { assume "P x = fv t" "P (x ·_stp δ) = fv (t · δ)"
    hence "fv t ⊆ wfrestrictedvars_st S ∪ V" using P_subset by auto
    hence "fv (t · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ‘ V)"
      using P subst_apply_fv_subset_strand_trm2 assms by blast
    hence ?thesis using ‹P (x ·_stp δ) = fv (t · δ)› by blast
  }
  ultimately show ?thesis by metis
next
  case (Receive t)
  hence *: "fv_stp x = fv t" "fv_stp (x ·_stp δ) = fv (t · δ)"
          "fv_rcv x = fv t" "fv_rcv (x ·_stp δ) = fv (t · δ)"
          "fv_snd x = {}" "fv_snd (x ·_stp δ) = {}"
          "fv_eq ac x = {}" "fv_eq ac (x ·_stp δ) = {}"
          "fv_ineq x = {}" "fv_ineq (x ·_stp δ) = {}"
          "fv_r_eq ac x = {}" "fv_r_eq ac (x ·_stp δ) = {}"
    by auto
  hence **: "(P x = fv t ∧ P (x ·_stp δ) = fv (t · δ)) ∨ (P x = {} ∧ P (x ·_stp δ) = {})" by (metis P)
  moreover
  { assume "P x = {}" "P (x ·_stp δ) = {}" hence ?thesis by simp }
  moreover
  { assume "P x = fv t" "P (x ·_stp δ) = fv (t · δ)"
    hence "fv t ⊆ wfrestrictedvars_st S ∪ V" using P_subset by auto
    hence "fv (t · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ‘ V)"
      using P subst_apply_fv_subset_strand_trm2 assms by blast
    hence ?thesis using ‹P (x ·_stp δ) = fv (t · δ)› by blast
  }
  ultimately show ?thesis by metis
next
  case (Equality ac’ t t’) show ?thesis
  proof (cases "ac’ = ac")
    case True
    hence *: "fv_stp x = fv t ∪ fv t’" "fv_stp (x ·_stp δ) = fv (t · δ) ∪ fv (t’ · δ)"
            "fv_rcv x = {}" "fv_rcv (x ·_stp δ) = {}"
            "fv_snd x = {}" "fv_snd (x ·_stp δ) = {}"
            "fv_eq ac x = fv t ∪ fv t’" "fv_eq ac (x ·_stp δ) = fv (t · δ) ∪ fv (t’ · δ)"
            "fv_ineq x = {}" "fv_ineq (x ·_stp δ) = {}"
            "fv_r_eq ac x = fv t’" "fv_r_eq ac (x ·_stp δ) = fv (t’ · δ)"
      using Equality by auto
    hence **: "(P x = fv t ∪ fv t’ ∧ P (x ·_stp δ) = fv (t · δ) ∪ fv (t’ · δ))
              ∨ (P x = {} ∧ P (x ·_stp δ) = {})
              ∨ (P x = fv t’ ∧ P (x ·_stp δ) = fv (t’ · δ))"
      by (metis P)
    moreover
    { assume "P x = {}" "P (x ·_stp δ) = {}" hence ?thesis by simp }
    moreover
    { assume "P x = fv t ∪ fv t’" "P (x ·_stp δ) = fv (t · δ) ∪ fv (t’ · δ)"
      hence "fv t ⊆ wfrestrictedvars_st S ∪ V" "fv t’ ⊆ wfrestrictedvars_st S ∪ V" using P_subset by
auto
      hence "fv (t · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ‘ V)"
           "fv (t’ · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ‘ V)"
        using P subst_apply_fv_subset_strand_trm2 assms by blast+
      hence ?thesis using ‹P (x ·_stp δ) = fv (t · δ) ∪ fv (t’ · δ)› by blast
    }
```

```
    moreover
    { assume "P x = fv t'" "P (x ·stp δ) = fv (t' · δ)"
      hence "fv t' ⊆ wfrestrictedvarsst S ∪ V" using P_subset by auto
      hence "fv (t' · δ) ⊆ wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` V)"
        using P subst_apply_fv_subset_strand_trm2 assms by blast+
      hence ?thesis using ‹P (x ·stp δ) = fv (t' · δ)› by blast
    }
    ultimately show ?thesis by metis
  next
    case False
    hence *: "fvstp x = fv t ∪ fv t'" "fvstp (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
             "fvrcv x = {}" "fvrcv (x ·stp δ) = {}"
             "fvsnd x = {}" "fvsnd (x ·stp δ) = {}"
             "fveq ac x = {}" "fveq ac (x ·stp δ) = {}"
             "fvineq x = {}" "fvineq (x ·stp δ) = {}"
             "fv_req ac x = {}" "fv_req ac (x ·stp δ) = {}"
      using Equality by auto
    hence **: "(P x = fv t ∪ fv t' ∧ P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ))
              ∨ (P x = {} ∧ P (x ·stp δ) = {})
              ∨ (P x = fv t' ∧ P (x ·stp δ) = fv (t' · δ))"
      by (metis P)
    moreover
    { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
    moreover
    { assume "P x = fv t ∪ fv t'" "P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)"
      hence "fv t ⊆ wfrestrictedvarsst S ∪ V" "fv t' ⊆ wfrestrictedvarsst S ∪ V"
        using P_subset by auto
      hence "fv (t · δ) ⊆ wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` V)"
            "fv (t' · δ) ⊆ wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` V)"
        using P subst_apply_fv_subset_strand_trm2 assms by blast+
      hence ?thesis using ‹P (x ·stp δ) = fv (t · δ) ∪ fv (t' · δ)› by blast
    }
    moreover
    { assume "P x = fv t'" "P (x ·stp δ) = fv (t' · δ)"
      hence "fv t' ⊆ wfrestrictedvarsst S ∪ V" using P_subset by auto
      hence "fv (t' · δ) ⊆ wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` V)"
        using P subst_apply_fv_subset_strand_trm2 assms by blast+
      hence ?thesis using ‹P (x ·stp δ) = fv (t' · δ)› by blast
    }
    ultimately show ?thesis by metis
  qed
next
  case (Inequality X F)
  hence *: "fvstp x = fvpairs F - set X" "fvstp (x ·stp δ) = fvpairs (F ·pairs δ) - set X"
           "fvrcv x = {}" "fvrcv (x ·stp δ) = {}"
           "fvsnd x = {}" "fvsnd (x ·stp δ) = {}"
           "fveq ac x = {}" "fveq ac (x ·stp δ) = {}"
           "fvineq x = fvpairs F - set X" "fvineq (x ·stp δ) = fvpairs (F ·pairs δ) - set X"
           "fv_req ac x = {}" "fv_req ac (x ·stp δ) = {}"
    using δ(2) ineq_apply_subst[of δ X F] by force+
  hence **: "(P x = fvpairs F - set X ∧ P (x ·stp δ) = fvpairs (F ·pairs δ) - set X)
            ∨ (P x = {} ∧ P (x ·stp δ) = {})"
    by (metis P)
  moreover
  { assume "P x = {}" "P (x ·stp δ) = {}" hence ?thesis by simp }
  moreover
  { assume "P x = fvpairs F - set X" "P (x ·stp δ) = fvpairs (F ·pairs δ) - set X"
    hence "fvpairs F - set X ⊆ wfrestrictedvarsst S ∪ V" using P_subset by auto
    hence "fvpairs (F ·pairs δ) ⊆ wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` (V ∪ set X))"
    proof (induction F)
      case (Cons f G)
      hence IH: "fvpairs (G ·pairs δ) ⊆wfrestrictedvarsst (S ·st δ) ∪ fvset (δ ` (V ∪ set X))"
        by (metis (no_types, lifting) Diff_subset_conv UN_insert le_sup_iff
```

```
                list.simps(15) fv_pairs.simps)
      obtain t t' where f: "f = (t,t')" by (metis surj_pair)
      hence "fv t ⊆ wfrestrictedvars_st S ∪ (V ∪ set X)" "fv t' ⊆ wfrestrictedvars_st S ∪ (V ∪ set
X)"
        using Cons.prems by auto
      hence "fv (t · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ` (V ∪ set X))"
            "fv (t' · δ) ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ` (V ∪ set X))"
        using subst_apply_fv_subset_strand_trm2[OF _ assms(3)] P
        by blast+
      thus ?case using f IH by (auto simp add: subst_apply_pairs_def)
    qed (simp add: subst_apply_pairs_def)
    moreover have "fv_set (δ ` set X) = set X" using assms(4) Inequality by force
    ultimately have "fv_pairs (F ·_pairs δ) - set X ⊆ wfrestrictedvars_st (S ·_st δ) ∪ fv_set (δ ` V)"
      by fastforce
    hence ?thesis using ‹P (x ·_stp δ) = fv_pairs (F ·_pairs δ) - set X› by blast
  }
  ultimately show ?thesis by metis
qed

lemma strand_subst_fv_bounded_if_img_bounded:
  assumes "range_vars δ ⊆ fv_st S"
  shows "fv_st (S ·_st δ) ⊆ fv_st S"
using subst_sends_strand_fv_to_img[of S δ] assms by blast

lemma strand_fv_subst_subset_if_subst_elim:
  assumes "subst_elim δ v" and "v ∈ fv_st S ∨ bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "v ∉ fv_st (S ·_st δ)"
proof (cases "v ∈ fv_st S")
  case True thus ?thesis
  proof (induction S)
    case (Cons x S)
    have *: "v ∉ fv_stp (x ·_stp δ)"
    using assms(1)
    proof (cases x)
      case (Inequality X F)
      hence "subst_elim (rm_vars (set X) δ) v ∨ v ∈ set X" using assms(1) by blast
      moreover have "fv_stp (Inequality X F ·_stp δ) = fv_pairs (F ·_pairs rm_vars (set X) δ) - set X"
        using Inequality by auto
      ultimately have "v ∉ fv_stp (Inequality X F ·_stp δ)"
        by (induct F) (auto simp add: subst_elim_def subst_apply_pairs_def)
      thus ?thesis using Inequality by simp
    qed (simp_all add: subst_elim_def)
    moreover have "v ∉ fv_st (S ·_st δ)" using Cons.IH
    proof (cases "v ∈ fv_st S")
      case False
      moreover have "v ∉ range_vars δ"
        by (simp add: subst_elimD''[OF assms(1)] range_vars_alt_def)
      ultimately show ?thesis by (meson UnE subsetCE subst_sends_strand_fv_to_img)
    qed simp
    ultimately show ?case by auto
  qed simp
next
  case False
  thus ?thesis
    using assms fv_strand_subst'
    unfolding subst_elim_def
    by (metis (mono_tags, hide_lams) fv_set.simps imageE mem_simps(8) subst_apply_term.simps(1))
qed

lemma strand_fv_subst_subset_if_subst_elim':
  assumes "subst_elim δ v" "v ∈ fv_st S" "range_vars δ ⊆ fv_st S"
  shows "fv_st (S ·_st δ) ⊂ fv_st S"
using strand_fv_subst_subset_if_subst_elim[OF assms(1)] assms(2)
```

```
    strand_subst_fv_bounded_if_img_bounded[OF assms(3)]
by blast

lemma fv_ik_is_fv_rcv: "fv_set (ik_st S) = ⋃(set (map fv_rcv S))"
by (induct S rule: ik_st.induct) auto

lemma fv_ik_subset_fv_st[simp]: "fv_set (ik_st S) ⊆ wfrestrictedvars_st S"
by (induct S rule: ik_st.induct) auto

lemma fv_assignment_rhs_subset_fv_st[simp]: "fv_set (assignment_rhs_st S) ⊆ wfrestrictedvars_st S"
by (induct S rule: assignment_rhs_st.induct) force+

lemma fv_ik_subset_fv_st'[simp]: "fv_set (ik_st S) ⊆ fv_st S"
by (induct S rule: ik_st.induct) auto

lemma ik_st_var_is_fv: "Var x ∈ subterms_set (ik_st A) ⟹ x ∈ fv_st A"
by (meson fv_ik_subset_fv_st'[of A] fv_subset_subterms subsetCE term.set_intros(3))

lemma fv_assignment_rhs_subset_fv_st'[simp]: "fv_set (assignment_rhs_st S) ⊆ fv_st S"
by (induct S rule: assignment_rhs_st.induct) auto

lemma ik_st_assignment_rhs_st_wfrestrictedvars_subset:
  "fv_set (ik_st A ∪ assignment_rhs_st A) ⊆ wfrestrictedvars_st A"
using fv_ik_subset_fv_st[of A] fv_assignment_rhs_subset_fv_st[of A]
by simp+

lemma strand_step_id_subst[iff]: "x ·_stp Var = x" by (cases x) auto

lemma strand_id_subst[iff]: "S ·_st Var = S" using strand_step_id_subst by (induct S) auto

lemma strand_subst_vars_union_bound[simp]: "vars_st (S ·_st δ) ⊆ vars_st S ∪ range_vars δ"
proof (induction S)
  case (Cons x S)
  moreover have "vars_stp (x ·_stp δ) ⊆ vars_stp x ∪ range_vars δ" using subst_sends_fv_to_img[of _ δ]
  proof (cases x)
    case (Inequality X F)
    define δ' where "δ' ≡ rm_vars (set X) δ"
    have 0: "range_vars δ' ⊆ range_vars δ"
      using rm_vars_img[of "set X" δ]
      by (auto simp add: δ'_def subst_domain_def range_vars_alt_def)

    have "vars_stp (x ·_stp δ) = fv_pairs (F ·_pairs δ') ∪ set X" "vars_stp x = fv_pairs F ∪ set X"
      using Inequality by (auto simp add: δ'_def)
    moreover have "fv_pairs (F ·_pairs δ') ⊆ fv_pairs F ∪ range_vars δ"
    proof (induction F)
      case (Cons f G)
      obtain t t' where f: "f = (t,t')" by moura
      hence "fv_pairs (f#G ·_pairs δ') = fv (t · δ') ∪ fv (t' · δ') ∪ fv_pairs (G ·_pairs δ')"
            "fv_pairs (f#G) = fv t ∪ fv t' ∪ fv_pairs G"
        by (auto simp add: subst_apply_pairs_def)
      thus ?case
        using 0 Cons.IH subst_sends_fv_to_img[of t δ'] subst_sends_fv_to_img[of t' δ']
        unfolding f by auto
    qed (simp add: subst_apply_pairs_def)
    ultimately show ?thesis by auto
  qed auto
  ultimately show ?case by auto
qed simp

lemma strand_vars_split:
  "vars_st (S@S') = vars_st S ∪ vars_st S'"
  "wfrestrictedvars_st (S@S') = wfrestrictedvars_st S ∪ wfrestrictedvars_st S'"
  "fv_st (S@S') = fv_st S ∪ fv_st S'"
```

```
by auto

lemma bvars_subst_ident: "bvars_st S = bvars_st (S ·_st δ)"
unfolding bvars_st_def
by (induct S) (simp_all add: subst_apply_strand_step_def split: strand_step.splits)

lemma strand_subst_subst_idem:
  assumes "subst_idem δ" "subst_domain δ ∪ range_vars δ ⊆ fv_st S" "subst_domain ϑ ∩ fv_st S = {}"
          "range_vars δ ∩ bvars_st S = {}" "range_vars ϑ ∩ bvars_st S = {}"
  shows "(S ·_st δ) ·_st ϑ = (S ·_st δ)"
  and   "(S ·_st δ) ·_st (ϑ ∘_s δ) = (S ·_st δ)"
proof -
  from assms(2,3) have "fv_st (S ·_st δ) ∩ subst_domain ϑ = {}"
    using subst_sends_strand_fv_to_img[of S δ] by blast
  thus "(S ·_st δ) ·_st ϑ = (S ·_st δ)" by blast
  thus "(S ·_st δ) ·_st (ϑ ∘_s δ) = (S ·_st δ)"
    by (metis assms(1,4,5) bvars_subst_ident strand_subst_comp subst_idem_def)
qed

lemma strand_subst_img_bound:
  assumes "subst_domain δ ∪ range_vars δ ⊆ fv_st S"
    and "(subst_domain δ ∪ range_vars δ) ∩ bvars_st S = {}"
  shows "range_vars δ ⊆ fv_st (S ·_st δ)"
proof -
  have "subst_domain δ ⊆ ⋃(set (map fv_stp S))" by (metis (no_types) fv_st_def Un_subset_iff assms(1))
  thus ?thesis
    unfolding range_vars_alt_def fv_st_def
    by (metis subst_range.simps fv_set_mono fv_strand_subst Int_commute assms(2) image_Un
             le_iff_sup)
qed

lemma strand_subst_img_bound':
  assumes "subst_domain δ ∪ range_vars δ ⊆ vars_st S"
    and "(subst_domain δ ∪ range_vars δ) ∩ bvars_st S = {}"
  shows "range_vars δ ⊆ vars_st (S ·_st δ)"
proof -
  have "(subst_domain δ ∪ fv_set (δ ` subst_domain δ)) ∩ vars_st S =
        subst_domain δ ∪ fv_set (δ ` subst_domain δ)"
    using assms(1) by (metis inf.absorb_iff1 range_vars_alt_def subst_range.simps)
  hence "range_vars δ ⊆ fv_st (S ·_st δ)"
    using vars_snd_rcv_strand fv_snd_rcv_strand assms(2) strand_subst_img_bound
    unfolding range_vars_alt_def
    by (metis (no_types) inf_le2 inf_sup_distrib1 subst_range.simps sup_bot.right_neutral)
  thus "range_vars δ ⊆ vars_st (S ·_st δ)"
    by (metis fv_snd_rcv_strand le_supI1 vars_snd_rcv_strand)
qed

lemma strand_subst_all_fv_subset:
  assumes "fv t ⊆ fv_st S" "(subst_domain δ ∪ range_vars δ) ∩ bvars_st S = {}"
  shows "fv (t · δ) ⊆ fv_st (S ·_st δ)"
using assms by (metis fv_strand_subst' Int_commute subst_apply_fv_subset)

lemma strand_subst_not_dom_fixed:
  assumes "v ∈ fv_st S" and "v ∉ subst_domain δ"
  shows "v ∈ fv_st (S ·_st δ)"
using assms
proof (induction S)
  case (Cons x S')
  have 1: "⋀X. v ∉ subst_domain (rm_vars (set X) δ)"
    using Cons.prems(2) rm_vars_dom_subset by force

  show ?case
  proof (cases "v ∈ fv_st S'")
```

```
    case True thus ?thesis using Cons.IH[OF _ Cons.prems(2)] by auto
  next
    case False
    hence 2: "v ∈ fv_stp x" using Cons.prems(1) by simp
    hence "v ∈ fv_stp (x ·_stp δ)" using Cons.prems(2) subst_not_dom_fixed
    proof (cases x)
      case (Inequality X F)
      hence "v ∈ fv_pairs F - set X" using 2 by simp
      hence "v ∈ fv_pairs (F ·_pairs rm_vars (set X) δ)"
        using subst_not_dom_fixed[OF _ 1]
        by (induct F) (auto simp add: subst_apply_pairs_def)
      thus ?thesis using Inequality 2 by auto
    qed (force simp add: subst_domain_def)+
    thus ?thesis by auto
  qed
qed simp

lemma strand_vars_unfold: "v ∈ vars_st S ⟹ ∃S' x S''. S = S'@x#S'' ∧ v ∈ vars_stp x"
proof (induction S)
  case (Cons x S) thus ?case
  proof (cases "v ∈ vars_stp x")
    case True thus ?thesis by blast
  next
    case False
    hence "v ∈ vars_st S" using Cons.prems by auto
    thus ?thesis using Cons.IH by (metis append_Cons)
  qed
qed simp

lemma strand_fv_unfold: "v ∈ fv_st S ⟹ ∃S' x S''. S = S'@x#S'' ∧ v ∈ fv_stp x"
proof (induction S)
  case (Cons x S) thus ?case
  proof (cases "v ∈ fv_stp x")
    case True thus ?thesis by blast
  next
    case False
    hence "v ∈ fv_st S" using Cons.prems by auto
    thus ?thesis using Cons.IH by (metis append_Cons)
  qed
qed simp

lemma subterm_if_in_strand_ik:
  "t ∈ ik_st S ⟹ ∃t'. Receive t' ∈ set S ∧ t ⊑ t'"
by (induct S rule: ik_st_induct) auto

lemma fv_subset_if_in_strand_ik:
  "t ∈ ik_st S ⟹ fv t ⊆ ⋃(set (map fv_rcv S))"
proof -
  assume "t ∈ ik_st S"
  then obtain t' where "Receive t' ∈ set S" "t ⊑ t'" by (metis subterm_if_in_strand_ik)
  hence "fv t ⊆ fv t'" by (simp add: subtermeq_vars_subset)
  thus ?thesis using in_strand_fv_subset_rcv[OF ‹Receive t' ∈ set S›] by auto
qed

lemma fv_subset_if_in_strand_ik':
  "t ∈ ik_st S ⟹ fv t ⊆ fv_st S"
using fv_subset_if_in_strand_ik[of t S] fv_snd_rcv_strand_subset(2)[of S] by blast

lemma vars_subset_if_in_strand_ik2:
  "t ∈ ik_st S ⟹ fv t ⊆ wfrestrictedvars_st S"
using fv_subset_if_in_strand_ik[of t S] vars_snd_rcv_strand_subset2(2)[of S] by blast
```

3.1.3 Lemmata: Simple Strands

```
lemma simple_Cons[dest]: "simple (s#S) ⟹ simple S"
unfolding simple_def by auto

lemma simple_split[dest]:
  assumes "simple (S@S')"
  shows "simple S" "simple S'"
using assms unfolding simple_def by auto

lemma simple_append[intro]: "⟦simple S; simple S'⟧ ⟹ simple (S@S')"
unfolding simple_def by auto

lemma simple_append_sym[sym]: "simple (S@S') ⟹ simple (S'@S)" by auto

lemma not_simple_if_snd_fun: "(∃S' S'' f X. S = S'@Send (Fun f X)#S'') ⟹ ¬simple S"
unfolding simple_def by auto

lemma not_list_all_elim: "¬list_all P A ⟹ ∃B x C. A = B@x#C ∧ ¬P x ∧ list_all P B"
proof (induction A rule: List.rev_induct)
  case (snoc a A)
  show ?case
  proof (cases "list_all P A")
    case True
    thus ?thesis using snoc.prems by auto
  next
    case False
    then obtain B x C where "A = B@x#C" "¬P x" "list_all P B" using snoc.IH[OF False] by auto
    thus ?thesis by auto
  qed
qed simp

lemma not_simple_stp_elim:
  assumes "¬simple_stp x"
  shows "(∃f T. x = Send (Fun f T)) ∨
         (∃a t t'. x = Equality a t t') ∨
         (∃X F. x = Inequality X F ∧ ¬(∃ℐ. ineq_model ℐ X F))"
using assms by (cases x) (fastforce elim: simple_stp.elims)+

lemma not_simple_elim:
  assumes "¬simple S"
  shows "(∃A B f T. S = A@Send (Fun f T)#B ∧ simple A) ∨
         (∃A B a t t'. S = A@Equality a t t'#B ∧ simple A) ∨
         (∃A B X F. S = A@Inequality X F#B ∧ ¬(∃ℐ. ineq_model ℐ X F))"
by (metis assms not_list_all_elim not_simple_stp_elim simple_def)

lemma simple_fun_prefix_unique:
  assumes "A = S@Send (Fun f X)#S'" "simple S"
  shows "∀T g Y T'. A = T@Send (Fun g Y)#T' ∧ simple T ⟶ S = T ∧ f = g ∧ X = Y ∧ S' = T'"
proof -
  { fix T g Y T' assume *: "A = T@Send (Fun g Y)#T'" "simple T"
    { assume "length S < length T" hence False using assms *
        by (metis id_take_nth_drop not_simple_if_snd_fun nth_append nth_append_length)
    }
    moreover
    { assume "length S > length T" hence False using assms *
        by (metis id_take_nth_drop not_simple_if_snd_fun nth_append nth_append_length)
    }
    ultimately have "S = T" using assms * by (meson List.append_eq_append_conv linorder_neqE_nat)
  }
  thus ?thesis using assms(1) by blast
qed
```

```
lemma simple_snd_is_var: "⟦Send t ∈ set S; simple S⟧ ⟹ ∃v. t = Var v"
unfolding simple_def
by (metis list_all_append list_all_simps(1) simple_stp.elims(2) split_list_first
        strand_step.distinct(1) strand_step.distinct(5) strand_step.inject(1))
```

3.1.4 Lemmata: Strand Measure

```
lemma measure_st_wellfounded: "wf measure_st" unfolding measure_st_def by simp

lemma strand_size_append[iff]: "size_st (S@S') = size_st S + size_st S'"
by (induct S) (auto simp add: size_st_def)

lemma strand_size_map_fun_lt[simp]:
  "size_st (map Send X) < size (Fun f X)"
  "size_st (map Send X) < size_st [Send (Fun f X)]"
  "size_st (map Send X) < size_st [Receive (Fun f X)]"
by (induct X) (auto simp add: size_st_def)

lemma strand_size_rm_fun_lt[simp]:
  "size_st (S@S') < size_st (S@Send (Fun f X)#S')"
  "size_st (S@S') < size_st (S@Receive (Fun f X)#S')"
by (induct S) (auto simp add: size_st_def)

lemma strand_fv_card_map_fun_eq:
  "card (fv_st (S@Send (Fun f X)#S')) = card (fv_st (S@(map Send X)@S'))"
proof -
  have "fv_st (S@Send (Fun f X)#S') = fv_st (S@(map Send X)@S')" by auto
  thus ?thesis by simp
qed

lemma strand_fv_card_rm_fun_le[simp]: "card (fv_st (S@S')) ≤ card (fv_st (S@Send (Fun f X)#S'))"
by (force intro: card_mono)

lemma strand_fv_card_rm_eq_le[simp]: "card (fv_st (S@S')) ≤ card (fv_st (S@Equality a t t'#S'))"
by (force intro: card_mono)
```

3.1.5 Lemmata: Well-formed Strands

```
lemma wf_prefix[dest]: "wf_st V (S@S') ⟹ wf_st V S"
by (induct S rule: wf_st.induct) auto

lemma wf_vars_mono[simp]: "wf_st V S ⟹ wf_st (V ∪ W) S"
proof (induction S arbitrary: V)
  case (Cons x S) thus ?case
  proof (cases x)
    case (Send t)
    hence "wf_st (V ∪ fv t ∪ W) S" using Cons.prems(1) Cons.IH by simp
    thus ?thesis using Send by (simp add: sup_commute sup_left_commute)
  next
    case (Equality a t t')
    show ?thesis
    proof (cases a)
      case Assign
      hence "wf_st (V ∪ fv t ∪ W) S" "fv t' ⊆ V ∪ W" using Equality Cons.prems(1) Cons.IH by auto
      thus ?thesis using Equality Assign by (simp add: sup_commute sup_left_commute)
    next
      case Check thus ?thesis using Equality Cons by auto
    qed
  qed auto
qed simp

lemma wf_stI[intro]: "wfrestrictedvars_st S ⊆ V ⟹ wf_st V S"
proof (induction S)
```

```
  case (Cons x S) thus ?case
  proof (cases x)
    case (Send t)
    hence "wf_st V S" "V ∪ fv t = V" using Cons by auto
    thus ?thesis using Send by simp
  next
    case (Equality a t t')
    show ?thesis
    proof (cases a)
      case Assign
      hence "wf_st V S" "fv t' ⊆ V" using Equality Cons by auto
      thus ?thesis using wf_vars_mono Equality Assign by simp
    next
      case Check thus ?thesis using Equality Cons by auto
    qed
  qed simp_all
qed simp

lemma wf_stI'[intro]: "⋃(fv_rcv ` set S) ∪ ⋃(fv_r_eq assign ` set S) ⊆ V ⟹ wf_st V S"
proof (induction S)
  case (Cons x S) thus ?case
  proof (cases x)
    case (Equality a t t') thus ?thesis using Cons by (cases a) auto
  qed simp_all
qed simp

lemma wf_append_exec: "wf_st V (S@S') ⟹ wf_st (V ∪ wfvarsoccs_st S) S'"
proof (induction S arbitrary: V)
  case (Cons x S V) thus ?case
  proof (cases x)
    case (Send t)
    hence "wf_st (V ∪ fv t ∪ wfvarsoccs_st S) S'" using Cons.prems Cons.IH by simp
    thus ?thesis using Send by (auto simp add: sup_assoc)
  next
    case (Equality a t t') show ?thesis
    proof (cases a)
      case Assign
      hence "wf_st (V ∪ fv t ∪ wfvarsoccs_st S) S'" using Equality Cons.prems Cons.IH by auto
      thus ?thesis using Equality Assign by (auto simp add: sup_assoc)
    next
      case Check
      hence "wf_st (V ∪ wfvarsoccs_st S) S'" using Equality Cons.prems Cons.IH by auto
      thus ?thesis using Equality Check by (auto simp add: sup_assoc)
    qed
  qed auto
qed simp

lemma wf_append_suffix:
  "wf_st V S ⟹ wfrestrictedvars_st S' ⊆ wfrestrictedvars_st S ∪ V ⟹ wf_st V (S@S')"
proof (induction V S rule: wf_st_induct)
  case (ConsSnd V t S)
  hence *: "wf_st (V ∪ fv t) S" by simp_all
  hence "wfrestrictedvars_st S' ⊆ wfrestrictedvars_st S ∪ (V ∪ fv t)"
    using ConsSnd.prems(2) by fastforce
  thus ?case using ConsSnd.IH * by simp
next
  case (ConsRcv V t S)
  hence *: "fv t ⊆ V" "wf_st V S" by simp_all
  hence "wfrestrictedvars_st S' ⊆ wfrestrictedvars_st S ∪ V"
    using ConsRcv.prems(2) by fastforce
  thus ?case using ConsRcv.IH * by simp
next
  case (ConsEq V t t' S)
```

```
  hence *: "fv t' ⊆ V" "wf_st (V ∪ fv t) S" by simp_all
  moreover have "vars_stp (Equality Assign t t') = fv t ∪ fv t'"
    by simp
  moreover have "wfrestrictedvars_st (Equality Assign t t'#S) = fv t ∪ fv t' ∪ wfrestrictedvars_st S"
    by auto
  ultimately have "wfrestrictedvars_st S' ⊆ wfrestrictedvars_st S ∪ (V ∪ fv t)"
    using ConsEq.prems(2) by blast
  thus ?case using ConsEq.IH * by simp
qed (simp_all add: wf_stI)

lemma wf_append_suffix':
  assumes "wf_st V S"
    and "⋃(fv_rcv ` set S') ∪ ⋃(fv_r_eq assign ` set S') ⊆ wfvarsoccs_st S ∪ V"
  shows "wf_st V (S@S')"
using assms
proof (induction V S rule: wf_st_induct)
  case (ConsSnd V t S)
  hence *: "wf_st (V ∪ fv t) S" by simp_all
  have "wfvarsoccs_st (send⟨t⟩_st#S) = fv t ∪ wfvarsoccs_st S"
    unfolding wfvarsoccs_st_def by simp
  hence "(⋃a∈set S'. fv_rcv a) ∪ (⋃a∈set S'. fv_r_eq assign a) ⊆ wfvarsoccs_st S ∪ (V ∪ fv t)"
    using ConsSnd.prems(2) unfolding wfvarsoccs_st_def by auto
  thus ?case using ConsSnd.IH[OF *] by auto
next
  case (ConsEq V t t' S)
  hence *: "fv t' ⊆ V" "wf_st (V ∪ fv t) S" by simp_all
  have "wfvarsoccs_st (⟨assign: t ≐ t'⟩_st#S) = fv t ∪ wfvarsoccs_st S"
    unfolding wfvarsoccs_st_def by simp
  hence "(⋃a∈set S'. fv_rcv a) ∪ (⋃a∈set S'. fv_r_eq assign a) ⊆ wfvarsoccs_st S ∪ (V ∪ fv t)"
    using ConsEq.prems(2) unfolding wfvarsoccs_st_def by auto
  thus ?case using ConsEq.IH[OF *(2)] *(1) by auto
qed (auto simp add: wf_stI')

lemma wf_send_compose: "wf_st V (S@(map Send X)@S') = wf_st V (S@Send (Fun f X)#S')"
proof (induction S arbitrary: V)
  case Nil thus ?case
  proof (induction X arbitrary: V)
    case (Cons y Y) thus ?case by (simp add: sup_assoc)
  qed simp
next
  case (Cons s S) thus ?case
  proof (cases s)
    case (Equality ac t t') thus ?thesis using Cons by (cases ac) auto
  qed auto
qed

lemma wf_snd_append[iff]: "wf_st V (S@[Send t]) = wf_st V S"
by (induct S rule: wf_st.induct) simp_all

lemma wf_snd_append': "wf_st V S ⟹ wf_st V (Send t#S)"
by simp

lemma wf_rcv_append[dest]: "wf_st V (S@Receive t#S') ⟹ wf_st V (S@S')"
by (induct S rule: wf_st.induct) simp_all

lemma wf_rcv_append'[intro]:
  "⟦wf_st V (S@S'); fv t ⊆ wfrestrictedvars_st S ∪ V⟧ ⟹ wf_st V (S@Receive t#S')"
proof (induction S rule: wf_st_induct)
  case (ConsRcv V t' S)
  hence "wf_st V (S@S')" "fv t ⊆ wfrestrictedvars_st S ∪ V"
    by auto+
  thus ?case using ConsRcv by auto
next
```

```
  case (ConsEq V t' t'' S)
  hence "fv t'' ⊆ V" by simp
  moreover have
      "wfrestrictedvars_st (Equality Assign t' t''#S) = fv t' ∪ fv t'' ∪ wfrestrictedvars_st S"
    by auto
  ultimately have "fv t ⊆ wfrestrictedvars_st S ∪ (V ∪ fv t')"
    using ConsEq.prems(2) by blast
  thus ?case using ConsEq by auto
qed auto

lemma wf_rcv_append''[intro]: "⟦wf_st V S; fv t ⊆ ⋃(set (map fv_snd S))⟧ ⟹ wf_st V (S@[Receive t])"
by (induct S)
   (simp, metis vars_snd_rcv_strand_subset2(1) append_Nil2 le_supI1 order_trans wf_rcv_append')

lemma wf_rcv_append'''[intro]: "⟦wf_st V S; fv t ⊆ wfrestrictedvars_st S ∪ V⟧ ⟹ wf_st V (S@[Receive
t])"
by (simp add: wf_rcv_append'[of _ _ "[]"])

lemma wf_eq_append[dest]: "wf_st V (S@Equality a t t'#S') ⟹ fv t ⊆ wfrestrictedvars_st S ∪ V ⟹
wf_st V (S@S')"
proof (induction S rule: wf_st_induct)
  case (Nil V)
  hence "wf_st (V ∪ fv t) S'" by (cases a) auto
  moreover have "V ∪ fv t = V" using Nil by auto
  ultimately show ?case by simp
next
  case (ConsRcv V u S)
  hence "wf_st V (S @ Equality a t t' # S')" "fv t ⊆ wfrestrictedvars_st S ∪ V" "fv u ⊆ V"
    by fastforce+
  hence "wf_st V (S@S')" using ConsRcv.IH by auto
  thus ?case using ‹fv u ⊆ V› by simp
next
  case (ConsEq V u u' S)
  hence "wf_st (V ∪ fv u) (S@Equality a t t'#S')" "fv t ⊆ wfrestrictedvars_st S ∪ (V ∪ fv u)" "fv u'
⊆ V"
    by auto
  hence "wf_st (V ∪ fv u) (S@S')" using ConsEq.IH by auto
  thus ?case using ‹fv u' ⊆ V› by simp
qed auto

lemma wf_eq_append'[intro]:
  "⟦wf_st V (S@S'); fv t' ⊆ wfrestrictedvars_st S ∪ V⟧ ⟹ wf_st V (S@Equality a t t'#S')"
proof (induction S rule: wf_st_induct)
  case Nil thus ?case by (cases a) auto
next
  case (ConsEq V u u' S)
  hence "wf_st (V ∪ fv u) (S@S')" "fv t' ⊆ wfrestrictedvars_st S ∪ V ∪ fv u"
    by fastforce+
  thus ?case using ConsEq by auto
next
  case (ConsEq2 V u u' S)
  hence "wf_st V (S@S')" by auto
  thus ?case using ConsEq2 by auto
next
  case (ConsRcv V u S)
  hence "wf_st V (S@S')" "fv t' ⊆ wfrestrictedvars_st S ∪ V"
    by fastforce+
  thus ?case using ConsRcv by auto
next
  case (ConsSnd V u S)
  hence "wf_st (V ∪ fv u) (S@S')" "fv t' ⊆ wfrestrictedvars_st S ∪ (V ∪ fv u)"
    by fastforce+
  thus ?case using ConsSnd by auto
```

```
qed auto

lemma wf_eq_append''[intro]:
  "⟦wf_st V (S@S'); fv t' ⊆ wfvarsoccs_st S ∪ V⟧ ⟹ wf_st V (S@[Equality a t t']@S')"
proof (induction S rule: wf_st_induct)
  case Nil thus ?case by (cases a) auto
next
  case (ConsEq V u u' S)
  hence "wf_st (V ∪ fv u) (S@S')" "fv t' ⊆ wfvarsoccs_st S ∪ V ∪ fv u" by fastforce+
  thus ?case using ConsEq by auto
next
  case (ConsEq2 V u u' S)
  hence "wf_st (V ∪ fv u) (S@S')" "fv t' ⊆ wfvarsoccs_st S ∪ V ∪ fv u" by fastforce+
  thus ?case using ConsEq2 by auto
next
  case (ConsRcv V u S)
  hence "wf_st V (S@S')" "fv t' ⊆ wfvarsoccs_st S ∪ V" by fastforce+
  thus ?case using ConsRcv by auto
next
  case (ConsSnd V u S)
  hence "wf_st (V ∪ fv u) (S@S')" "fv t' ⊆ wfvarsoccs_st S ∪ (V ∪ fv u)" by auto
  thus ?case using ConsSnd by auto
qed auto

lemma wf_eq_append'''[intro]:
  "⟦wf_st V S; fv t' ⊆ wfrestrictedvars_st S ∪ V⟧ ⟹ wf_st V (S@[Equality a t t'])"
by (simp add: wf_eq_append'[of _ _ "[]"])

lemma wf_eq_check_append[dest]: "wf_st V (S@Equality Check t t'#S') ⟹ wf_st V (S@S')"
by (induct S rule: wf_st.induct) simp_all

lemma wf_eq_check_append'[intro]: "wf_st V (S@S') ⟹ wf_st V (S@Equality Check t t'#S')"
by (induct S rule: wf_st.induct) auto

lemma wf_eq_check_append''[intro]: "wf_st V S ⟹ wf_st V (S@[Equality Check t t'])"
by (induct S rule: wf_st.induct) auto

lemma wf_ineq_append[dest]: "wf_st V (S@Inequality X F#S') ⟹ wf_st V (S@S')"
by (induct S rule: wf_st.induct) simp_all

lemma wf_ineq_append'[intro]: "wf_st V (S@S') ⟹ wf_st V (S@Inequality X F#S')"
by (induct S rule: wf_st.induct) auto

lemma wf_ineq_append''[intro]: "wf_st V S ⟹ wf_st V (S@[Inequality X F])"
by (induct S rule: wf_st.induct) auto

lemma wf_rcv_fv_single[elim]: "wf_st V (Receive t#S') ⟹ fv t ⊆ V"
by simp

lemma wf_rcv_fv: "wf_st V (S@Receive t#S') ⟹ fv t ⊆ wfvarsoccs_st S ∪ V"
by (induct S arbitrary: V) (auto split!: strand_step.split poscheckvariant.split)

lemma wf_eq_fv: "wf_st V (S@Equality Assign t t'#S') ⟹ fv t' ⊆ wfvarsoccs_st S ∪ V"
by (induct S arbitrary: V) (auto split!: strand_step.split poscheckvariant.split)

lemma wf_simple_fv_occurrence:
  assumes "wf_st {} S" "simple S" "v ∈ wfrestrictedvars_st S"
  shows "∃S_pre S_suf. S = S_pre@Send (Var v)#S_suf ∧ v ∉ wfrestrictedvars_st S_pre"
using assms
proof (induction S rule: List.rev_induct)
  case (snoc x S)
  from ⟨wf_st {} (S@[x])⟩ have "wf_st {} S" "wf_st (wfrestrictedvars_st S) [x]"
    using wf_append_exec[THEN wf_vars_mono, of "{}" S "[x]" "wfrestrictedvars_st S - wfvarsoccs_st S"]
```

```
        vars_snd_rcv_strand_subset2(4)[of S]
        Diff_partition[of "wfvarsoccs_st S" "wfrestrictedvars_st S"]
    by auto
  from ‹simple (S@[x])› have "simple S" "simple_stp x" unfolding simple_def by auto

  show ?case
  proof (cases "v ∈ wfrestrictedvars_st S")
    case False
    show ?thesis
    proof (cases x)
      case (Receive t)
      hence "fv t ⊆ wfrestrictedvars_st S" using ‹wf_st (wfrestrictedvars_st S) [x]› by simp
      hence "v ∈ wfrestrictedvars_st S"
        using ‹v ∈ wfrestrictedvars_st (S@[x])› ‹x = Receive t›
        by auto
      thus ?thesis using ‹x = Receive t› snoc.IH[OF ‹wf_st {} S› ‹simple S›] by fastforce
    next
      case (Send t)
      hence "v ∈ vars_stp x" using ‹v ∈ wfrestrictedvars_st (S@[x])› False by auto
      from Send obtain w where "t = Var w" using ‹simple_stp x› by (cases t) simp_all
      hence "v = w" using ‹x = Send t› ‹v ∈ vars_stp x› by simp
      thus ?thesis using ‹x = Send t› ‹v ∉ wfrestrictedvars_st S› ‹t = Var w› by auto
    next
      case (Equality ac t t') thus ?thesis using snoc.prems(2) unfolding simple_def by auto
    next
      case (Inequality t t') thus ?thesis using False snoc.prems(3) by auto
    qed
  qed (use snoc.IH[OF ‹wf_st {} S› ‹simple S›] in fastforce)
qed simp

lemma Unifier_strand_fv_subset:
  assumes g_in_ik: "t ∈ ik_st S"
  and δ: "Unifier δ (Fun f X) t"
  and disj: "bvars_st S ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "fv (Fun f X · δ) ⊆ ⋃(set (map fv_rcv (S ·_st δ)))"
by (metis (no_types) fv_subset_if_in_strand_ik[OF g_in_ik]
        disj δ fv_strand_subst subst_apply_fv_subset)

lemma wf_st_induct'[consumes 1, case_names Nil ConsSnd ConsRcv ConsEq ConsEq2 ConsIneq]:
  fixes S::"('a,'b) strand"
  assumes "wf_st V S"
        "P []"
        "⋀t S. ⟦wf_st V S; P S⟧ ⟹ P (S@[Send t])"
        "⋀t S. ⟦wf_st V S; P S; fv t ⊆ V ∪ wfvarsoccs_st S⟧ ⟹ P (S@[Receive t])"
        "⋀t t' S. ⟦wf_st V S; P S; fv t' ⊆ V ∪ wfvarsoccs_st S⟧ ⟹ P (S@[Equality Assign t t'])"
        "⋀t t' S. ⟦wf_st V S; P S⟧ ⟹ P (S@[Equality Check t t'])"
        "⋀X F S. ⟦wf_st V S; P S⟧ ⟹ P (S@[Inequality X F])"
  shows "P S"
using assms
proof (induction S rule: List.rev_induct)
  case (snoc x S)
  hence *: "wf_st V S" "wf_st (V ∪ wfvarsoccs_st S) [x]" by (metis wf_prefix, metis wf_append_exec)
  have IH: "P S" using snoc.IH[OF *(1)] snoc.prems by auto
  note ** = snoc.prems(3,4,5,6,7)[OF *(1) IH] *(2)
  show ?case using **(1,2,4,5,6)
  proof (cases x)
    case (Equality ac t t')
    then show ?thesis using **(3,4,6) by (cases ac) auto
  qed auto
qed simp

lemma wf_subst_apply:
  "wf_st V S ⟹ wf_st (fv_set (δ ` V)) (S ·_st δ)"
```

```
proof (induction S arbitrary: V rule: wf_st_induct)
  case (ConsRcv V t S)
  hence "wf_st V S" "fv t ⊆ V" by simp_all
  hence "wf_st (fv_set (δ ` V)) (S ·_st δ)" "fv (t · δ) ⊆ fv_set (δ ` V)"
    using ConsRcv.IH subst_apply_fv_subset by simp_all
  thus ?case by simp
next
  case (ConsSnd V t S)
  hence "wf_st (V ∪ fv t) S" by simp
  hence "wf_st (fv_set (δ ` (V ∪ fv t))) (S ·_st δ)" using ConsSnd.IH by metis
  hence "wf_st (fv_set (δ ` V) ∪ fv (t · δ)) (S ·_st δ)" using subst_apply_fv_union by metis
  thus ?case by simp
next
  case (ConsEq V t t' S)
  hence "wf_st (V ∪ fv t) S" "fv t' ⊆ V" by auto
  hence "wf_st (fv_set (δ ` (V ∪ fv t))) (S ·_st δ)" and *: "fv (t' · δ) ⊆ fv_set (δ ` V)"
    using ConsEq.IH subst_apply_fv_subset by force+
  hence "wf_st (fv_set (δ ` V) ∪ fv (t · δ)) (S ·_st δ)" using subst_apply_fv_union by metis
  thus ?case using * by simp
qed simp_all

lemma wf_unify:
  assumes wf: "wf_st V (S@Send (Fun f X)#S')"
  and g_in_ik: "t ∈ ik_st S"
  and δ: "Unifier δ (Fun f X) t"
  and disj: "bvars_st (S@Send (Fun f X)#S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "wf_st (fv_set (δ ` V)) ((S@S') ·_st δ)"
using assms
proof (induction S' arbitrary: V rule: List.rev_induct)
  case (snoc x S' V)
  have fun_fv_bound: "fv (Fun f X · δ) ⊆ ⋃(set (map fv_rcv (S ·_st δ)))"
    using snoc.prems(4) bvars_st_split Unifier_strand_fv_subset[OF g_in_ik δ] by auto
  hence "fv (Fun f X · δ) ⊆ fv_set (ik_st (S ·_st δ))" using fv_ik_is_fv_rcv by metis
  hence "fv (Fun f X · δ) ⊆ wfrestrictedvars_st (S ·_st δ)" using fv_ik_subset_fv_st[of "S ·_st δ"] by
blast
  hence *: "fv ((Fun f X) · δ) ⊆ wfrestrictedvars_st ((S@S') ·_st δ)" by fastforce

  from snoc.prems(1) have "wf_st V (S@Send (Fun f X)#S')"
    using wf_prefix[of V "S@Send (Fun f X)#S'" "[x]"] by simp
  hence **: "wf_st (fv_set (δ ` V)) ((S@S') ·_st δ)"
    using snoc.IH[OF _ snoc.prems(2,3)] snoc.prems(4) by auto

  from snoc.prems(1) have ***: "wf_st (V ∪ wfvarsoccs_st (S@Send (Fun f X)#S')) [x]"
    using wf_append_exec[of V "(S@Send (Fun f X)#S')" "[x]"] by simp

  from snoc.prems(4) have disj':
      "bvars_st (S@S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
      "set (bvars_stp x) ∩ (subst_domain δ ∪ range_vars δ) = {}"
    by auto

  show ?case
  proof (cases x)
    case (Send t)
    thus ?thesis using wf_snd_append[of "fv_set (δ ` V)" "(S@S') ·_st δ"] ** by auto
  next
    case (Receive t)
    hence "fv_stp x ⊆ V ∪ wfvarsoccs_st (S@Send (Fun f X)#S')" using *** by auto
    hence "fv_stp x ⊆ V ∪ wfrestrictedvars_st (S@Send (Fun f X)#S')"
      using vars_snd_rcv_strand_subset2(4)[of "S@Send (Fun f X)#S'"] by blast
    hence "fv_stp x ⊆ V ∪ fv (Fun f X) ∪ wfrestrictedvars_st (S@S')" by auto
    hence "fv_stp (x ·_stp δ) ⊆ fv_set (δ ` V) ∪ fv ((Fun f X) · δ) ∪ wfrestrictedvars_st ((S@S') ·_st δ)"
      by (metis (no_types) inf_sup_aci(5) subst_apply_fv_subset_strand2 subst_apply_fv_union disj')
    hence "fv_stp (x ·_stp δ) ⊆ fv_set (δ ` V) ∪ wfrestrictedvars_st ((S@S') ·_st δ)" using * by blast
```

```
    hence "fv (t · δ) ⊆ wfrestrictedvars_st ((S@S') ·_st δ) ∪ fv_set (δ ` V) " using ‹x = Receive t› by
auto
    hence "wf_st (fv_set (δ ` V)) (((S@S') ·_st δ)@[Receive (t · δ)])"
      using wf_rcv_append'''[OF **, of "t · δ"] by metis
    thus ?thesis using ‹x = Receive t› by auto
  next
    case (Equality ac s s') show ?thesis
    proof (cases ac)
      case Assign
      hence "fv s' ⊆ V ∪ wfvarsoccs_st (S@Send (Fun f X)#S')" using Equality *** by auto
      hence "fv s' ⊆ V ∪ wfrestrictedvars_st (S@Send (Fun f X)#S')"
        using vars_snd_rcv_strand_subset2(4)[of "S@Send (Fun f X)#S'"] by blast
      hence "fv s' ⊆ V ∪ fv (Fun f X) ∪ wfrestrictedvars_st (S@S')" by auto
      moreover have "fv s' = fv_r_eq ac x" "fv (s' · δ) = fv_r_eq ac (x ·_stp δ)"
        using Equality by simp_all
      ultimately have "fv (s' · δ) ⊆ fv_set (δ ` V) ∪ fv (Fun f X · δ) ∪ wfrestrictedvars_st ((S@S') ·_st
δ)"
        using subst_apply_fv_subset_strand2[of "fv_eq ac" ac x]
        by (metis disj'(1) subst_apply_fv_subset_strand_trm2 subst_apply_fv_union sup_commute)
      hence "fv (s' · δ) ⊆ fv_set (δ ` V) ∪ wfrestrictedvars_st ((S@S') ·_st δ)" using * by blast
      hence "fv (s' · δ) ⊆ wfrestrictedvars_st ((S@S') ·_st δ) ∪ fv_set (δ ` V)"
        using ‹x = Equality ac s s'› by auto
      hence "wf_st (fv_set (δ ` V)) (((S@S') ·_st δ)@[Equality ac (s · δ) (s' · δ)])"
        using wf_eq_append'''[OF **] by metis
      thus ?thesis using ‹x = Equality ac s s'› by auto
    next
      case Check thus ?thesis using wf_eq_check_append''[OF **] Equality by simp
    qed
  next
    case (Inequality t t') thus ?thesis using wf_ineq_append''[OF **] by simp
  qed
qed (auto dest: wf_subst_apply)

lemma wf_equality:
  assumes wf: "wf_st V (S@Equality ac t t'#S')"
  and δ: "mgu t t' = Some δ"
  and disj: "bvars_st (S@Equality ac t t'#S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "wf_st (fv_set (δ ` V)) ((S@S') ·_st δ)"
using assms
proof (induction S' arbitrary: V rule: List.rev_induct)
  case Nil thus ?case using wf_prefix[of V S "[Equality ac t t']"] wf_subst_apply[of V S δ] by auto
next
  case (snoc x S' V) show ?case
  proof (cases ac)
    case Assign
    hence "fv t' ⊆ V ∪ wfvarsoccs_st S"
      using wf_eq_fv[of V, of S t t' "S'@[x]"] snoc by auto
    hence "fv t' ⊆ V ∪ wfrestrictedvars_st S"
      using vars_snd_rcv_strand_subset2(4)[of S] by blast
    hence "fv t' ⊆ V ∪ wfrestrictedvars_st (S@S')" by force
    moreover have disj':
        "bvars_st (S@S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
        "set (bvars_stp x) ∩ (subst_domain δ ∪ range_vars δ) = {}"
        "bvars_st (S@Equality ac t t'#S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
      using snoc.prems(3) by auto
    ultimately have
        "fv (t' · δ) ⊆ fv_set (δ ` V) ∪ wfrestrictedvars_st ((S@S') ·_st δ)"
      by (metis inf_sup_aci(5) subst_apply_fv_subset_strand_trm2)
    moreover have "fv (t · δ) = fv (t' · δ)"
      by (metis MGU_is_Unifier[OF mgu_gives_MGU[OF δ]])
    ultimately have *:
        "fv (t · δ) ∪ fv (t' · δ) ⊆ fv_set (δ ` V) ∪ wfrestrictedvars_st ((S@S') ·_st δ)"
      by simp
```

```
  from snoc.prems(1) have "wf_st V (S@Equality ac t t'#S')"
    using wf_prefix[of V "S@Equality ac t t'#S'"] by simp
  hence **: "wf_st (fv_set (δ ` V)) ((S@S') ·_st δ)" by (metis snoc.IH δ disj'(3))

  from snoc.prems(1) have ***: "wf_st (V ∪ wfvarsoccs_st (S@Equality ac t t'#S')) [x]"
    using wf_append_exec[of V "(S@Equality ac t t'#S')" "[x]"] by simp

  show ?thesis
  proof (cases x)
    case (Send t)
    thus ?thesis using wf_snd_append[of "fv_set (δ ` V)" "(S@S') ·_st δ"] ** by auto
  next
    case (Receive s)
    hence "fv_stp x ⊆ V ∪ wfvarsoccs_st (S@Equality ac t t'#S')" using *** by auto
    hence "fv_stp x ⊆ V ∪ wfrestrictedvars_st (S@Equality ac t t'#S')"
      using vars_snd_rcv_strand_subset2(4)[of "S@Equality ac t t'#S'"] by blast
    hence "fv_stp x ⊆ V ∪ fv t ∪ fv t' ∪ wfrestrictedvars_st (S@S')"
      by (cases ac) auto
    hence "fv_stp (x ·_stp δ) ⊆ fv_set (δ ` V) ∪ fv (t · δ) ∪ fv (t' · δ) ∪ wfrestrictedvars_st ((S@S')
·_st δ)"
      using subst_apply_fv_subset_strand2[of fv_stp]
      by (metis (no_types) inf_sup_aci(5) subst_apply_fv_union disj'(1,2))
    hence "fv_stp (x ·_stp δ) ⊆ fv_set (δ ` V) ∪ wfrestrictedvars_st ((S@S') ·_st δ)"
      when "ac = Assign"
      using * that by blast
    hence "fv (s · δ) ⊆ wfrestrictedvars_st ((S@S') ·_st δ) ∪ (fv_set (δ ` V))"
      when "ac = Assign"
      using ‹x = Receive s› that by auto
    hence "wf_st (fv_set (δ ` V)) (((S@S') ·_st δ)@[Receive (s · δ)])"
      when "ac = Assign"
      using wf_rcv_append'''[OF **, of "s · δ"] that by metis
    thus ?thesis using ‹x = Receive s› Assign by auto
  next
    case (Equality ac' s s') show ?thesis
    proof (cases ac')
      case Assign
      hence "fv s' ⊆ V ∪ wfvarsoccs_st (S@Equality ac t t'#S')" using *** Equality by auto
      hence "fv s' ⊆ V ∪ wfrestrictedvars_st (S@Equality ac t t'#S')"
        using vars_snd_rcv_strand_subset2(4)[of "S@Equality ac t t'#S'"] by blast
      hence "fv s' ⊆ V ∪ fv t ∪ fv t' ∪ wfrestrictedvars_st (S@S')"
        by (cases ac) auto
      moreover have "fv s' = fv_r_eq ac' x" "fv (s' · δ) = fv_r_eq ac' (x ·_stp δ)"
        using Equality by simp_all
      ultimately have
          "fv (s' · δ) ⊆ fv_set (δ ` V) ∪ fv (t · δ) ∪ fv (t' · δ) ∪ wfrestrictedvars_st ((S@S') ·_st
δ)"
        using subst_apply_fv_subset_strand2[of "fv_r_eq ac'" ac' x]
        by (metis disj'(1) subst_apply_fv_subset_strand_trm2 subst_apply_fv_union sup_commute)
      hence "fv (s' · δ) ⊆ fv_set (δ ` V) ∪ wfrestrictedvars_st ((S@S') ·_st δ)"
        using * ‹ac = Assign› by blast
      hence ****:
          "fv (s' · δ) ⊆ wfrestrictedvars_st ((S@S') ·_st δ) ∪ fv_set (δ ` V)"
        using ‹x = Equality ac' s s'› ‹ac = Assign› by auto
      thus ?thesis
        using ‹x = Equality ac' s s'› ** **** wf_eq_append' ‹ac = Assign›
        by (metis (no_types, lifting) append.assoc append_Nil2 strand_step.case(3)
              strand_subst_hom subst_apply_strand_step_def)
    next
      case Check thus ?thesis using wf_eq_check_append''[OF **] Equality by simp
    qed
  next
    case (Inequality s s') thus ?thesis using wf_ineq_append''[OF **] by simp
```

```
    qed
  qed (metis snoc.prems(1) wf_eq_check_append wf_subst_apply)
qed

lemma wf_rcv_prefix_ground:
  "wf_st {} ((map Receive M)@S) ⟹ vars_st (map Receive M) = {}"
by (induct M) auto

lemma simple_wfvarsoccs_st_is_fv_snd:
  assumes "simple S"
  shows "wfvarsoccs_st S = ⋃(set (map fv_snd S))"
using assms unfolding simple_def
proof (induction S)
  case (Cons x S) thus ?case by (cases x) auto
qed simp

lemma wf_st_simple_induct[consumes 2, case_names Nil ConsSnd ConsRcv ConsIneq]:
  fixes S::"('a,'b) strand"
  assumes "wf_st V S" "simple S"
          "P []"
          "⋀v S. ⟦wf_st V S; simple S; P S⟧ ⟹ P (S@[Send (Var v)])"
          "⋀t S. ⟦wf_st V S; simple S; P S; fv t ⊆ V ∪ ⋃(set (map fv_snd S))⟧ ⟹ P (S@[Receive t])"
          "⋀X F S. ⟦wf_st V S; simple S; P S⟧ ⟹ P (S@[Inequality X F])"
  shows "P S"
using assms
proof (induction S rule: wf_st_induct')
  case (ConsSnd t S)
  hence "P S" by auto
  obtain v where "t = Var v" using simple_snd_is_var[OF _ ‹simple (S@[Send t])›] by auto
  thus ?case using ConsSnd.prems(3)[OF ‹wf_st V S› _ ‹P S›] ‹simple (S@[Send t])› by auto
next
  case (ConsRcv t S) thus ?case using simple_wfvarsoccs_st_is_fv_snd[of "S@[Receive t]"] by auto
qed (auto simp add: simple_def)

lemma wf_trm_stp_dom_fv_disjoint:
  "⟦wf_constr S ϑ; t ∈ trms_st S⟧ ⟹ subst_domain ϑ ∩ fv t = {}"
unfolding wf_constr_def by force

lemma wf_constr_bvars_disj: "wf_constr S ϑ ⟹ (subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st S = {}"
unfolding range_vars_alt_def wf_constr_def by fastforce

lemma wf_constr_bvars_disj':
  assumes "wf_constr S ϑ" "subst_domain δ ∪ range_vars δ ⊆ fv_st S"
  shows "(subst_domain δ ∪ range_vars δ) ∩ bvars_st S = {}" (is ?A)
  and "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st (S ·_st δ) = {}" (is ?B)
proof -
  have "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st S = {}" "fv_st S ∩ bvars_st S = {}"
    using assms(1) unfolding range_vars_alt_def wf_constr_def by fastforce+
  thus ?A and ?B using assms(2) bvars_subst_ident[of S δ] by blast+
qed

lemma (in intruder_model) wf_simple_strand_first_Send_var_split:
  assumes "wf_st {} S" "simple S" "∃v ∈ wfrestrictedvars_st S. t · ℐ = ℐ v"
  shows "∃v S_pre S_suf. S = S_pre@Send (Var v)#S_suf ∧ t · ℐ = ℐ v
                  ∧ ¬(∃w ∈ wfrestrictedvars_st S_pre. t · ℐ = ℐ w)"
    (is "?P S")
using assms
proof (induction S rule: wf_st_simple_induct)
  case (ConsSnd v S) show ?case
  proof (cases "∃w ∈ wfrestrictedvars_st S. t · ℐ = ℐ w")
    case True thus ?thesis using ConsSnd.IH by fastforce
  next
    case False thus ?thesis using ConsSnd.prems by auto
```

```
  qed
next
  case (ConsRcv t' S)
  have "fv t' ⊆ wfrestrictedvars_st S" using ConsRcv.hyps(3) vars_snd_rcv_strand_subset2(1) by force
  hence "∃v ∈ wfrestrictedvars_st S. t · I = I v"
    using ConsRcv.prems(1) by fastforce
  hence "?P S" by (metis ConsRcv.IH)
  thus ?case by fastforce
next
  case (ConsIneq X F S)
  moreover have "wfrestrictedvars_st (S @ [Inequality X F]) = wfrestrictedvars_st S" by auto
  ultimately have "?P S" by blast
  thus ?case by fastforce
qed simp

lemma (in intruder_model) wf_strand_first_Send_var_split:
  assumes "wf_st {} S" "∃v ∈ wfrestrictedvars_st S. t · I ⊑ I v"
  shows "∃S_pre S_suf. ¬(∃w ∈ wfrestrictedvars_st S_pre. t · I ⊑ I w)
          ∧ ((∃t'. S = S_pre@Send t'#S_suf ∧ t · I ⊑ t' · I)
            ∨ (∃t' t''. S = S_pre@Equality Assign t' t''#S_suf ∧ t · I ⊑ t' · I))"
    (is "∃S_pre S_suf. ?P S_pre ∧ ?Q S S_pre S_suf")
using assms
proof (induction S rule: wf_st_induct')
  case (ConsSnd t' S) show ?case
  proof (cases "∃w ∈ wfrestrictedvars_st S. t · I ⊑ I w")
    case True
    then obtain S_pre S_suf where "?P S_pre" "?Q S S_pre S_suf"
      using ConsSnd.IH by moura
    thus ?thesis by fastforce
  next
    case False
    then obtain v where v: "v ∈ fv t'" "t · I ⊑ I v"
      using ConsSnd.prems by auto
    hence "t · I ⊑ t' · I"
      using subst_mono[of "Var v" t' I] vars_iff_subtermeq[of v t'] term.order_trans
      by auto
    thus ?thesis using False v by auto
  qed
next
  case (ConsRcv t' S)
  have "fv t' ⊆ wfrestrictedvars_st S"
    using ConsRcv.hyps vars_snd_rcv_strand_subset2(4)[of S] by blast
  hence "∃v ∈ wfrestrictedvars_st S. t · I ⊑ I v"
    using ConsRcv.prems by fastforce
  then obtain S_pre S_suf where "?P S_pre" "?Q S S_pre S_suf"
    using ConsRcv.IH by moura
  thus ?case by fastforce
next
  case (ConsEq s s' S)
  have *: "fv s' ⊆ wfrestrictedvars_st S"
    using ConsEq.hyps vars_snd_rcv_strand_subset2(4)[of S]
    by blast
  show ?case
  proof (cases "∃v ∈ wfrestrictedvars_st S. t · I ⊑ I v")
    case True
    then obtain S_pre S_suf where "?P S_pre" "?Q S S_pre S_suf"
      using ConsEq.IH by moura
    thus ?thesis by fastforce
  next
    case False
    then obtain v where "v ∈ fv s" "t · I ⊑ I v" using ConsEq.prems * by auto
    hence "t · I ⊑ s · I"
      using vars_iff_subtermeq[of v s] subst_mono[of "Var v" s I] term.order_trans
```

```
      by auto
    thus ?thesis using False by fastforce
  qed
next
  case (ConsEq2 s s' S)
  have "wfrestrictedvars_st (S@[Equality Check s s']) = wfrestrictedvars_st S" by auto
  hence "∃v ∈ wfrestrictedvars_st S. t · I ⊑ I v" using ConsEq2.prems by metis
  then obtain S_pre S_suf where "?P S_pre" "?Q S S_pre S_suf"
    using ConsEq2.IH by moura
  thus ?case by fastforce
next
  case (ConsIneq X F S)
  hence "∃v ∈ wfrestrictedvars_st S. t · I ⊑ I v" by fastforce
  then obtain S_pre S_suf where "?P S_pre" "?Q S S_pre S_suf"
    using ConsIneq.IH by moura
  thus ?case by fastforce
qed simp
```

3.1.6 Constraint Semantics

```
context intruder_model
begin
```

Definitions

The constraint semantics in which the intruder is limited to composition only

```
fun strand_sem_c::"('fun,'var) terms ⇒ ('fun,'var) strand ⇒ ('fun,'var) subst ⇒ bool" ("⟦_; _⟧_c")
where
  "⟦M; []⟧_c = (λI. True)"
| "⟦M; Send t#S⟧_c = (λI. M ⊢_c t · I ∧ ⟦M; S⟧_c I)"
| "⟦M; Receive t#S⟧_c = (λI. ⟦insert (t · I) M; S⟧_c I)"
| "⟦M; Equality _ t t'#S⟧_c = (λI. t · I = t' · I ∧ ⟦M; S⟧_c I)"
| "⟦M; Inequality X F#S⟧_c = (λI. ineq_model I X F ∧ ⟦M; S⟧_c I)"

definition constr_sem_c ("_ ⊨_c ⟨_,_⟩") where "I ⊨_c ⟨S,ϑ⟩ ≡ (ϑ supports I ∧ ⟦{}; S⟧_c I)"
abbreviation constr_sem_c' ("_ ⊨_c ⟨_⟩" 90) where "I ⊨_c ⟨S⟩ ≡ I ⊨_c ⟨S,Var⟩"
```

The full constraint semantics

```
fun strand_sem_d::"('fun,'var) terms ⇒ ('fun,'var) strand ⇒ ('fun,'var) subst ⇒ bool" ("⟦_; _⟧_d")
where
  "⟦M; []⟧_d = (λI. True)"
| "⟦M; Send t#S⟧_d = (λI. M ⊢ t · I ∧ ⟦M; S⟧_d I)"
| "⟦M; Receive t#S⟧_d = (λI. ⟦insert (t · I) M; S⟧_d I)"
| "⟦M; Equality _ t t'#S⟧_d = (λI. t · I = t' · I ∧ ⟦M; S⟧_d I)"
| "⟦M; Inequality X F#S⟧_d = (λI. ineq_model I X F ∧ ⟦M; S⟧_d I)"

definition constr_sem_d ("_ ⊨ ⟨_,_⟩") where "I ⊨ ⟨S,ϑ⟩ ≡ (ϑ supports I ∧ ⟦{}; S⟧_d I)"
abbreviation constr_sem_d' ("_ ⊨ ⟨_⟩" 90) where "I ⊨ ⟨S⟩ ≡ I ⊨ ⟨S,Var⟩"

lemmas strand_sem_induct = strand_sem_c.induct[case_names Nil ConsSnd ConsRcv ConsEq ConsIneq]
```

Lemmata

```
lemma strand_sem_d_if_c: "I ⊨_c ⟨S,ϑ⟩ ⟹ I ⊨ ⟨S,ϑ⟩"
proof -
  assume *: "I ⊨_c ⟨S,ϑ⟩"
  { fix M have "⟦M; S⟧_c I ⟹ ⟦M; S⟧_d I"
    proof (induction S rule: strand_sem_induct)
      case (ConsSnd M t S)
      hence "M ⊢_c t · I" "⟦M; S⟧_d I" by auto
      thus ?case using strand_sem_d.simps(2)[of M t S] by auto
    qed (auto simp add: ineq_model_def)
  }
```

```
  thus ?thesis using * by (simp add: constr_sem_c_def constr_sem_d_def)
qed

lemma strand_sem_mono_ik:
  "⟦M ⊆ M'; ⟦M; S⟧c ϑ⟧ ⟹ ⟦M'; S⟧c ϑ" (is "⟦?A'; ?A''⟧ ⟹ ?A")
  "⟦M ⊆ M'; ⟦M; S⟧d ϑ⟧ ⟹ ⟦M'; S⟧d ϑ" (is "⟦?B'; ?B''⟧ ⟹ ?B")
proof -
  show "⟦?A'; ?A''⟧ ⟹ ?A"
  proof (induction M S arbitrary: M M' rule: strand_sem_induct)
    case (ConsRcv M t S)
    thus ?case using ConsRcv.IH[of "insert (t · ϑ) M" "insert (t · ϑ) M'"] by auto
  next
    case (ConsSnd M t S)
    hence "M ⊢c t · ϑ" "⟦M'; S⟧c ϑ" by auto
    hence "M' ⊢c t · ϑ" using ideduct_synth_mono ⟨M ⊆ M'⟩ by metis
    thus ?case using ⟨⟦M'; S⟧c ϑ⟩ by simp
  qed auto

  show "⟦?B'; ?B''⟧ ⟹ ?B"
  proof (induction M S arbitrary: M M' rule: strand_sem_induct)
    case (ConsRcv M t S)
    thus ?case using ConsRcv.IH[of "insert (t · ϑ) M" "insert (t · ϑ) M'"] by auto
  next
    case (ConsSnd M t S)
    hence "M ⊢ t · ϑ" "⟦M'; S⟧d ϑ" by auto
    hence "M' ⊢ t · ϑ" using ideduct_mono ⟨M ⊆ M'⟩ by metis
    thus ?case using ⟨⟦M'; S⟧d ϑ⟩ by simp
  qed auto
qed

context
begin
private lemma strand_sem_split_left:
  "⟦M; S@S'⟧c ϑ ⟹ ⟦M; S⟧c ϑ"
  "⟦M; S@S'⟧d ϑ ⟹ ⟦M; S⟧d ϑ"
proof (induct S arbitrary: M)
  case (Cons x S)
  { case 1 thus ?case using Cons by (cases x) simp_all }
  { case 2 thus ?case using Cons by (cases x) simp_all }
qed simp_all

private lemma strand_sem_split_right:
  "⟦M; S@S'⟧c ϑ ⟹ ⟦M ∪ (ik_st S ·set ϑ); S'⟧c ϑ"
  "⟦M; S@S'⟧d ϑ ⟹ ⟦M ∪ (ik_st S ·set ϑ); S'⟧d ϑ"
proof (induction S arbitrary: M rule: ik_st_induct)
  case (ConsRcv t S)
  { case 1 thus ?case using ConsRcv.IH[of "insert (t · ϑ) M"] by simp }
  { case 2 thus ?case using ConsRcv.IH[of "insert (t · ϑ) M"] by simp }
qed simp_all

lemmas strand_sem_split[dest] =
  strand_sem_split_left(1) strand_sem_split_right(1)
  strand_sem_split_left(2) strand_sem_split_right(2)
end

lemma strand_sem_Send_split[dest]:
  "⟦⟦M; map Send T⟧c ϑ; t ∈ set T⟧ ⟹ ⟦M; [Send t]⟧c ϑ" (is "⟦?A'; ?A''⟧ ⟹ ?A")
  "⟦⟦M; map Send T⟧d ϑ; t ∈ set T⟧ ⟹ ⟦M; [Send t]⟧d ϑ" (is "⟦?B'; ?B''⟧ ⟹ ?B")
  "⟦⟦M; map Send T@S⟧c ϑ; t ∈ set T⟧ ⟹ ⟦M; Send t#S⟧c ϑ" (is "⟦?C'; ?C''⟧ ⟹ ?C")
  "⟦⟦M; map Send T@S⟧d ϑ; t ∈ set T⟧ ⟹ ⟦M; Send t#S⟧d ϑ" (is "⟦?D'; ?D''⟧ ⟹ ?D")
proof -
  show A: "⟦?A'; ?A''⟧ ⟹ ?A" by (induct "map Send T" arbitrary: T rule: strand_sem_c.induct) auto
  show B: "⟦?B'; ?B''⟧ ⟹ ?B" by (induct "map Send T" arbitrary: T rule: strand_sem_d.induct) auto
```

```
  show "⟦?C'; ?C''⟧ ⟹ ?C" "⟦?D'; ?D''⟧ ⟹ ?D"
    using list.set_map list.simps(8) set_empty ik_snd_empty sup_bot.right_neutral
    by (metis (no_types, lifting) A strand_sem_split(1,2) strand_sem_c.simps(2),
        metis (no_types, lifting) B strand_sem_split(3,4) strand_sem_d.simps(2))
qed

lemma strand_sem_Send_map:
  "(⋀t. t ∈ set T ⟹ ⟦M; [Send t]⟧c ℐ) ⟹ ⟦M; map Send T⟧c ℐ"
  "(⋀t. t ∈ set T ⟹ ⟦M; [Send t]⟧d ℐ) ⟹ ⟦M; map Send T⟧d ℐ"
by (induct T) auto

lemma strand_sem_Receive_map: "⟦M; map Receive T⟧c ℐ" "⟦M; map Receive T⟧d ℐ"
by (induct T arbitrary: M) auto

lemma strand_sem_append[intro]:
  "⟦⟦M; S⟧c ϑ; ⟦M ∪ (ikst S ·set ϑ); S'⟧c ϑ⟧ ⟹ ⟦M; S@S'⟧c ϑ"
  "⟦⟦M; S⟧d ϑ; ⟦M ∪ (ikst S ·set ϑ); S'⟧d ϑ⟧ ⟹ ⟦M; S@S'⟧d ϑ"
proof (induction S arbitrary: M)
  case (Cons x S)
  { case 1 thus ?case using Cons by (cases x) auto }
  { case 2 thus ?case using Cons by (cases x) auto }
qed simp_all

lemma ineq_model_subst:
  fixes F::"(('a,'b) term × ('a,'b) term) list"
  assumes "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
    and "ineq_model (δ ∘s ϑ) X F"
  shows "ineq_model ϑ X (F ·pairs δ)"
proof -
  { fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. fst f · (σ ∘s (δ ∘s ϑ)) ≠ snd f · (σ ∘s (δ ∘s ϑ))) F"
    obtain f where f: "f ∈ set F" "fst f · σ ∘s (δ ∘s ϑ) ≠ snd f · σ ∘s (δ ∘s ϑ)"
      using * by (induct F) auto
    have "σ ∘s (δ ∘s ϑ) = δ ∘s (σ ∘s ϑ)"
      by (metis (no_types, lifting) σ subst_compose_assoc assms(1) inf_sup_aci(1)
              subst_comp_eq_if_disjoint_vars sup_inf_absorb range_vars_alt_def)
    hence "(fst f · δ) · σ ∘s ϑ ≠ (snd f · δ) · σ ∘s ϑ" using f by auto
    moreover have "(fst f · δ, snd f · δ) ∈ set (F ·pairs δ)"
      using f(1) by (auto simp add: subst_apply_pairs_def)
    ultimately have "list_ex (λf. fst f · (σ ∘s ϑ) ≠ snd f · (σ ∘s ϑ)) (F ·pairs δ)"
      using f(1) Bex_set by fastforce
  }
  thus ?thesis using assms unfolding ineq_model_def by simp
qed

lemma ineq_model_subst':
  fixes F::"(('a,'b) term × ('a,'b) term) list"
  assumes "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
    and "ineq_model ϑ X (F ·pairs δ)"
  shows "ineq_model (δ ∘s ϑ) X F"
proof -
  { fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. fst f · (σ ∘s ϑ) ≠ snd f · (σ ∘s ϑ)) (F ·pairs δ)"
    obtain f where f: "f ∈ set (F ·pairs δ)" "fst f · σ ∘s ϑ ≠ snd f · σ ∘s ϑ"
      using * by (induct F) (auto simp add: subst_apply_pairs_def)
    then obtain g where g: "g ∈ set F" "f = g ·p δ" by (auto simp add: subst_apply_pairs_def)
    have "σ ∘s (δ ∘s ϑ) = δ ∘s (σ ∘s ϑ)"
      by (metis (no_types, lifting) σ subst_compose_assoc assms(1) inf_sup_aci(1)
              subst_comp_eq_if_disjoint_vars sup_inf_absorb range_vars_alt_def)
    hence "fst g · σ ∘s (δ ∘s ϑ) ≠ snd g · σ ∘s (δ ∘s ϑ)"
      using f(2) g by (simp add: prod.case_eq_if)
```

```
    hence "list_ex (λf. fst f · (σ ∘s (δ ∘s ϑ)) ≠ snd f · (σ ∘s (δ ∘s ϑ))) F"
      using g Bex_set by fastforce
  }
  thus ?thesis using assms unfolding ineq_model_def by simp
qed

lemma ineq_model_ground_subst:
  fixes F::"(('a,'b) term × ('a,'b) term) list"
  assumes "fvpairs F - set X ⊆ subst_domain δ"
    and "ground (subst_range δ)"
    and "ineq_model δ X F"
  shows "ineq_model (δ ∘s ϑ) X F"
proof -
  { fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. fst f · (σ ∘s δ) ≠ snd f · (σ ∘s δ )) F"
    obtain f where f: "f ∈ set F" "fst f · σ ∘s δ ≠ snd f · σ ∘s δ"
      using * by (induct F) auto
    hence "fv (fst f) ⊆ fvpairs F" "fv (snd f) ⊆ fvpairs F" by auto
    hence "fv (fst f) - set X ⊆ subst_domain δ" "fv (snd f) - set X ⊆ subst_domain δ"
      using assms(1) by auto
    hence "fv (fst f · σ) ⊆ subst_domain δ" "fv (snd f · σ) ⊆ subst_domain δ"
      using σ by (simp_all add: range_vars_alt_def subst_fv_unfold_ground_img)
    hence "fv (fst f · σ ∘s δ) = {}" "fv (snd f · σ ∘s δ) = {}"
      using assms(2) by (simp_all add: subst_fv_dom_ground_if_ground_img)
    hence "fst f · σ ∘s (δ ∘s ϑ) ≠ snd f · σ ∘s (δ ∘s ϑ)" using f(2) subst_ground_ident by fastforce

    hence "list_ex (λf. fst f · (σ ∘s (δ ∘s ϑ)) ≠ snd f · (σ ∘s (δ ∘s ϑ))) F"
      using f(1) Bex_set by fastforce
  }
  thus ?thesis using assms unfolding ineq_model_def by simp
qed

context
begin
private lemma strand_sem_subst_c:
  assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
  shows "⟦M; S⟧c (δ ∘s ϑ) ⟹ ⟦M; S ·st δ⟧c ϑ"
using assms
proof (induction S arbitrary: δ M rule: strand_sem_induct)
  case (ConsSnd M t S)
  hence "⟦M; S ·st δ⟧c ϑ" "M ⊢c t · (δ ∘s ϑ)" by auto
  hence "M ⊢c (t · δ) · ϑ"
    using subst_comp_all[of δ ϑ M] subst_subst_compose[of t δ ϑ] by simp
  thus ?case
    using ‹⟦M; S ·st δ⟧c ϑ›
    unfolding subst_apply_strand_def
    by simp
next
  case (ConsRcv M t S)
  have *: "⟦insert (t · δ ∘s ϑ) M; S⟧c (δ ∘s ϑ)" using ConsRcv.prems(1) by simp
  have "bvarsst (Receive t#S) = bvarsst S" by auto
  hence **: "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}" using ConsRcv.prems(2) by blast
  have "⟦M; Receive (t · δ)#(S ·st δ)⟧c ϑ"
    using ConsRcv.IH[OF * **] by (simp add: subst_all_insert)
  thus ?case by simp
next
  case (ConsIneq M X F S)
  hence *: "⟦M; S ·st δ⟧c ϑ" and
        ***: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
    unfolding bvarsst_def ineq_model_def by auto
  have **: "ineq_model (δ ∘s ϑ) X F"
    using ConsIneq by (auto simp add: subst_compose_assoc ineq_model_def)
```

```
  have "∀γ. subst_domain γ = set X ∧ ground (subst_range γ)
         ⟶ (subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"
    using * ** *** unfolding range_vars_alt_def by auto
  hence "∀γ. subst_domain γ = set X ∧ ground (subst_range γ) ⟶ γ ∘s δ = δ ∘s γ"
    by (metis subst_comp_eq_if_disjoint_vars)
  hence "ineq_model ϑ X (F ·pairs δ)"
    using ineq_model_subst[OF *** **]
    by blast
  moreover have "rm_vars (set X) δ = δ" using ConsIneq.prems(2) by force
  ultimately show ?case using * by auto
qed simp_all

private lemma strand_sem_subst_c':
  assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
  shows "⟦M; S ·st δ⟧c ϑ ⟹ ⟦M; S⟧c (δ ∘s ϑ)"
using assms
proof (induction S arbitrary: δ M rule: strand_sem_induct)
  case (ConsSnd M t S)
  hence "⟦M; [Send t] ·st δ⟧c ϑ" "⟦M; S ·st δ⟧c ϑ" by auto
  hence "⟦M; S⟧c (δ ∘s ϑ)" using ConsSnd.IH[OF _] ConsSnd.prems(2) by auto
  moreover have "⟦M; [Send t]⟧c (δ ∘s ϑ)"
  proof -
    have "M ⊢c t · δ · ϑ" using ‹⟦M; [Send t] ·st δ⟧c ϑ› by auto
    hence "M ⊢c t · (δ ∘s ϑ)" using subst_subst_compose by metis
    thus "⟦M; [Send t]⟧c (δ ∘s ϑ)" by auto
  qed
  ultimately show ?case by auto
next
  case (ConsRcv M t S)
  hence "⟦(insert (t · δ · ϑ) M); S ·st δ⟧c ϑ" by (simp add: subst_all_insert)
  thus ?case using ConsRcv.IH ConsRcv.prems(2) by auto
next
  case (ConsIneq M X F S)
  have δ: "rm_vars (set X) δ = δ" using ConsIneq.prems(2) by force
  hence *: "⟦M; S⟧c (δ ∘s ϑ)"
    and ***: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
    using ConsIneq unfolding bvarsst_def ineq_model_def by auto
  have **: "ineq_model ϑ X (F ·pairs δ)"
    using ConsIneq.prems(1) δ by (auto simp add: subst_compose_assoc ineq_model_def)
  have "∀γ. subst_domain γ = set X ∧ ground (subst_range γ)
         ⟶ (subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"
    using * ** *** unfolding range_vars_alt_def by auto
  hence "∀γ. subst_domain γ = set X ∧ ground (subst_range γ) ⟶ γ ∘s δ = δ ∘s γ"
    by (metis subst_comp_eq_if_disjoint_vars)
  hence "ineq_model (δ ∘s ϑ) X F"
    using ineq_model_subst'[OF *** **]
    by blast
  thus ?case using * by auto
next
  case ConsEq thus ?case unfolding bvarsst_def by auto
qed simp_all

private lemma strand_sem_subst_d:
  assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
  shows "⟦M; S⟧d (δ ∘s ϑ) ⟹ ⟦M; S ·st δ⟧d ϑ"
using assms
proof (induction S arbitrary: δ M rule: strand_sem_induct)
  case (ConsSnd M t S)
  hence "⟦M; S ·st δ⟧d ϑ" "M ⊢ t · (δ ∘s ϑ)" by auto
  hence "M ⊢ (t · δ) · ϑ"
    using subst_comp_all[of δ ϑ M] subst_subst_compose[of t δ ϑ] by simp
  thus ?case using ‹⟦M; S ·st δ⟧d ϑ› by simp
next
```

```
  case (ConsRcv M t S)
  have *: "⟦insert (t · δ ∘s ϑ) M; S⟧d (δ ∘s ϑ)" using ConsRcv.prems(1) by simp
  have "bvarsst (Receive t#S) = bvarsst S" by auto
  hence **: "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}" using ConsRcv.prems(2) by blast
  have "⟦M; Receive (t · δ)#(S ·st δ)⟧d ϑ"
    using ConsRcv.IH[OF * **] by (simp add: subst_all_insert)
  thus ?case by simp
next
  case (ConsIneq M X F S)
  hence *: "⟦M; S ·st δ⟧d ϑ" and
        ***: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
    unfolding bvarsst_def ineq_model_def by auto
  have **: "ineq_model (δ ∘s ϑ) X F"
    using ConsIneq by (auto simp add: subst_compose_assoc ineq_model_def)
  have "∀γ. subst_domain γ = set X ∧ ground (subst_range γ)
          ⟶ (subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"
    using * ** *** unfolding range_vars_alt_def by auto
  hence "∀γ. subst_domain γ = set X ∧ ground (subst_range γ) ⟶ γ ∘s δ = δ ∘s γ"
    by (metis subst_comp_eq_if_disjoint_vars)
  hence "ineq_model ϑ X (F ·pairs δ)"
    using ineq_model_subst[OF *** **]
    by blast
  moreover have "rm_vars (set X) δ = δ" using ConsIneq.prems(2) by force
  ultimately show ?case using * by auto
next
  case ConsEq thus ?case unfolding bvarsst_def by auto
qed simp_all

private lemma strand_sem_subst_d':
  assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
  shows "⟦M; S ·st δ⟧d ϑ ⟹ ⟦M; S⟧d (δ ∘s ϑ)"
using assms
proof (induction S arbitrary: δ M rule: strand_sem_induct)
  case (ConsSnd M t S)
  hence "⟦M; [Send t] ·st δ⟧d ϑ" "⟦M; S ·st δ⟧d ϑ" by auto
  hence "⟦M; S⟧d (δ ∘s ϑ)" using ConsSnd.IH[OF _] ConsSnd.prems(2) by auto
  moreover have "⟦M; [Send t]⟧d (δ ∘s ϑ)"
  proof -
    have "M ⊢ t · δ · ϑ" using ‹⟦M; [Send t] ·st δ⟧d ϑ› by auto
    hence "M ⊢ t · (δ ∘s ϑ)" using subst_subst_compose by metis
    thus "⟦M; [Send t]⟧d (δ ∘s ϑ)" by auto
  qed
  ultimately show ?case by auto
next
  case (ConsRcv M t S)
  hence "⟦insert (t · δ · ϑ) M; S ·st δ⟧d ϑ" by (simp add: subst_all_insert)
  thus ?case using ConsRcv.IH ConsRcv.prems(2) by auto
next
  case (ConsIneq M X F S)
  have δ: "rm_vars (set X) δ = δ" using ConsIneq.prems(2) by force
  hence *: "⟦M; S⟧d (δ ∘s ϑ)"
    and ***: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
    using ConsIneq unfolding bvarsst_def ineq_model_def by auto
  have **: "ineq_model ϑ X (F ·pairs δ)"
    using ConsIneq.prems(1) δ by (auto simp add: subst_compose_assoc ineq_model_def)
  have "∀γ. subst_domain γ = set X ∧ ground (subst_range γ)
          ⟶ (subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"
    using * ** *** unfolding range_vars_alt_def by auto
  hence "∀γ. subst_domain γ = set X ∧ ground (subst_range γ) ⟶ γ ∘s δ = δ ∘s γ"
    by (metis subst_comp_eq_if_disjoint_vars)
  hence "ineq_model (δ ∘s ϑ) X F"
    using ineq_model_subst'[OF *** **]
    by blast
```

```
  thus ?case using * by auto
next
  case ConsEq thus ?case unfolding bvars_st_def by auto
qed simp_all

lemmas strand_sem_subst =
  strand_sem_subst_c strand_sem_subst_c' strand_sem_subst_d strand_sem_subst_d'
end

lemma strand_sem_subst_subst_idem:
  assumes δ: "(subst_domain δ ∪ range_vars δ) ∩ bvars_st S = {}"
  shows "⟦⟦M; S ·_st δ⟧_c (δ ∘_s ϑ); subst_idem δ⟧ ⟹ ⟦M; S⟧_c (δ ∘_s ϑ)"
using strand_sem_subst(2)[OF assms, of M "δ ∘_s ϑ"] subst_compose_assoc[of δ δ ϑ]
unfolding subst_idem_def by argo

lemma strand_sem_subst_comp:
  assumes "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st S = {}"
    and "⟦M; S⟧_c δ" "subst_domain ϑ ∩ (vars_st S ∪ fv_set M) = {}"
  shows "⟦M; S⟧_c (ϑ ∘_s δ)"
proof -
  from assms(3) have "subst_domain ϑ ∩ vars_st S = {}" "subst_domain ϑ ∩ fv_set M = {}" by auto
  hence "S ·_st ϑ = S" "M ·_set ϑ = M" using strand_substI set_subst_ident[of M ϑ] by (blast, blast)
  thus ?thesis using assms(2) by (auto simp add: strand_sem_subst(2)[OF assms(1)])
qed

lemma strand_sem_c_imp_ineqs_neq:
  assumes "⟦M; S⟧_c ℐ" "Inequality X [(t,t')] ∈ set S"
  shows "t ≠ t' ∧ (∀δ. subst_domain δ = set X ∧ ground (subst_range δ)
                      ⟶ t · δ ≠ t' · δ ∧ t · δ · ℐ ≠ t' · δ · ℐ)"
using assms
proof (induction rule: strand_sem_induct)
  case (ConsIneq M Y F S) thus ?case
  proof (cases "Inequality X [(t,t')] ∈ set S")
    case False
    hence "X = Y" "F = [(t,t')]" using ConsIneq by auto
    hence *: "∀ϑ. subst_domain ϑ = set X ∧ ground (subst_range ϑ) ⟶ t · ϑ · ℐ ≠ t' · ϑ · ℐ"
      using ConsIneq by (auto simp add: ineq_model_def)
    then obtain ϑ where ϑ: "subst_domain ϑ = set X" "ground (subst_range ϑ)" "t · ϑ · ℐ ≠ t' · ϑ ·
ℐ"
      using interpretation_subst_exists'[of "set X"] by moura
    hence "t ≠ t'" by auto
    moreover have "⋀ℐ ϑ. t · ϑ · ℐ ≠ t' · ϑ · ℐ ⟹ t · ϑ ≠ t' · ϑ" by auto
    ultimately show ?thesis using * by auto
  qed simp
qed simp_all

lemma strand_sem_c_imp_ineq_model:
  assumes "⟦M; S⟧_c ℐ" "Inequality X F ∈ set S"
  shows "ineq_model ℐ X F"
using assms by (induct S rule: strand_sem_induct) force+

lemma strand_sem_wf_simple_fv_sat:
  assumes "wf_st {} S" "simple S" "⟦{}; S⟧_c ℐ"
  shows "⋀v. v ∈ wfrestrictedvars_st S ⟹ ik_st S ·_set ℐ ⊢_c ℐ v"
using assms
proof (induction S rule: wf_st_simple_induct)
  case (ConsRcv t S)
  have "v ∈ wfrestrictedvars_st S"
    using ConsRcv.hyps(3) ConsRcv.prems(1) vars_snd_rcv_strand2
    by fastforce
  moreover have "⟦{}; S⟧_c ℐ" using ‹⟦{}; S@[Receive t]⟧_c ℐ› by blast
  moreover have "ik_st S ·_set ℐ ⊆ ik_st (S@[Receive t]) ·_set ℐ" by auto
  ultimately show ?case using ConsRcv.IH ideduct_synth_mono by meson
```

```
next
  case (ConsIneq X F S)
  hence "v ∈ wfrestrictedvars_st S" by fastforce
  moreover have "⟦{}; S⟧_c ℐ" using ‹⟦{}; S@[Inequality X F]⟧_c ℐ› by blast
  moreover have "ik_st S ·_set ℐ ⊆ ik_st (S@[Inequality X F]) ·_set ℐ" by auto
  ultimately show ?case using ConsIneq.IH ideduct_synth_mono by meson
next
  case (ConsSnd w S)
  hence *: "⟦{}; S⟧_c ℐ" "ik_st S ·_set ℐ ⊢_c ℐ w" by auto
  have **: "ik_st S ·_set ℐ ⊆ ik_st (S@[Send (Var w)]) ·_set ℐ" by simp
  show ?case
  proof (cases "v = w")
    case True thus ?thesis using *(2) ideduct_synth_mono[OF _ **] by meson
  next
    case False
    hence "v ∈ wfrestrictedvars_st S" using ConsSnd.prems(1) by auto
    thus ?thesis using ConsSnd.IH[OF _ *(1)] ideduct_synth_mono[OF _ **] by metis
  qed
qed simp

lemma strand_sem_wf_ik_or_assignment_rhs_fun_subterm:
  assumes "wf_st {} A" "⟦{}; A⟧_c ℐ" "Var x ∈ ik_st A" "ℐ x = Fun f T"
          "t_i ∈ set T" "¬ik_st A ·_set ℐ ⊢_c t_i" "interpretation_subst ℐ"
  obtains S where
    "Fun f S ∈ subterms_set (ik_st A) ∨ Fun f S ∈ subterms_set (assignment_rhs_st A)"
    "Fun f T = Fun f S · ℐ"
proof -
  have "x ∈ wfrestrictedvars_st A"
    by (metis (no_types) assms(3) set_rev_mp term.set_intros(3) vars_subset_if_in_strand_ik2)
  moreover have "Fun f T · ℐ = Fun f T"
    by (metis subst_ground_ident interpretation_grounds_all assms(4,7))
  ultimately obtain A_pre A_suf where *:
      "¬(∃w ∈ wfrestrictedvars_st A_pre. Fun f T ⊑ ℐ w)"
      "(∃t. A = A_pre@Send t#A_suf ∧ Fun f T ⊑ t · ℐ) ∨
       (∃t t'. A = A_pre@Equality Assign t t'#A_suf ∧ Fun f T ⊑ t · ℐ)"
    using wf_strand_first_Send_var_split[OF assms(1)] assms(4) subtermeqI' by metis
  moreover
  { fix t assume **: "A = A_pre@Send t#A_suf" "Fun f T ⊑ t · ℐ"
    hence "ik_st A_pre ·_set ℐ ⊢_c t · ℐ" "¬ik_st A_pre ·_set ℐ ⊢_c t_i"
      using assms(2,6) by (auto intro: ideduct_synth_mono)
    then obtain s where s: "s ∈ ik_st A_pre" "Fun f T ⊑ s · ℐ"
      using assms(5) **(2) by (induct rule: intruder_synth_induct) auto
    then obtain g S where gS: "Fun g S ⊑ s" "Fun f T = Fun g S · ℐ"
      using subterm_subst_not_img_subterm[OF s(2)] *(1) by force
    hence ?thesis using that **(1) s(1) by force
  }
  moreover
  { fix t t' assume **: "A = A_pre@Equality Assign t t'#A_suf" "Fun f T ⊑ t · ℐ"
    with assms(2) have "t · ℐ = t' · ℐ" by auto
    hence "Fun f T ⊑ t' · ℐ" using **(2) by auto
    from assms(1) **(1) have "fv t' ⊆ wfrestrictedvars_st A_pre"
      using wf_eq_fv[of "{}" A_pre t t' A_suf] vars_snd_rcv_strand_subset2(4)[of A_pre]
      by blast
    then obtain g S where gS: "Fun g S ⊑ t'" "Fun f T = Fun g S · ℐ"
      using subterm_subst_not_img_subterm[OF ‹Fun f T ⊑ t' · ℐ›] *(1) by fastforce
    hence ?thesis using that **(1) by auto
  }
  ultimately show ?thesis by auto
qed

lemma strand_sem_not_unif_is_sat_ineq:
  assumes "∄ϑ. Unifier ϑ t t'"
  shows "⟦M; [Inequality X [(t,t')]]⟧_c ℐ" "⟦M; [Inequality X [(t,t')]]⟧_d ℐ"
```

```
using assms list_ex_simps(1)[of _ "(t,t')" "[]"] prod.sel[of t t']
      strand_sem_c.simps(1,5) strand_sem_d.simps(1,5)
unfolding ineq_model_def by presburger+

lemma ineq_model_singleI[intro]:
  assumes "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ t · δ · ℐ ≠ t' · δ · ℐ"
  shows "ineq_model ℐ X [(t,t')]"
using assms unfolding ineq_model_def by auto

lemma ineq_model_singleE:
  assumes "ineq_model ℐ X [(t,t')]"
  shows "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ t · δ · ℐ ≠ t' · δ · ℐ"
using assms unfolding ineq_model_def by auto

lemma ineq_model_single_iff:
  fixes F::"(('a,'b) term × ('a,'b) term) list"
  shows "ineq_model ℐ X F ⟷
         ineq_model ℐ X [(Fun f (Fun c []#map fst F),Fun f (Fun c []#map snd F))]"
    (is "?A ⟷ ?B")
proof -
  let ?P = "λδ f. fst f · (δ ∘s ℐ) ≠ snd f · (δ ∘s ℐ)"
  let ?Q = "λδ t t'. t · (δ ∘s ℐ) ≠ t' · (δ ∘s ℐ)"
  let ?T = "λg. Fun c []#map g F"
  let ?S = "λδ g. map (λx. x · (δ ∘s ℐ)) (Fun c []#map g F)"
  let ?t = "Fun f (?T fst)"
  let ?t' = "Fun f (?T snd)"

  have len: "⋀g h. length (?T g) = length (?T h)"
            "⋀g h δ. length (?S δ g) = length (?T h)"
            "⋀g h δ. length (?S δ g) = length (?T h)"
            "⋀g h δ σ. length (?S δ g) = length (?S σ h)"
    by simp_all

  { fix δ::"('a,'b) subst"
    assume δ: "subst_domain δ = set X" "ground (subst_range δ)"
    have "list_ex (?P δ) F ⟷ ?Q δ ?t ?t'"
    proof
      assume "list_ex (?P δ) F"
      then obtain a where a: "a ∈ set F" "?P δ a" by (metis (mono_tags, lifting) Bex_set)
      thus "?Q δ ?t ?t'" by auto
    qed (fastforce simp add: Bex_set)
  } thus ?thesis unfolding ineq_model_def by auto
qed
```

3.1.7 Constraint Semantics (Alternative, Equivalent Version)

These are the constraint semantics used in the CSF 2017 paper

```
fun strand_sem_c'::"('fun,'var) terms ⇒ ('fun,'var) strand ⇒ ('fun,'var) subst ⇒ bool"  ("⟦_;
_⟧c''")
  where
  "⟦M; []⟧c' = (λℐ. True)"
| "⟦M; Send t#S⟧c' = (λℐ. M ·set ℐ ⊢c t · ℐ ∧ ⟦M; S⟧c' ℐ)"
| "⟦M; Receive t#S⟧c' = ⟦insert t M; S⟧c'"
| "⟦M; Equality _ t t'#S⟧c' = (λℐ. t · ℐ = t' · ℐ ∧ ⟦M; S⟧c' ℐ)"
| "⟦M; Inequality X F#S⟧c' = (λℐ. ineq_model ℐ X F ∧ ⟦M; S⟧c' ℐ)"

fun strand_sem_d'::"('fun,'var) terms ⇒ ('fun,'var) strand ⇒ ('fun,'var) subst ⇒ bool" ("⟦_;
_⟧d''")
where
  "⟦M; []⟧d' = (λℐ. True)"
| "⟦M; Send t#S⟧d' = (λℐ. M ·set ℐ ⊢ t · ℐ ∧ ⟦M; S⟧d' ℐ)"
| "⟦M; Receive t#S⟧d' = ⟦insert t M; S⟧d'"
```

```
| "⟦M; Equality _ t t'#S⟧d' = (λI. t · I = t' · I ∧ ⟦M; S⟧d' I)"
| "⟦M; Inequality X F#S⟧d' = (λI. ineq_model I X F ∧ ⟦M; S⟧d' I)"

lemma strand_sem_eq_defs:
  "⟦M; A⟧c' I = ⟦M ·set I; A⟧c I"
  "⟦M; A⟧d' I = ⟦M ·set I; A⟧d I"
proof -
  have 1: "⟦M; A⟧c' I ⟹ ⟦M ·set I; A⟧c I"
    by (induct A arbitrary: M rule: strand_sem_induct) force+
  have 2: "⟦M ·set I; A⟧c I ⟹ ⟦M; A⟧c' I"
    by (induct A arbitrary: M rule: strand_sem_c'.induct) auto
  have 3: "⟦M; A⟧d' I ⟹ ⟦M ·set I; A⟧d I"
    by (induct A arbitrary: M rule: strand_sem_induct) force+
  have 4: "⟦M ·set I; A⟧d I ⟹ ⟦M; A⟧d' I"
    by (induct A arbitrary: M rule: strand_sem_d'.induct) auto

  show "⟦M; A⟧c' I = ⟦M ·set I; A⟧c I" "⟦M; A⟧d' I = ⟦M ·set I; A⟧d I"
    by (metis 1 2, metis 3 4)
qed

lemma strand_sem_split'[dest]:
  "⟦M; S@S'⟧c' ϑ ⟹ ⟦M; S⟧c' ϑ"
  "⟦M; S@S'⟧c' ϑ ⟹ ⟦M ∪ ikst S; S'⟧c' ϑ"
  "⟦M; S@S'⟧d' ϑ ⟹ ⟦M; S⟧d' ϑ"
  "⟦M; S@S'⟧d' ϑ ⟹ ⟦M ∪ ikst S; S'⟧d' ϑ"
using strand_sem_eq_defs[of M "S@S'" ϑ]
      strand_sem_eq_defs[of M S ϑ]
      strand_sem_eq_defs[of "M ∪ ikst S" S' ϑ]
      strand_sem_split(2,4)
by (auto simp add: image_Un)

lemma strand_sem_append'[intro]:
  "⟦M; S⟧c' ϑ ⟹ ⟦M ∪ ikst S; S'⟧c' ϑ ⟹ ⟦M; S@S'⟧c' ϑ"
  "⟦M; S⟧d' ϑ ⟹ ⟦M ∪ ikst S; S'⟧d' ϑ ⟹ ⟦M; S@S'⟧d' ϑ"
using strand_sem_eq_defs[of M "S@S'" ϑ]
      strand_sem_eq_defs[of M S ϑ]
      strand_sem_eq_defs[of "M ∪ ikst S" S' ϑ]
by (auto simp add: image_Un)

end
```

3.1.8 Dual Strands

```
fun dualst::"('a,'b) strand ⇒ ('a,'b) strand" where
  "dualst [] = []"
| "dualst (Receive t#S) = Send t#(dualst S)"
| "dualst (Send t#S) = Receive t#(dualst S)"
| "dualst (x#S) = x#(dualst S)"

lemma dualst_append: "dualst (A@B) = (dualst A)@(dualst B)"
by (induct A rule: dualst.induct) auto

lemma dualst_self_inverse: "dualst (dualst S) = S"
proof (induction S)
  case (Cons x S) thus ?case by (cases x) auto
qed simp

lemma dualst_trms_eq: "trmsst (dualst S) = trmsst S"
proof (induction S)
  case (Cons x S) thus ?case by (cases x) auto
qed simp

lemma dualst_fv: "fvst (dualst A) = fvst A"
```

```
by (induct A rule: dual_st.induct) auto

lemma dual_st_bvars: "bvars_st (dual_st A) = bvars_st A"
by (induct A rule: dual_st.induct) fastforce+


end
```

3.2 The Lazy Intruder (Lazy_Intruder)

```
theory Lazy_Intruder
imports Strands_and_Constraints Intruder_Deduction
begin

context intruder_model
begin
```

3.2.1 Definition of the Lazy Intruder

The lazy intruder constraint reduction system, defined as a relation on constraint states

```
inductive_set LI_rel::
    "((('fun,'var) strand × (('fun,'var) subst)) ×
       ('fun,'var) strand × (('fun,'var) subst)) set"
  and LI_rel' (infix "⇝" 50)
  and LI_rel_trancl (infix "⇝⁺" 50)
  and LI_rel_rtrancl (infix "⇝*" 50)
where
  "A ⇝ B ≡ (A,B) ∈ LI_rel"
| "A ⇝⁺ B ≡ (A,B) ∈ LI_rel⁺"
| "A ⇝* B ≡ (A,B) ∈ LI_rel*"

| Compose: "⟦simple S; length T = arity f; public f⟧
            ⟹ (S@Send (Fun f T)#S',ϑ) ⇝ (S@(map Send T)@S',ϑ)"
| Unify: "⟦simple S; Fun f T' ∈ ik_st S; Some δ = mgu (Fun f T) (Fun f T')⟧
          ⟹ (S@Send (Fun f T)#S',ϑ) ⇝ ((S@S') ·_st δ,ϑ ∘_s δ)"
| Equality: "⟦simple S; Some δ = mgu t t'⟧
          ⟹ (S@Equality _ t t'#S',ϑ) ⇝ ((S@S') ·_st δ,ϑ ∘_s δ)"
```

3.2.2 Lemma: The Lazy Intruder is Well-founded

```
context
begin
private lemma LI_compose_measure_lt: "((S@(map Send T)@S',ϑ₁), (S@Send (Fun f T)#S',ϑ₂)) ∈
measure_st"
using strand_fv_card_map_fun_eq[of S f T S'] strand_size_map_fun_lt(2)[of T f]
by (simp add: measure_st_def size_st_def)

private lemma LI_unify_measure_lt:
  assumes "Some δ = mgu (Fun f T) t" "fv t ⊆ fv_st S"
  shows "(((S@S') ·_st δ,ϑ₁), (S@Send (Fun f T)#S',ϑ₂)) ∈ measure_st"
proof (cases "δ = Var")
  assume "δ = Var"
  hence "(S@S') ·_st δ = S@S'" by blast
  thus ?thesis
    using strand_fv_card_rm_fun_le[of S S' f T]
    by (auto simp add: measure_st_def size_st_def)
next
  assume "δ ≠ Var"
  then obtain v where "v ∈ fv (Fun f T) ∪ fv t" "subst_elim δ v"
    using mgu_eliminates[OF assms(1)[symmetric]] by metis
  hence v_in: "v ∈ fv_st (S@Send (Fun f T)#S')"
    using assms(2) by (auto simp add: measure_st_def size_st_def)
```

```
  have "range_vars δ ⊆ fv (Fun f T) ∪ fv_st S"
    using assms(2) mgu_vars_bounded[OF assms(1)[symmetric]] by auto
  hence img_bound: "range_vars δ ⊆ fv_st (S@Send (Fun f T)#S')" by auto

  have finite_fv: "finite (fv_st (S@Send (Fun f T)#S'))" by auto

  have "v ∉ fv_st ((S@Send (Fun f T)#S') ·_st δ)"
    using strand_fv_subst_subset_if_subst_elim[OF ‹subst_elim δ v›] v_in by metis
  hence v_not_in: "v ∉ fv_st ((S@S') ·_st δ)" by auto

  have "fv_st ((S@S') ·_st δ) ⊆ fv_st (S@Send (Fun f T)#S')"
    using strand_subst_fv_bounded_if_img_bounded[OF img_bound] by simp
  hence "fv_st ((S@S') ·_st δ) ⊂ fv_st (S@Send (Fun f T)#S')" using v_in v_not_in by blast
  hence "card (fv_st ((S@S') ·_st δ)) < card (fv_st (S@Send (Fun f T)#S'))"
    using psubset_card_mono[OF finite_fv] by simp
  thus ?thesis by (auto simp add: measure_st_def size_st_def)
qed

private lemma LI_equality_measure_lt:
  assumes "Some δ = mgu t t'"
  shows "(((S@S') ·_st δ,ϑ1), (S@Equality a t t'#S',ϑ2)) ∈ measure_st"
proof (cases "δ = Var")
  assume "δ = Var"
  hence "(S@S') ·_st δ = S@S'" by blast
  thus ?thesis
    using strand_fv_card_rm_eq_le[of S S' a t t']
    by (auto simp add: measure_st_def size_st_def)
next
  assume "δ ≠ Var"
  then obtain v where "v ∈ fv t ∪ fv t'" "subst_elim δ v"
    using mgu_eliminates[OF assms(1)[symmetric]] by metis
  hence v_in: "v ∈ fv_st (S@Equality a t t'#S')" using assms by auto

  have "range_vars δ ⊆ fv t ∪ fv t' ∪ fv_st S"
    using assms mgu_vars_bounded[OF assms(1)[symmetric]] by auto
  hence img_bound: "range_vars δ ⊆ fv_st (S@Equality a t t'#S')" by auto

  have finite_fv: "finite (fv_st (S@Equality a t t'#S'))" by auto

  have "v ∉ fv_st ((S@Equality a t t'#S') ·_st δ)"
    using strand_fv_subst_subset_if_subst_elim[OF ‹subst_elim δ v›] v_in by metis
  hence v_not_in: "v ∉ fv_st ((S@S') ·_st δ)" by auto

  have "fv_st ((S@S') ·_st δ) ⊆ fv_st (S@Equality a t t'#S')"
    using strand_subst_fv_bounded_if_img_bounded[OF img_bound] by simp
  hence "fv_st ((S@S') ·_st δ) ⊂ fv_st (S@Equality a t t'#S')" using v_in v_not_in by blast
  hence "card (fv_st ((S@S') ·_st δ)) < card (fv_st (S@Equality a t t'#S'))"
    using psubset_card_mono[OF finite_fv] by simp
  thus ?thesis by (auto simp add: measure_st_def size_st_def)
qed

private lemma LI_in_measure: "(S1,ϑ1) ⇝ (S2,ϑ2) ⟹ ((S2,ϑ2),(S1,ϑ1)) ∈ measure_st"
proof (induction rule: LI_rel.induct)
  case (Compose S T f S' ϑ) thus ?case using LI_compose_measure_lt[of S T S'] by metis
next
  case (Unify S f U δ T S' ϑ)
  hence "fv (Fun f U) ⊆ fv_st S"
    using fv_snd_rcv_strand_subset(2)[of S] by force
  thus ?case using LI_unify_measure_lt[OF Unify.hyps(3), of S S'] by metis
qed (metis LI_equality_measure_lt)

private lemma LI_in_measure_trans: "(S1,ϑ1) ⇝+ (S2,ϑ2) ⟹ ((S2,ϑ2),(S1,ϑ1)) ∈ measure_st"
```

```
by (induction rule: trancl.induct, metis surjective_pairing LI_in_measure)
   (metis (no_types, lifting) surjective_pairing LI_in_measure measure_st_trans trans_def)

private lemma LI_converse_wellfounded_trans: "wf ((LI_rel^+)^-1)"
proof -
  have "(LI_rel^+)^-1 ⊆ measure_st" using LI_in_measure_trans by auto
  thus ?thesis using measure_st_wellfounded wf_subset by metis
qed

private lemma LI_acyclic_trans: "acyclic (LI_rel^+)"
using wf_acyclic[OF LI_converse_wellfounded_trans] acyclic_converse by metis

private lemma LI_acyclic: "acyclic LI_rel"
using LI_acyclic_trans acyclic_subset by (simp add: acyclic_def)

lemma LI_no_infinite_chain: "¬(∃f. ∀i. f i ⇝^+ f (Suc i))"
proof -
  have "¬(∃f. ∀i. (f (Suc i), f i) ∈ (LI_rel^+)^-1)"
    using wf_iff_no_infinite_down_chain LI_converse_wellfounded_trans by metis
  thus ?thesis by simp
qed

private lemma LI_unify_finite:
  assumes "finite M"
  shows "finite {((S@Send (Fun f T)#S',ϑ), ((S@S') ·_st δ,ϑ ∘_s δ)) | δ T'.
                simple S ∧ Fun f T' ∈ M ∧ Some δ = mgu (Fun f T) (Fun f T')}"
using assms
proof (induction M rule: finite_induct)
  case (insert m M) thus ?case
  proof (cases m)
    case (Fun g U)
    let ?a = "λδ. ((S@Send (Fun f T)#S',ϑ), ((S@S') ·_st δ,ϑ ∘_s δ))"
    let ?A = "λB. {?a δ | δ T'. simple S ∧ Fun f T' ∈ B ∧ Some δ = mgu (Fun f T) (Fun f T')}"

    have "?A (insert m M) = (?A M) ∪ (?A {m})" by auto
    moreover have "finite (?A {m})"
    proof (cases "∃δ. Some δ = mgu (Fun f T) (Fun g U)")
      case True
      then obtain δ where δ: "Some δ = mgu (Fun f T) (Fun g U)" by blast

      have A_m_eq: "⋀δ'. ?a δ' ∈ ?A {m} ⟹ ?a δ = ?a δ'"
      proof -
        fix δ' assume "?a δ' ∈ ?A {m}"
        hence "∃σ. Some σ = mgu (Fun f T) (Fun g U) ∧ ?a σ = ?a δ'"
          using ‹m = Fun g U› by auto
        thus "?a δ = ?a δ'" by (metis δ option.inject)
      qed

      have "?A {m} = {} ∨ ?A {m} = {?a δ}"
      proof (cases "simple S ∧ ?A {m} ≠ {}")
        case True
        hence "simple S" "?A {m} ≠ {}" by meson+
        hence "?A {m} = {?a δ | δ. Some δ = mgu (Fun f T) (Fun g U)}" using ‹m = Fun g U› by auto
        hence "?a δ ∈ ?A {m}" using δ by auto
       show ?thesis
        proof (rule ccontr)
          assume "¬(?A {m} = {} ∨ ?A {m} = {?a δ})"
          then obtain B where B: "?A {m} = insert (?a δ) B" "?a δ ∉ B" "B ≠ {}"
            using ‹?A {m} ≠ {}› ‹?a δ ∈ ?A {m}› by (metis (no_types, lifting) Set.set_insert)
          then obtain b where b: "?a δ ≠ b" "b ∈ B" by (metis (no_types, lifting) ex_in_conv)
          then obtain δ' where δ': "b = ?a δ'" using B(1) by blast
          moreover have "?a δ' ∈ ?A {m}" using B(1) b(2) δ' by auto
          hence "?a δ = ?a δ'" by (blast dest!: A_m_eq)
```

```
        ultimately show False using b(1) by simp
      qed
    qed auto
    thus ?thesis by (metis (no_types, lifting) finite.emptyI finite_insert)
  next
    case False
    hence "?A {m} = {}" using ‹m = Fun g U› by blast
    thus ?thesis by (metis finite.emptyI)
  qed
  ultimately show ?thesis using insert.IH by auto
 qed simp
qed fastforce
end
```

3.2.3 Lemma: The Lazy Intruder Preserves Well-formedness

```
context
begin
private lemma LI_preserves_subst_wf_single:
  assumes "(S₁,ϑ₁) ⇝ (S₂,ϑ₂)" "fv_st S₁ ∩ bvars_st S₁ = {}" "wf_subst ϑ₁"
  and "subst_domain ϑ₁ ∩ vars_st S₁ = {}" "range_vars ϑ₁ ∩ bvars_st S₁ = {}"
  shows "fv_st S₂ ∩ bvars_st S₂ = {}" "wf_subst ϑ₂"
  and "subst_domain ϑ₂ ∩ vars_st S₂ = {}" "range_vars ϑ₂ ∩ bvars_st S₂ = {}"
using assms
proof (induction rule: LI_rel.induct)
  case (Compose S X f S' ϑ)
  { case 1 thus ?case using vars_st_snd_map by auto }
  { case 2 thus ?case using vars_st_snd_map by auto }
  { case 3 thus ?case using vars_st_snd_map by force }
  { case 4 thus ?case using vars_st_snd_map by auto }
next
  case (Unify S f U δ T S' ϑ)
  hence "fv (Fun f U) ⊆ fv_st S" using fv_subset_if_in_strand_ik' by blast
  hence *: "subst_domain δ ∪ range_vars δ ⊆ fv_st (S@Send (Fun f T)#S')"
    using mgu_vars_bounded[OF Unify.hyps(3)[symmetric]]
    unfolding range_vars_alt_def by (fastforce simp del: subst_range.simps)

  have "fv_st (S@S') ⊆ fv_st (S@Send (Fun f T)#S')" "vars_st (S@S') ⊆ vars_st (S@Send (Fun f T)#S')"
    by auto
  hence **: "fv_st (S@S' ·_st δ) ⊆ fv_st (S@Send (Fun f T)#S')"
            "vars_st (S@S' ·_st δ) ⊆ vars_st (S@Send (Fun f T)#S')"
    using subst_sends_strand_fv_to_img[of "S@S'" δ]
          strand_subst_vars_union_bound[of "S@S'" δ] *
    by blast+

  have "wf_subst δ" by (fact mgu_gives_wellformed_subst[OF Unify.hyps(3)[symmetric]])

  { case 1
    have "bvars_st (S@S' ·_st δ) = bvars_st (S@Send (Fun f T)#S')"
      using bvars_subst_ident[of "S@S'" δ] by auto
    thus ?case using 1 ** by blast
  }
  { case 2
    hence "subst_domain ϑ ∩ subst_domain δ = {}" "subst_domain ϑ ∩ range_vars δ = {}"
      using * by blast+
    thus ?case by (metis wf_subst_compose[OF ‹wf_subst ϑ› ‹wf_subst δ›])
  }
  { case 3
    hence "subst_domain ϑ ∩ vars_st (S@S' ·_st δ) = {}" using ** by blast
    moreover have "v ∈ fv_st (S@Send (Fun f T)#S')" when "v ∈ subst_domain δ" for v
      using * that by blast
    hence "subst_domain δ ∩ fv_st (S@S' ·_st δ) = {}"
      using mgu_eliminates_dom[OF Unify.hyps(3)[symmetric],
```

```
            THEN strand_fv_subst_subset_if_subst_elim, of _ "S@Send (Fun f T)#S'"]
      unfolding subst_elim_def by auto
    moreover have "bvars_st (S@S' ·_st δ) = bvars_st (S@Send (Fun f T)#S')"
      using bvars_subst_ident[of "S@S'" δ] by auto
    hence "subst_domain δ ∩ bvars_st (S@S' ·_st δ) = {}" using 3(1) * by blast
    ultimately show ?case
      using ** * subst_domain_compose[of ϑ δ] vars_st_is_fv_st_bvars_st[of "S@S' ·_st δ"]
      by blast
  }
  { case 4
    have ***: "bvars_st (S@S' ·_st δ) = bvars_st (S@Send (Fun f T)#S')"
      using bvars_subst_ident[of "S@S'" δ] by auto
    hence "range_vars δ ∩ bvars_st (S@S' ·_st δ) = {}" using 4(1) * by blast
    thus ?case using subst_img_comp_subset[of ϑ δ] 4(4) *** by blast
  }
next
  case (Equality S δ t t' a S' ϑ)
  hence *: "subst_domain δ ∪ range_vars δ ⊆ fv_st (S@Equality a t t'#S')"
    using mgu_vars_bounded[OF Equality.hyps(2)[symmetric]]
    unfolding range_vars_alt_def by fastforce

  have "fv_st (S@S') ⊆ fv_st (S@Equality a t t'#S')" "vars_st (S@S') ⊆ vars_st (S@Equality a t t'#S')"
    by auto
  hence **: "fv_st (S@S' ·_st δ) ⊆ fv_st (S@Equality a t t'#S')"
            "vars_st (S@S' ·_st δ) ⊆ vars_st (S@Equality a t t'#S')"
    using subst_sends_strand_fv_to_img[of "S@S'" δ]
          strand_subst_vars_union_bound[of "S@S'" δ] *
    by blast+

  have "wf_subst δ" by (fact mgu_gives_wellformed_subst[OF Equality.hyps(2)[symmetric]])

  { case 1
    have "bvars_st (S@S' ·_st δ) = bvars_st (S@Equality a t t'#S')"
      using bvars_subst_ident[of "S@S'" δ] by auto
    thus ?case using 1 ** by blast
  }
  { case 2
    hence "subst_domain ϑ ∩ subst_domain δ = {}" "subst_domain ϑ ∩ range_vars δ = {}"
      using * by blast+
    thus ?case by (metis wf_subst_compose[OF ‹wf_subst ϑ› ‹wf_subst δ›])
  }
  { case 3
    hence "subst_domain ϑ ∩ vars_st (S@S' ·_st δ) = {}" using ** by blast
    moreover have "v ∈ fv_st (S@Equality a t t'#S')" when "v ∈ subst_domain δ" for v
      using * that by blast
    hence "subst_domain δ ∩ fv_st (S@S' ·_st δ) = {}"
      using mgu_eliminates_dom[OF Equality.hyps(2)[symmetric],
            THEN strand_fv_subst_subset_if_subst_elim, of _ "S@Equality a t t'#S'"]
      unfolding subst_elim_def by auto
    moreover have "bvars_st (S@S' ·_st δ) = bvars_st (S@Equality a t t'#S')"
      using bvars_subst_ident[of "S@S'" δ] by auto
    hence "subst_domain δ ∩ bvars_st (S@S' ·_st δ) = {}" using 3(1) * by blast
    ultimately show ?case
      using ** * subst_domain_compose[of ϑ δ] vars_st_is_fv_st_bvars_st[of "S@S' ·_st δ"]
      by blast
  }
  { case 4
    have ***: "bvars_st (S@S' ·_st δ) = bvars_st (S@Equality a t t'#S')"
      using bvars_subst_ident[of "S@S'" δ] by auto
    hence "range_vars δ ∩ bvars_st (S@S' ·_st δ) = {}" using 4(1) * by blast
    thus ?case using subst_img_comp_subset[of ϑ δ] 4(4) *** by blast
  }
qed
```

```
private lemma LI_preserves_subst_wf:
  assumes "(S₁,ϑ₁) ⇝* (S₂,ϑ₂)" "fv_st S₁ ∩ bvars_st S₁ = {}" "wf_subst ϑ₁"
  and "subst_domain ϑ₁ ∩ vars_st S₁ = {}" "range_vars ϑ₁ ∩ bvars_st S₁ = {}"
  shows "fv_st S₂ ∩ bvars_st S₂ = {}" "wf_subst ϑ₂"
  and "subst_domain ϑ₂ ∩ vars_st S₂ = {}" "range_vars ϑ₂ ∩ bvars_st S₂ = {}"
using assms
proof (induction S₂ ϑ₂ rule: rtrancl_induct2)
  case (step S_i ϑ_i S_j ϑ_j)
  { case 1 thus ?case using LI_preserves_subst_wf_single[OF ‹(S_i,ϑ_i) ⇝ (S_j,ϑ_j)›] step.IH by metis }
  { case 2 thus ?case using LI_preserves_subst_wf_single[OF ‹(S_i,ϑ_i) ⇝ (S_j,ϑ_j)›] step.IH by metis }
  { case 3 thus ?case using LI_preserves_subst_wf_single[OF ‹(S_i,ϑ_i) ⇝ (S_j,ϑ_j)›] step.IH by metis }
  { case 4 thus ?case using LI_preserves_subst_wf_single[OF ‹(S_i,ϑ_i) ⇝ (S_j,ϑ_j)›] step.IH by metis }
qed metis

lemma LI_preserves_wellformedness:
  assumes "(S₁,ϑ₁) ⇝* (S₂,ϑ₂)" "wf_constr S₁ ϑ₁"
  shows "wf_constr S₂ ϑ₂"
proof -
  have *: "wf_st {} S_j"
    when "(S_i, ϑ_i) ⇝ (S_j, ϑ_j)" "wf_constr S_i ϑ_i" for S_i ϑ_i S_j ϑ_j
    using that
  proof (induction rule: LI_rel.induct)
    case (Unify S f U δ T S' ϑ)
    have "fv (Fun f T) ∪ fv (Fun f U) ⊆ fv_st (S@Send (Fun f T)#S')" using Unify.hyps(2) by force
    hence "subst_domain δ ∪ range_vars δ ⊆ fv_st (S@Send (Fun f T)#S')"
      using mgu_vars_bounded[OF Unify.hyps(3)[symmetric]] by (metis subset_trans)
    hence "(subst_domain δ ∪ range_vars δ) ∩ bvars_st (S@Send (Fun f T)#S') = {}"
      using Unify.prems unfolding wf_constr_def by blast
    thus ?case
      using wf_unify[OF _ Unify.hyps(2) MGU_is_Unifier[OF mgu_gives_MGU], of "{}",
                     OF _ Unify.hyps(3)[symmetric], of S'] Unify.prems(1)
      by (auto simp add: wf_constr_def)
  next
    case (Equality S δ t t' a S' ϑ)
    have "fv t ∪ fv t' ⊆ fv_st (S@Equality a t t'#S')" using Equality.hyps(2) by force
    hence "subst_domain δ ∪ range_vars δ ⊆ fv_st (S@Equality a t t'#S')"
      using mgu_vars_bounded[OF Equality.hyps(2)[symmetric]] by (metis subset_trans)
    hence "(subst_domain δ ∪ range_vars δ) ∩ bvars_st (S@Equality a t t'#S') = {}"
      using Equality.prems unfolding wf_constr_def by blast
    thus ?case
      using wf_equality[OF _ Equality.hyps(2)[symmetric], of "{}" S a S'] Equality.prems(1)
      by (auto simp add: wf_constr_def)
  qed (metis wf_send_compose wf_constr_def)

  show ?thesis using assms
  proof (induction rule: rtrancl_induct2)
    case (step S_i ϑ_i S_j ϑ_j) thus ?case
      using LI_preserves_subst_wf_single[OF ‹(S_i,ϑ_i) ⇝ (S_j,ϑ_j)›] *[OF ‹(S_i,ϑ_i) ⇝ (S_j,ϑ_j)›]
      by (metis wf_constr_def)
  qed simp
qed

lemma LI_preserves_trm_wf:
  assumes "(S,ϑ) ⇝* (S',ϑ')" "wf_trms (trms_st S)"
  shows "wf_trms (trms_st S')"
proof -
  { fix S ϑ S' ϑ'
    assume "(S,ϑ) ⇝ (S',ϑ')" "wf_trms (trms_st S)"
    hence "wf_trms (trms_st S')"
    proof (induction rule: LI_rel.induct)
      case (Compose S T f S' ϑ)
      hence "wf_trm (Fun f T)"
```

```
      and *: "t ∈ set S ⟹ wf_trms (trms_stp t)" "t ∈ set S' ⟹ wf_trms (trms_stp t)" for t
      by auto
    hence "wf_trm t" when "t ∈ set T" for t using that unfolding wf_trm_def by auto
    hence "wf_trms (trms_stp t)" when "t ∈ set (map Send T)" for t
      using that unfolding wf_trm_def by auto
    thus ?case using * by force
  next
    case (Unify S f U δ T S' ϑ)
    have "wf_trm (Fun f T)" "wf_trm (Fun f U)"
      using Unify.prems(1) Unify.hyps(2) wf_trm_subterm[of _ "Fun f U"]
      by (simp, force)
    hence range_wf: "wf_trms (subst_range δ)"
      using mgu_wf_trm[OF Unify.hyps(3)[symmetric]] by simp

    { fix s assume "s ∈ set (S@S' ·_st δ)"
      hence "∃s' ∈ set (S@S'). s = s' ·_stp δ ∧ wf_trms (trms_stp s')"
        using Unify.prems(1) by (auto simp add: subst_apply_strand_def)
      moreover {
        fix s' assume s': "s = s' ·_stp δ" "wf_trms (trms_stp s')" "s' ∈ set (S@S')"
        from s'(2) have "trms_stp (s' ·_stp δ) = trms_stp s' ·_set (rm_vars (set (bvars_stp s')) δ)"
        proof (induction s')
          case (Inequality X F) thus ?case by (induct F) (auto simp add: subst_apply_pairs_def)
        qed auto
        hence "wf_trms (trms_stp s)"
          using wf_trm_subst[OF wf_trms_subst_rm_vars'[OF range_wf]] ⟨wf_trms (trms_stp s')⟩ s'(1)
          by simp
      }
      ultimately have "wf_trms (trms_stp s)" by auto
    }
    thus ?case by auto
  next
    case (Equality S δ t t' a S' ϑ)
    hence "wf_trm t" "wf_trm t'" by simp_all
    hence range_wf: "wf_trms (subst_range δ)"
      using mgu_wf_trm[OF Equality.hyps(2)[symmetric]] by simp

    { fix s assume "s ∈ set (S@S' ·_st δ)"
      hence "∃s' ∈ set (S@S'). s = s' ·_stp δ ∧ wf_trms (trms_stp s')"
        using Equality.prems(1) by (auto simp add: subst_apply_strand_def)
      moreover {
        fix s' assume s': "s = s' ·_stp δ" "wf_trms (trms_stp s')" "s' ∈ set (S@S')"
        from s'(2) have "trms_stp (s' ·_stp δ) = trms_stp s' ·_set (rm_vars (set (bvars_stp s')) δ)"
        proof (induction s')
          case (Inequality X F) thus ?case by (induct F) (auto simp add: subst_apply_pairs_def)
        qed auto
        hence "wf_trms (trms_stp s)"
          using wf_trm_subst[OF wf_trms_subst_rm_vars'[OF range_wf]] ⟨wf_trms (trms_stp s')⟩ s'(1)
          by simp
      }
      ultimately have "wf_trms (trms_stp s)" by auto
    }
    thus ?case by auto
  qed
}
with assms show ?thesis by (induction rule: rtrancl_induct2) metis+
qed
end
```

3.2.4 Theorem: Soundness of the Lazy Intruder

```
context
begin
private lemma LI_soundness_single:
```

```
  assumes "wf_constr S₁ ϑ₁" "(S₁,ϑ₁) ⇝ (S₂,ϑ₂)" "ℐ ⊨c ⟨S₂,ϑ₂⟩"
  shows "ℐ ⊨c ⟨S₁,ϑ₁⟩"
using assms(2,1,3)
proof (induction rule: LI_rel.induct)
  case (Compose S T f S' ϑ)
  hence *: "⟦{}; S⟧c ℐ" "⟦ik_st S ·set ℐ; map Send T⟧c ℐ" "⟦ik_st S ·set ℐ; S'⟧c ℐ"
    unfolding constr_sem_c_def by force+

  have "ik_st S ·set ℐ ⊢c Fun f T · ℐ"
    using *(2) Compose.hyps(2) ComposeC[OF _ Compose.hyps(3), of "map (λx. x · ℐ) T"]
    unfolding subst_compose_def by force
  thus "ℐ ⊨c ⟨S@Send (Fun f T)#S',ϑ⟩"
    using *(1,3) ‹ℐ ⊨c ⟨S@map Send T@S',ϑ⟩›
    by (auto simp add: constr_sem_c_def)
next
  case (Unify S f U δ T S' ϑ)
  have "(ϑ ∘s δ) supports ℐ" "⟦{}; S@S' ·st δ⟧c ℐ"
    using Unify.prems(2) unfolding constr_sem_c_def by metis+
  then obtain σ where σ: "ϑ ∘s δ ∘s σ = ℐ" unfolding subst_compose_def by auto

  have ϑfun_id: "Fun f U · ϑ = Fun f U" "Fun f T · ϑ = Fun f T"
    using Unify.prems(1) trm_subst_ident[of "Fun f U" ϑ]
          fv_subset_if_in_strand_ik[of "Fun f U" S] Unify.hyps(2)
          fv_snd_rcv_strand_subset(2)[of S]
          strand_vars_split(1)[of S "Send (Fun f T)#S'"]
    unfolding wf_constr_def apply blast
    using Unify.prems(1) trm_subst_ident[of "Fun f T" ϑ]
    unfolding wf_constr_def by fastforce
  hence ϑδ_disj:
      "subst_domain ϑ ∩ subst_domain δ = {}"
      "subst_domain ϑ ∩ range_vars δ = {}"
      "subst_domain ϑ ∩ range_vars ϑ = {}"
    using trm_subst_disj mgu_vars_bounded[OF Unify.hyps(3)[symmetric]] apply (blast,blast)
    using Unify.prems(1) unfolding wf_constr_def wf_subst_def by blast
  hence ϑδ_support: "ϑ supports ℐ" "δ supports ℐ"
    by (simp_all add: subst_support_comp_split[OF ‹(ϑ ∘s δ) supports ℐ›])

  have "fv (Fun f T) ⊆ fv_st (S@Send (Fun f T)#S')" "fv (Fun f U) ⊆ fv_st (S@Send (Fun f T)#S')"
    using Unify.hyps(2) by force+
  hence δ_vars_bound: "subst_domain δ ∪ range_vars δ ⊆ fv_st (S@Send (Fun f T)#S')"
    using mgu_vars_bounded[OF Unify.hyps(3)[symmetric]] by blast

  have "⟦ik_st S ·set ℐ; [Send (Fun f T)]⟧c ℐ"
  proof -
    from Unify.hyps(2) have "Fun f U · ℐ ∈ ik_st S ·set ℐ" by blast
    hence "Fun f U · ℐ ∈ ik_st S ·set ℐ" by blast
    moreover have "Unifier δ (Fun f T) (Fun f U)"
      by (fact MGU_is_Unifier[OF mgu_gives_MGU[OF Unify.hyps(3)[symmetric]]])
    ultimately have "Fun f T · ℐ ∈ ik_st S ·set ℐ"
      using σ by (metis ϑfun_id subst_subst_compose)
    thus ?thesis by simp
  qed

  have "⟦{}; S⟧c ℐ" "⟦ik_st S ·set ℐ; S'⟧c ℐ"
  proof -
    have "(S@S' ·st δ) ·st ϑ = S@S' ·st δ" "(S@S') ·st ϑ = S@S'"
    proof -
      have "subst_domain ϑ ∩ vars_st (S@S') = {}"
        using Unify.prems(1) by (auto simp add: wf_constr_def)
      hence "subst_domain ϑ ∩ vars_st (S@S' ·st δ) = {}"
        using ϑδ_disj(2) strand_subst_vars_union_bound[of "S@S'" δ] by blast
      thus "(S@S' ·st δ) ·st ϑ = S@S' ·st δ" "(S@S') ·st ϑ = S@S'"
        using strand_subst_comp ‹subst_domain ϑ ∩ vars_st (S@S') = {}› by (blast,blast)
```

```
    qed
    moreover have "subst_idem δ" by (fact mgu_gives_subst_idem[OF Unify.hyps(3)[symmetric]])
    moreover have
        "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st (S@S') = {}"
        "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st (S@S' ·_st δ) = {}"
        "(subst_domain δ ∪ range_vars δ) ∩ bvars_st (S@S') = {}"
      using wf_constr_bvars_disj[OF Unify.prems(1)]
            wf_constr_bvars_disj'[OF Unify.prems(1) δ_vars_bound]
      by auto
    ultimately have "⟦{}; S@S'⟧_c ℐ"
      using ‹⟦{}; S@S' ·_st δ⟧_c ℐ› σ
            strand_sem_subst(1)[of ϑ "S@S' ·_st δ" "{}" "δ ∘_s σ"]
            strand_sem_subst(2)[of ϑ "S@S'" "{}" "δ ∘_s σ"]
            strand_sem_subst_subst_idem[of δ "S@S'" "{}" σ]
      unfolding constr_sem_c_def
      by (metis subst_compose_assoc)
    thus "⟦{}; S⟧_c ℐ" "⟦ik_st S ·_set ℐ; S'⟧_c ℐ" by auto
  qed

  show "ℐ ⊨_c ⟨S@Send (Fun f T)#S',ϑ⟩"
    using ϑδ_support(1) ‹⟦ik_st S ·_set ℐ; [Send (Fun f T)]⟧_c ℐ› ‹⟦{}; S⟧_c ℐ› ‹⟦ik_st S ·_set ℐ; S'⟧_c ℐ›
    by (auto simp add: constr_sem_c_def)
next
  case (Equality S δ t t' a S' ϑ)
  have "(ϑ ∘_s δ) supports ℐ" "⟦{}; S@S' ·_st δ⟧_c ℐ"
    using Equality.prems(2) unfolding constr_sem_c_def by metis+
  then obtain σ where σ: "ϑ ∘_s δ ∘_s σ = ℐ" unfolding subst_compose_def by auto

  have "fv t ⊆ vars_st (S@Equality a t t'#S')" "fv t' ⊆ vars_st (S@Equality a t t'#S')"
    by auto
  moreover have "subst_domain ϑ ∩ vars_st (S@Equality a t t'#S') = {}"
    using Equality.prems(1) unfolding wf_constr_def by auto
  ultimately have ϑfun_id: "t · ϑ = t" "t' · ϑ = t'"
    using trm_subst_ident[of t ϑ] trm_subst_ident[of t' ϑ]
    by auto
  hence ϑδ_disj:
      "subst_domain ϑ ∩ subst_domain δ = {}"
      "subst_domain ϑ ∩ range_vars δ = {}"
      "subst_domain ϑ ∩ range_vars ϑ = {}"
    using trm_subst_disj mgu_vars_bounded[OF Equality.hyps(2)[symmetric]] apply (blast,blast)
    using Equality.prems(1) unfolding wf_constr_def wf_subst_def by blast
  hence ϑδ_support: "ϑ supports ℐ" "δ supports ℐ"
    by (simp_all add: subst_support_comp_split[OF ‹(ϑ ∘_s δ) supports ℐ›])

  have "fv t ⊆ fv_st (S@Equality a t t'#S')" "fv t' ⊆ fv_st (S@Equality a t t'#S')" by auto
  hence δ_vars_bound: "subst_domain δ ∪ range_vars δ ⊆ fv_st (S@Equality a t t'#S')"
    using mgu_vars_bounded[OF Equality.hyps(2)[symmetric]] by blast

  have "⟦ik_st S ·_set ℐ; [Equality a t t']⟧_c ℐ"
  proof -
    have "t · δ = t' · δ"
      using MGU_is_Unifier[OF mgu_gives_MGU[OF Equality.hyps(2)[symmetric]]]
      by metis
    hence "t · (ϑ ∘_s δ) = t' · (ϑ ∘_s δ)" by (metis ϑfun_id subst_subst_compose)
    hence "t · ℐ = t' · ℐ" by (metis σ subst_subst_compose)
    thus ?thesis by simp
  qed

  have "⟦{}; S⟧_c ℐ" "⟦ik_st S ·_set ℐ; S'⟧_c ℐ"
  proof -
    have "(S@S' ·_st δ) ·_st ϑ = S@S' ·_st δ" "(S@S') ·_st ϑ = S@S'"
    proof -
      have "subst_domain ϑ ∩ vars_st (S@S') = {}"
```

```
        using Equality.prems(1)
        by (fastforce simp add: wf_constr_def simp del: subst_range.simps)
      hence "subst_domain ϑ ∩ fv_st (S@S') = {}" by blast
      hence "subst_domain ϑ ∩ fv_st (S@S' ·_st δ) = {}"
        using ϑδ_disj(2) subst_sends_strand_fv_to_img[of "S@S'" δ] by blast
      thus "(S@S' ·_st δ) ·_st ϑ = S@S' ·_st δ" "(S@S') ·_st ϑ = S@S'"
        using strand_subst_comp ‹subst_domain ϑ ∩ vars_st (S@S') = {}› by (blast,blast)
    qed
    moreover have
        "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st (S@S') = {}"
        "(subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st (S@S' ·_st δ) = {}"
        "(subst_domain δ ∪ range_vars δ) ∩ bvars_st (S@S') = {}"
      using wf_constr_bvars_disj[OF Equality.prems(1)]
            wf_constr_bvars_disj'[OF Equality.prems(1) δ_vars_bound]
      by auto
    ultimately have "⟦{}; S@S'⟧_c ℐ"
      using ‹⟦{}; S@S' ·_st δ⟧_c ℐ› σ
            strand_sem_subst(1)[of ϑ "S@S' ·_st δ" "{}" "δ ∘_s σ"]
            strand_sem_subst(2)[of ϑ "S@S'" "{}" "δ ∘_s σ"]
            strand_sem_subst_subst_idem[of δ "S@S'" "{}" σ]
            mgu_gives_subst_idem[OF Equality.hyps(2)[symmetric]]
      unfolding constr_sem_c_def
      by (metis subst_compose_assoc)
    thus "⟦{}; S⟧_c ℐ" "⟦ik_st S ·_set ℐ; S'⟧_c ℐ" by auto
  qed

  show "ℐ ⊨_c ⟨S@Equality a t t'#S',ϑ⟩"
    using ϑδ_support(1) ‹⟦ik_st S ·_set ℐ; [Equality a t t']⟧_c ℐ› ‹⟦{}; S⟧_c ℐ› ‹⟦ik_st S ·_set ℐ; S'⟧_c ℐ›
    by (auto simp add: constr_sem_c_def)
qed

theorem LI_soundness:
  assumes "wf_constr S1 ϑ1" "(S1,ϑ1) ⇝* (S2,ϑ2)" "ℐ ⊨_c ⟨S2, ϑ2⟩"
  shows "ℐ ⊨_c ⟨S1, ϑ1⟩"
using assms(2,1,3)
proof (induction S2 ϑ2 rule: rtrancl_induct2)
  case (step Si ϑi Sj ϑj) thus ?case
    using LI_preserves_wellformedness[OF ‹(S1, ϑ1) ⇝* (Si, ϑi)› ‹wf_constr S1 ϑ1›]
          LI_soundness_single[OF _ ‹(Si, ϑi) ⇝ (Sj, ϑj)› ‹ℐ ⊨_c ⟨Sj, ϑj⟩›]
          step.IH[OF ‹wf_constr S1 ϑ1›]
    by metis
qed metis
end
```

3.2.5 Theorem: Completeness of the Lazy Intruder

```
context
begin
private lemma LI_completeness_single:
  assumes "wf_constr S1 ϑ1" "ℐ ⊨_c ⟨S1, ϑ1⟩" "¬simple S1"
  shows "∃S2 ϑ2. (S1,ϑ1) ⇝ (S2,ϑ2) ∧ (ℐ ⊨_c ⟨S2, ϑ2⟩)"
using not_simple_elim[OF ‹¬simple S1›]
proof -
  { — In this case S1 isn't simple because it contains an equality constraint, so we can simply proceed with the reduction
by computing the MGU for the equation
    assume "∃S' S'' a t t'. S1 = S'@Equality a t t'#S'' ∧ simple S'"
    then obtain S a t t' S' where S1: "S1 = S@Equality a t t'#S'" "simple S" by moura
    hence *: "wf_st {} S" "ℐ ⊨_c ⟨S, ϑ1⟩" "ϑ1 supports ℐ" "t · ℐ = t' · ℐ"
      using ‹ℐ ⊨_c ⟨S1, ϑ1⟩› ‹wf_constr S1 ϑ1› wf_eq_fv[of "{}" S t t' S']
            fv_snd_rcv_strand_subset(5)[of S]
      by (auto simp add: constr_sem_c_def wf_constr_def)

    from * have "Unifier ℐ t t'" by simp
```

```
then obtain δ where δ:
    "Some δ = mgu t t'" "subst_idem δ" "subst_domain δ ∪ range_vars δ ⊆ fv t ∪ fv t'"
  using mgu_always_unifies mgu_gives_subst_idem mgu_vars_bounded by metis+

have "δ ≼∘ ℐ"
  using mgu_gives_MGU[OF δ(1)[symmetric]]
  by (metis ‹Unifier ℐ t t'›)
hence "δ supports ℐ" using subst_support_if_mgt_subst_idem[OF _ δ(2)] by metis
hence "(ϑ1 ∘s δ) supports ℐ" using subst_support_comp ‹ϑ1 supports ℐ› by metis

have "⟦{}; S@S' ·st δ⟧c ℐ"
proof -
  have "subst_domain δ ∪ range_vars δ ⊆ fvst S1" using δ(3) S1(1) by auto
  hence "⟦{}; S1 ·st δ⟧c ℐ"
    using ‹subst_idem δ› ‹δ ≼∘ ℐ› ‹ℐ ⊨c ⟨S1, ϑ1⟩› strand_sem_subst
          wf_constr_bvars_disj'(1)[OF assms(1)]
    unfolding subst_idem_def constr_sem_c_def
    by (metis (no_types) subst_compose_assoc)
  thus "⟦{}; S@S' ·st δ⟧c ℐ" using S1(1) by force
qed
moreover have "(S@Equality a t t'#S', ϑ1) ⇝ (S@S' ·st δ, ϑ1 ∘s δ)"
  using LI_rel.Equality[OF ‹simple S› δ(1)] S1 by metis
ultimately have ?thesis
  using S1(1) ‹(ϑ1 ∘s δ) supports ℐ›
  by (auto simp add: constr_sem_c_def)
} moreover {
```

— In this case S_1 isn't simple because it contains a deduction constraint for a composed term, so we must look at how this composed term is derived under the interpretation $\mathcal{I}$

```
assume "∃S' S'' f T. S1 = S'@Send (Fun f T)#S'' ∧ simple S'"
with assms obtain S f T S' where S1: "S1 = S@Send (Fun f T)#S'" "simple S" by moura
hence "wfst {} S" "ℐ ⊨c ⟨S, ϑ1⟩" "ϑ1 supports ℐ"
  using ‹ℐ ⊨c ⟨S1, ϑ1⟩› ‹wfconstr S1 ϑ1›
  by (auto simp add: constr_sem_c_def wfconstr_def)
```

— Lemma for a common subcase

```
have fun_sat: "ℐ ⊨c ⟨S@(map Send T)@S', ϑ1⟩" when T: "⋀t. t ∈ set T ⟹ ikst S ·set ℐ ⊢c t · ℐ"
proof -
  have "⋀t. t ∈ set T ⟹ ⟦ikst S ·set ℐ; [Send t]⟧c ℐ" using T by simp
  hence "⟦ikst S ·set ℐ; map Send T⟧c ℐ" using ‹ℐ ⊨c ⟨S1, ϑ1⟩› strand_sem_Send_map by metis
  moreover have "⟦ikst (S@(map Send T)) ·set ℐ; S'⟧c ℐ"
    using ‹ℐ ⊨c ⟨S1, ϑ1⟩› S1
    by (auto simp add: constr_sem_c_def)
  ultimately show ?thesis
    using ‹ℐ ⊨c ⟨S, ϑ1⟩› ‹ℐ ⊨c ⟨S1, ϑ1⟩›
    by (force simp add: constr_sem_c_def)
qed

from S1 ‹ℐ ⊨c ⟨S1, ϑ1⟩› have "ikst S ·set ℐ ⊢c Fun f T · ℐ" by (auto simp add: constr_sem_c_def)
hence ?thesis
proof cases
```

— Case 1: `ℐ(f(T))` has been derived using the `AxiomC` rule.

```
  case AxiomC
  hence ex_t: "∃t. t ∈ ikst S ∧ Fun f T · ℐ = t · ℐ" by auto
  show ?thesis
  proof (cases "∀T'. Fun f T' ∈ ikst S ⟶ Fun f T · ℐ ≠ Fun f T' · ℐ")
```

— Case 1.1: `f(T)` is equal to a variable in the intruder knowledge under $\mathcal{I}$. Hence there must exists a deduction constraint in the simple prefix of the constraint in which this variable occurs/"is sent" for the first time. Since this variable itself cannot have been derived from the `AxiomC` rule (because it must be equal under the interpretation to `f(T)`, which is by assumption not in the intruder knowledge under $\mathcal{I}$) it must be the case that we can derive it using the `ComposeC` rule. Hence we can apply the `Compose` rule of the lazy intruder to `f(T)`.

```
    case True
    have "∃v. Var v ∈ ikst S ∧ Fun f T · ℐ = ℐ v"
    proof -
```

```
        obtain t where "t ∈ ik_st S" "Fun f T · ℐ = t · ℐ" using ex_t by moura
        thus ?thesis
          using ⟨∀T'. Fun f T' ∈ ik_st S ⟶ Fun f T · ℐ ≠ Fun f T' · ℐ⟩
          by (cases t) auto
      qed
      hence "∃v ∈ wfrestrictedvars_st S. Fun f T · ℐ = ℐ v"
        using vars_subset_if_in_strand_ik2[of _ S] by fastforce
      then obtain v S_pre S_suf
        where S: "S = S_pre@Send (Var v)#S_suf" "Fun f T · ℐ = ℐ v"
                 "¬(∃w ∈ wfrestrictedvars_st S_pre. Fun f T · ℐ = ℐ w)"
        using ⟨wf_st {} S⟩ wf_simple_strand_first_Send_var_split[OF _ ⟨simple S⟩, of "Fun f T" ℐ]
        by auto
      hence "∀w. Var w ∈ ik_st S_pre ⟶ ℐ v ≠ Var w · ℐ" by auto
      moreover have "∀T'. Fun f T' ∈ ik_st S_pre ⟶ Fun f T · ℐ ≠ Fun f T' · ℐ"
        using ⟨∀T'. Fun f T' ∈ ik_st S ⟶ Fun f T · ℐ ≠ Fun f T' · ℐ⟩ S(1)
        by (meson contra_subsetD ik_append_subset(1))
      hence "∀g T'. Fun g T' ∈ ik_st S_pre ⟶ ℐ v ≠ Fun g T' · ℐ" using S(2) by simp
      ultimately have "∀t ∈ ik_st S_pre. ℐ v ≠ t · ℐ" by (metis term.exhaust)
      hence "ℐ v ∉ (ik_st S_pre) ·_set ℐ" by auto

      have "ik_st S_pre ·_set ℐ ⊢_c ℐ v"
        using S_1(1) S(1) ⟨ℐ ⊨_c ⟨S_1, ϑ_1⟩⟩
        by (auto simp add: constr_sem_c_def)
      hence "ik_st S_pre ·_set ℐ ⊢_c Fun f T · ℐ" using ⟨Fun f T · ℐ = ℐ v⟩ by metis
      hence "length T = arity f" "public f" "⋀t. t ∈ set T ⟹ ik_st S_pre ·_set ℐ ⊢_c t · ℐ"
        using ⟨Fun f T · ℐ = ℐ v⟩ ⟨ℐ v ∉ ik_st S_pre ·_set ℐ⟩
              intruder_synth.simps[of "ik_st S_pre ·_set ℐ" "ℐ v"]
        by auto
      hence *: "⋀t. t ∈ set T ⟹ ik_st S ·_set ℐ ⊢_c t · ℐ"
        using S(1) by (auto intro: ideduct_synth_mono)
      hence "ℐ ⊨_c ⟨S@(map Send T)@S', ϑ_1⟩" by (metis fun_sat)
      moreover have "(S@Send (Fun f T)#S', ϑ_1) ⇝ (S@map Send T@S', ϑ_1)"
        by (metis LI_rel.Compose[OF ⟨simple S⟩ ⟨length T = arity f⟩ ⟨public f⟩])
      ultimately show ?thesis using S_1 by auto
    next
```

— Case 1.2: ℐ(f(T)) can be derived from an interpreted composed term in the intruder knowledge. Use the `Unify` rule on this composed term to further reduce the constraint.

```
      case False
      then obtain T' where t: "Fun f T' ∈ ik_st S" "Fun f T · ℐ = Fun f T' · ℐ"
        by auto
      hence "fv (Fun f T') ⊆ fv_st S_1"
        using S_1(1) fv_subset_if_in_strand_ik'[OF t(1)]
              fv_snd_rcv_strand_subset(2)[of S]
        by auto
      from t have "Unifier ℐ (Fun f T) (Fun f T')" by simp
      then obtain δ where δ:
          "Some δ = mgu (Fun f T) (Fun f T')" "subst_idem δ"
          "subst_domain δ ∪ range_vars δ ⊆ fv (Fun f T) ∪ fv (Fun f T')"
        using mgu_always_unifies mgu_gives_subst_idem mgu_vars_bounded by metis+

      have "δ ≼∘ ℐ"
        using mgu_gives_MGU[OF δ(1)[symmetric]]
        by (metis ⟨Unifier ℐ (Fun f T) (Fun f T')⟩)
      hence "δ supports ℐ" using subst_support_if_mgt_subst_idem[OF _ δ(2)] by metis
      hence "(ϑ_1 ∘_s δ) supports ℐ" using subst_support_comp ⟨ϑ_1 supports ℐ⟩ by metis

      have "⟦{}; S@S' ·_st δ⟧_c ℐ"
      proof -
        have "subst_domain δ ∪ range_vars δ ⊆ fv_st S_1"
          using δ(3) S_1(1) ⟨fv (Fun f T') ⊆ fv_st S_1⟩
          unfolding range_vars_alt_def by (fastforce simp del: subst_range.simps)
        hence "⟦{}; S_1 ·_st δ⟧_c ℐ"
          using ⟨subst_idem δ⟩ ⟨δ ≼∘ ℐ⟩ ⟨ℐ ⊨_c ⟨S_1, ϑ_1⟩⟩ strand_sem_subst
```

```
            wf_constr_bvars_disj'(1)[OF assms(1)]
        unfolding subst_idem_def constr_sem_c_def
        by (metis (no_types) subst_compose_assoc)
      thus "⟦{}; S@S' ·st δ⟧c ℐ" using S1(1) by force
    qed
    moreover have "(S@Send (Fun f T)#S', ϑ1) ⇝ (S@S' ·st δ, ϑ1 ∘s δ)"
      using LI_rel.Unify[OF ‹simple S› t(1) δ(1)] S1 by metis
    ultimately show ?thesis
      using S1(1) ‹(ϑ1 ∘s δ) supports ℐ›
      by (auto simp add: constr_sem_c_def)
  qed
next
```

— Case 2: ℐ(f(T)) has been derived using the `ComposeC` rule. Simply use the `Compose` rule of the lazy intruder to proceed with the reduction.

```
  case (ComposeC T' g)
  hence "f = g" "length T = arity f" "public f"
    and "⋀x. x ∈ set T ⟹ ikst S ·set ℐ ⊢c x · ℐ"
    by auto
  hence "ℐ ⊨c ⟨S@(map Send T)@S', ϑ1⟩" using fun_sat by metis
  moreover have "(S1, ϑ1) ⇝ (S@(map Send T)@S', ϑ1)"
    using S1 LI_rel.Compose[OF ‹simple S› ‹length T = arity f› ‹public f›]
    by metis
  ultimately show ?thesis by metis
 qed
} moreover have "⋀A B X F. S1 = A@Inequality X F#B ⟹ ineq_model ℐ X F"
  using assms(2) by (auto simp add: constr_sem_c_def)
 ultimately show ?thesis using not_simple_elim[OF ‹¬simple S1›] by metis
qed

theorem LI_completeness:
 assumes "wfconstr S1 ϑ1" "ℐ ⊨c ⟨S1, ϑ1⟩"
 shows "∃S2 ϑ2. (S1,ϑ1) ⇝* (S2,ϑ2) ∧ simple S2 ∧ (ℐ ⊨c ⟨S2, ϑ2⟩)"
proof (cases "simple S1")
 case False
 let ?Stuck = "λS2 ϑ2. ¬(∃S3 ϑ3. (S2,ϑ2) ⇝ (S3,ϑ3) ∧ (ℐ ⊨c ⟨S3, ϑ3⟩))"
 let ?Sats = "{((S,ϑ),(S',ϑ')). (S,ϑ) ⇝ (S',ϑ') ∧ (ℐ ⊨c ⟨S, ϑ⟩) ∧ (ℐ ⊨c ⟨S', ϑ'⟩)}"

 have simple_if_stuck:
    "⋀S2 ϑ2. ⟦(S1,ϑ1) ⇝+ (S2,ϑ2); ℐ ⊨c ⟨S2, ϑ2⟩; ?Stuck S2 ϑ2⟧ ⟹ simple S2"
  using LI_completeness_single
        LI_preserves_wellformedness
        ‹wfconstr S1 ϑ1›
        trancl_into_rtrancl
  by metis

 have base: "∃b. ((S1,ϑ1),b) ∈ ?Sats"
  using LI_completeness_single[OF assms False] assms(2)
  by auto

 have *: "⋀S ϑ S' ϑ'. ((S,ϑ),(S',ϑ')) ∈ ?Sats+ ⟹ (S,ϑ) ⇝+ (S',ϑ') ∧ (ℐ ⊨c ⟨S', ϑ'⟩)"
 proof -
  fix S ϑ S' ϑ'
  assume "((S,ϑ),(S',ϑ')) ∈ ?Sats+"
  thus "(S,ϑ) ⇝+ (S',ϑ') ∧ (ℐ ⊨c ⟨S', ϑ'⟩)"
    by (induct rule: trancl_induct2) auto
 qed

 have "∃S2 ϑ2. ((S1,ϑ1),(S2,ϑ2)) ∈ ?Sats+ ∧ ?Stuck S2 ϑ2"
 proof (rule ccontr)
  assume "¬(∃S2 ϑ2. ((S1,ϑ1),(S2,ϑ2)) ∈ ?Sats+ ∧ ?Stuck S2 ϑ2)"
  hence sat_not_stuck: "⋀S2 ϑ2. ((S1,ϑ1),(S2,ϑ2)) ∈ ?Sats+ ⟹ ¬?Stuck S2 ϑ2" by blast

  have "∀S ϑ. ((S1,ϑ1),(S,ϑ)) ∈ ?Sats+ ⟶ (∃b. ((S,ϑ),b) ∈ ?Sats)"
```

```
    proof (intro allI impI)
      fix S ϑ assume a: "((S₁,ϑ₁),(S,ϑ)) ∈ ?Sats⁺"
      have "⋀b. ((S₁,ϑ₁),b) ∈ ?Sats⁺ ⟹ ∃c. b ⇝ c ∧ ((S₁,ϑ₁),c) ∈ ?Sats⁺"
      proof -
        fix b assume in_sat: "((S₁,ϑ₁),b) ∈ ?Sats⁺"
        hence "∃c. (b,c) ∈ ?Sats" using * sat_not_stuck by (cases b) blast
        thus "∃c. b ⇝ c ∧ ((S₁,ϑ₁),c) ∈ ?Sats⁺"
          using trancl_into_trancl[OF in_sat] by blast
      qed
      hence "∃S' ϑ'. (S,ϑ) ⇝ (S',ϑ') ∧ ((S₁,ϑ₁),(S',ϑ')) ∈ ?Sats⁺" using a by auto
      then obtain S' ϑ' where S'ϑ': "(S,ϑ) ⇝ (S',ϑ')" "((S₁,ϑ₁),(S',ϑ')) ∈ ?Sats⁺" by auto
      hence "ℐ ⊨c ⟨S', ϑ'⟩" using * by blast
      moreover have "(S₁, ϑ₁) ⇝⁺ (S,ϑ)" using a trancl_mono by blast
      ultimately have "((S,ϑ),(S',ϑ')) ∈ ?Sats" using S'ϑ'(1) * a by blast
      thus "∃b. ((S,ϑ),b) ∈ ?Sats" using S'ϑ'(2) by blast
    qed
    hence "∃f. ∀i::nat. (f i, f (Suc i)) ∈ ?Sats"
      using infinite_chain_intro'[OF base] by blast
    moreover have "?Sats ⊆ LI_rel⁺" by auto
    hence "¬(∃f. ∀i::nat. (f i, f (Suc i)) ∈ ?Sats)"
      using LI_no_infinite_chain infinite_chain_mono by blast
    ultimately show False by auto
  qed
  hence "∃S₂ ϑ₂. (S₁, ϑ₁) ⇝⁺ (S₂, ϑ₂) ∧ simple S₂ ∧ (ℐ ⊨c ⟨S₂, ϑ₂⟩)"
    using simple_if_stuck * by blast
  thus ?thesis by (meson trancl_into_rtrancl)
qed (blast intro: ‹ℐ ⊨c ⟨S₁, ϑ₁⟩›)
end
```

3.2.6 Corollary: Soundness and Completeness as a Single Theorem

```
corollary LI_soundness_and_completeness:
  assumes "wf_constr S₁ ϑ₁"
  shows "ℐ ⊨c ⟨S₁, ϑ₁⟩ ⟷ (∃S₂ ϑ₂. (S₁,ϑ₁) ⇝* (S₂,ϑ₂) ∧ simple S₂ ∧ (ℐ ⊨c ⟨S₂, ϑ₂⟩))"
by (metis LI_soundness[OF assms] LI_completeness[OF assms])

end

end
```

3.3 The Typed Model (Typed_Model)

```
theory Typed_Model
imports Lazy_Intruder
begin
```

Term types

```
type_synonym ('f,'v) term_type = "('f,'v) term"
```

Constructors for term types

```
abbreviation (input) TAtom::"'v ⇒ ('f,'v) term_type" where
  "TAtom a ≡ Var a"

abbreviation (input) TComp::"['f, ('f,'v) term_type list] ⇒ ('f,'v) term_type" where
  "TComp f T ≡ Fun f T"
```

The typed model extends the intruder model with a typing function Γ that assigns types to terms.

```
locale typed_model = intruder_model arity public Ana
  for arity::"'fun ⇒ nat"
    and public::"'fun ⇒ bool"
    and Ana::"('fun,'var) term ⇒ (('fun,'var) term list × ('fun,'var) term list)"
  +
```

```
  fixes Γ::"('fun,'var) term ⇒ ('fun,'atom::finite) term_type"
  assumes const_type: "⋀c. arity c = 0 ⟹ ∃a. ∀T. Γ (Fun c T) = TAtom a"
    and fun_type: "⋀f T. arity f > 0 ⟹ Γ (Fun f T) = TComp f (map Γ T)"
    and infinite_typed_consts: "⋀a. infinite {c. Γ (Fun c []) = TAtom a ∧ public c}"
    and Γ_wf: "⋀t f T. TComp f T ⊑ Γ t ⟹ arity f > 0"
              "⋀x. wf_trm (Γ (Var x))"
    and no_private_funs[simp]: "⋀f. arity f > 0 ⟹ public f"
begin
```

3.3.1 Definitions

The set of atomic types

```
abbreviation "𝔗_a ≡ UNIV::('atom set)"
```

Well-typed substitutions

```
definition wt_subst where
  "wt_subst σ ≡ (∀v. Γ (Var v) = Γ (σ v))"
```

The set of sub-message patterns (SMP)

```
inductive_set SMP::"('fun,'var) terms ⇒ ('fun,'var) terms" for M where
  MP[intro]: "t ∈ M ⟹ t ∈ SMP M"
| Subterm[intro]: "⟦t ∈ SMP M; t' ⊑ t⟧ ⟹ t' ∈ SMP M"
| Substitution[intro]: "⟦t ∈ SMP M; wt_subst δ; wf_trms (subst_range δ)⟧ ⟹ (t · δ) ∈ SMP M"
| Ana[intro]: "⟦t ∈ SMP M; Ana t = (K,T); k ∈ set K⟧ ⟹ k ∈ SMP M"
```

Type-flaw resistance for sets: Unifiable sub-message patterns must have the same type (unless they are variables)

```
definition tfr_set where
  "tfr_set M ≡ (∀s ∈ SMP M - (Var`𝒱). ∀t ∈ SMP M - (Var`𝒱). (∃δ. Unifier δ s t) ⟶ Γ s = Γ t)"
```

Type-flaw resistance for strand steps: - The terms in a satisfiable equality step must have the same types - Inequality steps must satisfy the conditions of the "inequality lemma"

```
fun tfr_stp where
  "tfr_stp (Equality a t t') = ((∃δ. Unifier δ t t') ⟶ Γ t = Γ t')"
| "tfr_stp (Inequality X F) = (
      (∀x ∈ fv_pairs F - set X. ∃a. Γ (Var x) = TAtom a) ∨
      (∀f T. Fun f T ∈ subterms_set (trms_pairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)))"
| "tfr_stp _ = True"
```

Type-flaw resistance for strands: - The set of terms in strands must be type-flaw resistant - The steps of strands must be type-flaw resistant

```
definition tfr_st where
  "tfr_st S ≡ tfr_set (trms_st S) ∧ list_all tfr_stp S"
```

3.3.2 Small Lemmata

```
lemma tfr_stp_list_all_alt_def:
  "list_all tfr_stp S ⟷
    ((∀a t t'. Equality a t t' ∈ set S ∧ (∃δ. Unifier δ t t') ⟶ Γ t = Γ t') ∧
    (∀X F. Inequality X F ∈ set S ⟶
      (∀x ∈ fv_pairs F - set X. ∃a. Γ (Var x) = TAtom a)
    ∨ (∀f T. Fun f T ∈ subterms_set (trms_pairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))))"
  (is "?P S ⟷ ?Q S")
proof
  show "?P S ⟹ ?Q S"
  proof (induction S)
    case (Cons x S) thus ?case by (cases x) auto
  qed simp

  show "?Q S ⟹ ?P S"
  proof (induction S)
```

```
    case (Cons x S) thus ?case by (cases x) auto
  qed simp
qed


lemma Γ_wf': "wf_trm t ⟹ wf_trm (Γ t)"
proof (induction t)
  case (Fun f T)
  hence *: "arity f = length T" "⋀t. t ∈ set T ⟹ wf_trm (Γ t)" unfolding wf_trm_def by auto
  { assume "arity f = 0" hence ?case using const_type[of f] by auto }
  moreover
  { assume "arity f > 0" hence ?case using fun_type[of f] * by force }
  ultimately show ?case by auto
qed (metis Γ_wf(2))

lemma fun_type_inv: assumes "Γ t = TComp f T" shows "arity f > 0" "public f"
using Γ_wf(1)[of f T t] assms by simp_all

lemma fun_type_inv_wf: assumes "Γ t = TComp f T" "wf_trm t" shows "arity f = length T"
using Γ_wf'[OF assms(2)] assms(1) unfolding wf_trm_def by auto

lemma const_type_inv: "Γ (Fun c X) = TAtom a ⟹ arity c = 0"
by (rule ccontr, simp add: fun_type)

lemma const_type_inv_wf: assumes "Γ (Fun c X) = TAtom a" and "wf_trm (Fun c X)" shows "X = []"
by (metis assms const_type_inv length_0_conv subtermeqI' wf_trm_def)

lemma const_type': "∀c ∈ 𝒞. ∃a ∈ 𝔗_a. ∀X. Γ (Fun c X) = TAtom a" using const_type by simp
lemma fun_type': "∀f ∈ Σ_f. ∀X. Γ (Fun f X) = TComp f (map Γ X)" using fun_type by simp

lemma infinite_public_consts[simp]: "infinite {c. public c ∧ arity c = 0}"
proof -
  fix a::'atom
  define A where "A ≡ {c. Γ (Fun c []) = TAtom a ∧ public c}"
  define B where "B ≡ {c. public c ∧ arity c = 0}"

  have "arity c = 0" when c: "c ∈ A" for c
    using c const_type_inv unfolding A_def by blast
  hence "A ⊆ B" unfolding A_def B_def by blast
  hence "infinite B"
    using infinite_typed_consts[of a, unfolded A_def[symmetric]]
    by (metis infinite_super)
  thus ?thesis unfolding B_def by blast
qed

lemma infinite_fun_syms[simp]:
  "infinite {c. public c ∧ arity c > 0} ⟹ infinite Σ_f"
  "infinite 𝒞" "infinite 𝒞_pub" "infinite (UNIV::'fun set)"
by (metis Σ_f_unfold finite_Collect_conjI,
    metis infinite_public_consts finite_Collect_conjI,
    use infinite_public_consts 𝒞pub_unfold in ‹force simp add: Collect_conj_eq›,
    metis UNIV_I finite_subset subsetI infinite_public_consts(1))

lemma id_univ_proper_subset[simp]: "Σ_f ⊂ UNIV" "(∃f. arity f > 0) ⟹ 𝒞 ⊂ UNIV"
by (metis finite.emptyI inf_top.right_neutral top.not_eq_extremum disjoint_fun_syms
          infinite_fun_syms(2) inf_commute)
   (metis top.not_eq_extremum UNIV_I const_arity_eq_zero less_irrefl)

lemma exists_fun_notin_funs_term: "∃f::'fun. f ∉ funs_term t"
by (metis UNIV_eq_I finite_fun_symbols infinite_fun_syms(4))

lemma exists_fun_notin_funs_terms:
  assumes "finite M" shows "∃f::'fun. f ∉ ⋃(funs_term ` M)"
```

```
by (metis assms finite_fun_symbols infinite_fun_syms(4) ex_new_if_finite finite_UN)

lemma exists_notin_funsst: "∃f. f ∉ funsst (S::('fun,'var) strand)"
by (metis UNIV_eq_I finite_funsst infinite_fun_syms(4))

lemma infinite_typed_consts': "infinite {c. Γ (Fun c []) = TAtom a ∧ public c ∧ arity c = 0}"
proof -
  { fix c assume "Γ (Fun c []) = TAtom a" "public c"
    hence "arity c = 0" using const_type[of c] fun_type[of c "[]"] by auto
  } hence "{c. Γ (Fun c []) = TAtom a ∧ public c ∧ arity c = 0} =
           {c. Γ (Fun c []) = TAtom a ∧ public c}"
    by auto
  thus "infinite {c. Γ (Fun c []) = TAtom a ∧ public c ∧ arity c = 0}"
    using infinite_typed_consts[of a] by metis
qed

lemma atypes_inhabited: "∃c. Γ (Fun c []) = TAtom a ∧ wftrm (Fun c []) ∧ public c ∧ arity c = 0"
proof -
  obtain c where "Γ (Fun c []) = TAtom a" "public c" "arity c = 0"
    using infinite_typed_consts'(1)[of a] not_finite_existsD by blast
  thus ?thesis using const_type_inv[OF ‹Γ (Fun c []) = TAtom a›] unfolding wftrm_def by auto
qed

lemma atype_ground_term_ex: "∃t. fv t = {} ∧ Γ t = TAtom a ∧ wftrm t"
using atypes_inhabited[of a] by force

lemma fun_type_id_eq: "Γ (Fun f X) = TComp g Y ⟹ f = g"
by (metis const_type fun_type neq0_conv "term.inject"(2) "term.simps"(4))

lemma fun_type_length_eq: "Γ (Fun f X) = TComp g Y ⟹ length X = length Y"
by (metis fun_type fun_type_id_eq fun_type_inv(1) length_map term.inject(2))

lemma type_ground_inhabited: "∃t'. fv t' = {} ∧ Γ t = Γ t'"
proof -
  { fix τ::"('fun, 'atom) term_type" assume "⋀f T. Fun f T ⊑ τ ⟹ 0 < arity f"
    hence "∃t'. fv t' = {} ∧ τ = Γ t'"
    proof (induction τ)
      case (Fun f T)
      hence "arity f > 0" by auto

      from Fun.IH Fun.prems(1) have "∃Y. map Γ Y = T ∧ (∀x ∈ set Y. fv x = {})"
      proof (induction T)
        case (Cons x X)
        hence "⋀g Y. Fun g Y ⊑ Fun f X ⟹ 0 < arity g" by auto
        hence "∃Y. map Γ Y = X ∧ (∀x∈set Y. fv x = {})" using Cons by auto
        moreover have "∃t'. fv t' = {} ∧ x = Γ t'" using Cons by auto
        ultimately obtain y Y where
            "fv y = {}" "Γ y = x" "map Γ Y = X" "∀x∈set Y. fv x = {}"
          using Cons by moura
        hence "map Γ (y#Y) = x#X ∧ (∀x∈set (y#Y). fv x = {})" by auto
        thus ?case by meson
      qed simp
      then obtain Y where "map Γ Y = T" "∀x ∈ set Y. fv x = {}" by metis
      hence "fv (Fun f Y) = {}" "Γ (Fun f Y) = TComp f T" using fun_type[OF ‹arity f > 0›] by auto
      thus ?case by (metis exI[of "λt. fv t = {} ∧ Γ t = TComp f T" "Fun f Y"])
    qed (metis atype_ground_term_ex)
  }
  thus ?thesis by (metis Γ_wf(1))
qed

lemma type_wfttype_inhabited:
  assumes "⋀f T. Fun f T ⊑ τ ⟹ 0 < arity f" "wftrm τ"
  shows "∃t. Γ t = τ ∧ wftrm t"
```

```
using assms
proof (induction τ)
  case (Fun f Y)
  have IH: "∃t. Γ t = y ∧ wf_trm t" when y: "y ∈ set Y " for y
  proof -
    have "wf_trm y"
      using Fun y unfolding wf_trm_def
      by (metis Fun_param_is_subterm term.le_less_trans)
    moreover have "Fun g Z ⊑ y ⟹ 0 < arity g" for g Z
      using Fun y by auto
    ultimately show ?thesis using Fun.IH[OF y] by auto
  qed

  from Fun have "arity f = length Y" "arity f > 0" unfolding wf_trm_def by force+
  moreover from IH have "∃X. map Γ X = Y ∧ (∀x ∈ set X. wf_trm x)"
    by (induct Y, simp_all, metis list.simps(9) set_ConsD)
  ultimately show ?case by (metis fun_type length_map wf_trmI)
qed (use atypes_inhabited wf_trm_def in blast)

lemma type_pgwt_inhabited: "wf_trm t ⟹ ∃t'. Γ t = Γ t' ∧ public_ground_wf_term t'"
proof -
  assume "wf_trm t"
  { fix τ assume "Γ t = τ"
    hence "∃t'. Γ t = Γ t' ∧ public_ground_wf_term t'" using ‹wf_trm t›
    proof (induction τ arbitrary: t)
      case (Var a t)
      then obtain c where "Γ t = Γ (Fun c [])" "arity c = 0" "public c"
        using const_type_inv[of _ "[]" a] infinite_typed_consts(1)[of a]  not_finite_existsD
        by force
      thus ?case using PGWT[OF ‹public c›, of "[]"] by auto
    next
      case (Fun f Y t)
      have *: "arity f > 0" "public f" "arity f = length Y"
        using fun_type_inv[OF ‹Γ t = TComp f Y›] fun_type_inv_wf[OF ‹Γ t = TComp f Y› ‹wf_trm t›]
        by auto
      have "⋀y. y ∈ set Y ⟹ ∃t'. y = Γ t' ∧ public_ground_wf_term t'"
        using Fun.prems(1) Fun.IH Γ_wf(1)[of _ _ t] Γ_wf'[OF ‹wf_trm t›] type_wfttype_inhabited
        by (metis Fun_param_is_subterm term.order_trans wf_trm_subtermeq)
      hence "∃X. map Γ X = Y ∧ (∀x ∈ set X. public_ground_wf_term x)"
        by (induct Y, simp_all, metis list.simps(9) set_ConsD)
      then obtain X where X: "map Γ X = Y" "⋀x. x ∈ set X ⟹ public_ground_wf_term x" by moura
      hence "arity f = length X" using *(3) by auto
      have "Γ t = Γ (Fun f X)" "public_ground_wf_term (Fun f X)"
        using fun_type[OF *(1), of X] Fun.prems(1) X(1) apply simp
        using PGWT[OF *(2) ‹arity f = length X› X(2)] by metis
      thus ?case by metis
    qed
  }
  thus ?thesis using ‹wf_trm t› by auto
qed

lemma pgwt_type_map:
  assumes "public_ground_wf_term t"
  shows "Γ t = TAtom a ⟹ ∃f. t = Fun f []" "Γ t = TComp g Y ⟹ ∃X. t = Fun g X ∧ map Γ X = Y"
proof -
  let ?A = "Γ t = TAtom a ⟶ (∃f. t = Fun f [])"
  let ?B = "Γ t = TComp g Y ⟶ (∃X. t = Fun g X ∧ map Γ X = Y)"
  have "?A ∧ ?B"
  proof (cases "Γ t")
    case (Var a)
    obtain f X where "t = Fun f X" "arity f = length X"
      using pgwt_fun[OF assms(1)] pgwt_arity[OF assms(1)] by fastforce+
    thus ?thesis using const_type_inv ‹Γ t = TAtom a› by auto
```

```
  next
    case (Fun g Y)
    obtain f X where *: "t = Fun f X" using pgwt_fun[OF assms(1)] by force
    hence "f = g" "map Γ X = Y"
      using fun_type_id_eq ‹Γ t = TComp g Y› fun_type[OF fun_type_inv(1)[OF ‹Γ t = TComp g Y›]]
      by fastforce+
    thus ?thesis using *(1) ‹Γ t = TComp g Y› by auto
  qed
  thus "Γ t = TAtom a ⟹ ∃f. t = Fun f []" "Γ t = TComp g Y ⟹ ∃X. t = Fun g X ∧ map Γ X = Y"
    by auto
qed

lemma wt_subst_Var[simp]: "wt_subst Var" by (metis wt_subst_def)

lemma wt_subst_trm: "(⋀v. v ∈ fv t ⟹ Γ (Var v) = Γ (ϑ v)) ⟹ Γ t = Γ (t · ϑ)"
proof (induction t)
  case (Fun f X)
  hence *: "⋀x. x ∈ set X ⟹ Γ x = Γ (x · ϑ)" by auto
  show ?case
  proof (cases "f ∈ Σf")
    case True
    hence "∀X. Γ (Fun f X) = TComp f (map Γ X)" using fun_type' by auto
    thus ?thesis using * by auto
  next
    case False
    hence "∃a ∈ 𝔗a. ∀X. Γ (Fun f X) = TAtom a" using const_type' by auto
    thus ?thesis by auto
  qed
qed auto

lemma wt_subst_trm': "⟦wt_subst σ; Γ s = Γ t⟧ ⟹ Γ (s · σ) = Γ (t · σ)"
by (metis wt_subst_trm wt_subst_def)

lemma wt_subst_trm'': "wt_subst σ ⟹ Γ t = Γ (t · σ)"
by (metis wt_subst_trm wt_subst_def)

lemma wt_subst_compose:
  assumes "wt_subst ϑ" "wt_subst δ" shows "wt_subst (ϑ ∘s δ)"
proof -
  have "⋀v. Γ (ϑ v) = Γ (ϑ v · δ)" using wt_subst_trm ‹wt_subst δ› unfolding wt_subst_def by metis
  moreover have "⋀v. Γ (Var v) = Γ (ϑ v)" using ‹wt_subst ϑ› unfolding wt_subst_def by metis
  ultimately have "⋀v. Γ (Var v) = Γ (ϑ v · δ)" by metis
  thus ?thesis unfolding wt_subst_def subst_compose_def by metis
qed

lemma wt_subst_TAtom_Var_cases:
  assumes ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
  and x: "Γ (Var x) = TAtom a"
  shows "(∃y. ϑ x = Var y) ∨ (∃c. ϑ x = Fun c [])"
proof (cases "(∃y. ϑ x = Var y)")
  case False
  then obtain c T where c: "ϑ x = Fun c T"
    by (cases "ϑ x") simp_all
  hence "wf_trm (Fun c T)"
    using ϑ(2) by fastforce
  hence "T = []"
    using const_type_inv_wf[of c T a] x c wt_subst_trm''[OF ϑ(1), of "Var x"]
    by fastforce
  thus ?thesis
    using c by blast
qed simp

lemma wt_subst_TAtom_fv:
```

```
  assumes ϑ: "wt_subst ϑ" "∀x. wf_trm (ϑ x)"
  and "∀x ∈ fv t - X. ∃a. Γ (Var x) = TAtom a"
  shows "∀x ∈ fv (t · ϑ) - fv_set (ϑ ` X). ∃a. Γ (Var x) = TAtom a"
using assms(3)
proof (induction t)
  case (Var x) thus ?case
  proof (cases "x ∈ X")
    case False
    with Var obtain a where "Γ (Var x) = TAtom a" by moura
    hence *: "Γ (ϑ x) = TAtom a" "wf_trm (ϑ x)" using ϑ unfolding wt_subst_def by auto
    show ?thesis
    proof (cases "ϑ x")
      case (Var y) thus ?thesis using * by auto
    next
      case (Fun f T)
      hence "T = []" using * const_type_inv[of f T a] unfolding wf_trm_def by auto
      thus ?thesis using Fun by auto
    qed
  qed auto
qed fastforce

lemma wt_subst_TAtom_subterms_subst:
  assumes "wt_subst ϑ" "∀x ∈ fv t. ∃a. Γ (Var x) = TAtom a" "wf_trms (ϑ ` fv t)"
  shows "subterms (t · ϑ) = subterms t ·_set ϑ"
using assms(2,3)
proof (induction t)
  case (Var x)
  obtain a where a: "Γ (Var x) = TAtom a" using Var.prems(1) by moura
  hence "Γ (ϑ x) = TAtom a" using wt_subst_trm''[OF assms(1), of "Var x"] by simp
  hence "(∃y. ϑ x = Var y) ∨ (∃c. ϑ x = Fun c [])"
    using const_type_inv_wf Var.prems(2) by (cases "ϑ x") auto
  thus ?case by auto
next
  case (Fun f T)
  have "subterms (t · ϑ) = subterms t ·_set ϑ" when "t ∈ set T" for t
    using that Fun.prems(1,2) Fun.IH[OF that]
    by auto
  thus ?case by auto
qed

lemma wt_subst_TAtom_subterms_set_subst:
  assumes "wt_subst ϑ" "∀x ∈ fv_set M. ∃a. Γ (Var x) = TAtom a" "wf_trms (ϑ ` fv_set M)"
  shows "subterms_set (M ·_set ϑ) = subterms_set M ·_set ϑ"
proof
  show "subterms_set (M ·_set ϑ) ⊆ subterms_set M ·_set ϑ"
  proof
    fix t assume "t ∈ subterms_set (M ·_set ϑ)"
    then obtain s where s: "s ∈ M" "t ∈ subterms (s · ϑ)" by auto
    thus "t ∈ subterms_set M ·_set ϑ"
      using assms(2,3) wt_subst_TAtom_subterms_subst[OF assms(1), of s]
      by auto
  qed

  show "subterms_set M ·_set ϑ ⊆ subterms_set (M ·_set ϑ)"
  proof
    fix t assume "t ∈ subterms_set M ·_set ϑ"
    then obtain s where s: "s ∈ M" "t ∈ subterms s ·_set ϑ" by auto
    thus "t ∈ subterms_set (M ·_set ϑ)"
      using assms(2,3) wt_subst_TAtom_subterms_subst[OF assms(1), of s]
      by auto
  qed
qed
```

```
lemma wt_subst_subst_upd:
  assumes "wt_subst ϑ"
    and "Γ (Var x) = Γ t"
  shows "wt_subst (ϑ(x := t))"
using assms unfolding wt_subst_def
by (metis fun_upd_other fun_upd_same)

lemma wt_subst_const_fv_type_eq:
  assumes "∀x ∈ fv t. ∃a. Γ (Var x) = TAtom a"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)"
  shows "∀x ∈ fv (t · δ). ∃y ∈ fv t. Γ (Var x) = Γ (Var y)"
using assms(1)
proof (induction t)
  case (Var x)
  then obtain a where a: "Γ (Var x) = TAtom a" by moura
  show ?case
  proof (cases "δ x")
    case (Fun f T)
    hence "wf_trm (Fun f T)" "Γ (Fun f T) = TAtom a"
      using a wt_subst_trm''[OF δ(1), of "Var x"] δ(2) by fastforce+
    thus ?thesis using const_type_inv_wf Fun by fastforce
  qed (use a wt_subst_trm''[OF δ(1), of "Var x"] in simp)
qed fastforce

lemma TComp_term_cases:
  assumes "wf_trm t" "Γ t = TComp f T"
  shows "(∃v. t = Var v) ∨ (∃T'. t = Fun f T' ∧ T = map Γ T' ∧ T' ≠ [])"
proof (cases "∃v. t = Var v")
  case False
  then obtain T' where T': "t = Fun f T'" "T = map Γ T'"
    using assms fun_type[OF fun_type_inv(1)[OF assms(2)]] fun_type_id_eq
    by (cases t) force+
  thus ?thesis using assms fun_type_inv(1) fun_type_inv_wf by fastforce
qed metis

lemma TAtom_term_cases:
  assumes "wf_trm t" "Γ t = TAtom τ"
  shows "(∃v. t = Var v) ∨ (∃f. t = Fun f [])"
using assms const_type_inv unfolding wf_trm_def by (cases t) auto

lemma subtermeq_imp_subtermtypeeq:
  assumes "wf_trm t" "s ⊑ t"
  shows "Γ s ⊑ Γ t"
using assms(2,1)
proof (induction t)
  case (Fun f T) thus ?case
  proof (cases "s = Fun f T")
    case False
    then obtain x where x: "x ∈ set T" "s ⊑ x" using Fun.prems(1) by auto
    hence "wf_trm x" using wf_trm_subtermeq[OF Fun.prems(2)] Fun_param_is_subterm[of _ T f] by auto
    hence "Γ s ⊑ Γ x" using Fun.IH[OF x] by simp
    moreover have "arity f > 0" using x fun_type_inv_wf Fun.prems
      by (metis length_pos_if_in_set term.order_refl wf_trm_def)
    ultimately show ?thesis using x Fun.prems fun_type[of f T] by auto
  qed simp
qed simp

lemma subterm_funs_term_in_type:
  assumes "wf_trm t" "Fun f T ⊑ t" "Γ (Fun f T) = TComp f (map Γ T)"
  shows "f ∈ funs_term (Γ t)"
using assms(2,1,3)
proof (induction t)
  case (Fun f' T')
```

```
  hence [simp]: "wf_trm (Fun f T)" by (metis wf_trm_subtermeq)
  { fix a assume τ: "Γ (Fun f' T') = TAtom a"
    hence "Fun f T = Fun f' T'" using Fun TAtom_term_cases subtermeq_Var_const by metis
    hence False using Fun.prems(3) τ by simp
  }
  moreover
  { fix g S assume τ: "Γ (Fun f' T') = TComp g S"
    hence "g = f'" "S = map Γ T'"
      using Fun.prems(2) fun_type_id_eq[OF τ] fun_type[OF fun_type_inv(1)[OF τ]]
      by auto
    hence τ': "Γ (Fun f' T') = TComp f' (map Γ T')" using τ by auto
    hence "g ∈ funs_term (Γ (Fun f' T'))" using τ by auto
    moreover {
      assume "Fun f T ≠ Fun f' T'"
      then obtain x where "x ∈ set T'" "Fun f T ⊑ x" using Fun.prems(1) by auto
      hence "f ∈ funs_term (Γ x)"
        using Fun.IH[OF _ _ _ Fun.prems(3), of x] wf_trm_subtermeq[OF ‹wf_trm (Fun f' T')›, of x]
        by force
      moreover have "Γ x ∈ set (map Γ T')" using τ' ‹x ∈ set T'› by auto
      ultimately have "f ∈ funs_term (Γ (Fun f' T'))" using τ' by auto
    }
    ultimately have ?case by (cases "Fun f T = Fun f' T'") (auto simp add: ‹g = f'›)
  }
  ultimately show ?case by (cases "Γ (Fun f' T')") auto
qed simp

lemma wt_subst_fv_termtype_subterm:
  assumes "x ∈ fv (ϑ y)"
    and "wt_subst ϑ"
    and "wf_trm (ϑ y)"
  shows "Γ (Var x) ⊑ Γ (Var y)"
using subtermeq_imp_subtermtypeeq[OF assms(3) var_is_subterm[OF assms(1)]]
      wt_subst_trm''[OF assms(2), of "Var y"]
by auto

lemma wt_subst_fv_set_termtype_subterm:
  assumes "x ∈ fv_set (ϑ ` Y)"
    and "wt_subst ϑ"
    and "wf_trms (subst_range ϑ)"
  shows "∃y ∈ Y. Γ (Var x) ⊑ Γ (Var y)"
using wt_subst_fv_termtype_subterm[OF _ assms(2), of x] assms(1,3)
by fastforce

lemma funs_term_type_iff:
  assumes t: "wf_trm t"
    and f: "arity f > 0"
  shows "f ∈ funs_term (Γ t) ⟷ (f ∈ funs_term t ∨ (∃x ∈ fv t. f ∈ funs_term (Γ (Var x))))"
    (is "?P t ⟷ ?Q t")
using t
proof (induction t)
  case (Fun g T)
  hence IH: "?P s ⟷ ?Q s" when "s ∈ set T" for s
    using that wf_trm_subterm[OF _ Fun_param_is_subterm]
    by blast
  have 0: "arity g = length T" using Fun.prems unfolding wf_trm_def by auto
  show ?case
  proof (cases "f = g")
    case True thus ?thesis using fun_type[OF f] by simp
  next
    case False
    have "?P (Fun g T) ⟷ (∃s ∈ set T. ?P s)"
    proof
      assume *: "?P (Fun g T)"
```

```
      hence "Γ (Fun g T) = TComp g (map Γ T)"
        using const_type[of g] fun_type[of g] by force
      thus "∃s ∈ set T. ?P s" using False * by force
    next
      assume *: "∃s ∈ set T. ?P s"
      hence "Γ (Fun g T) = TComp g (map Γ T)"
        using 0 const_type[of g] fun_type[of g] by force
      thus "?P (Fun g T)" using False * by force
    qed
    thus ?thesis using False f IH by auto
  qed
qed simp

lemma funs_term_type_iff':
  assumes M: "wf_trms M"
    and f: "arity f > 0"
  shows "f ∈ ⋃(funs_term ` Γ ` M) ⟷
        (f ∈ ⋃(funs_term ` M) ∨ (∃x ∈ fv_set M. f ∈ funs_term (Γ (Var x))))" (is "?A ⟷ ?B")
proof
  assume ?A
  then obtain t where "t ∈ M" "wf_trm t" "f ∈ funs_term (Γ t)" using M by moura
  thus ?B using funs_term_type_iff[OF _ f, of t] by auto
next
  assume ?B
  then obtain t where "t ∈ M" "wf_trm t" "f ∈ funs_term t ∨ (∃x ∈ fv t. f ∈ funs_term (Γ (Var
x)))"
    using M by auto
  thus ?A using funs_term_type_iff[OF _ f, of t] by blast
qed

lemma Ana_subterm_type:
  assumes "Ana t = (K,M)"
    and "wf_trm t"
    and "m ∈ set M"
  shows "Γ m ⊑ Γ t"
proof -
  have "m ⊑ t" using Ana_subterm[OF assms(1)] assms(3) by auto
  thus ?thesis using subtermeq_imp_subtermtypeeq[OF assms(2)] by simp
qed

lemma wf_trm_TAtom_subterms:
  assumes "wf_trm t" "Γ t = TAtom τ"
  shows "subterms t = {t}"
using assms const_type_inv unfolding wf_trm_def by (cases t) force+

lemma wf_trm_TComp_subterm:
  assumes "wf_trm s" "t ⊏ s"
  obtains f T where "Γ s = TComp f T"
proof (cases s)
  case (Var x) thus ?thesis using ‹t ⊏ s› by simp
next
  case (Fun g S)
  hence "length S > 0" using assms Fun_subterm_inside_params[of t g S] by auto
  hence "arity g > 0" by (metis ‹wf_trm s› ‹s = Fun g S› term.order_refl wf_trm_def)
  thus ?thesis using fun_type ‹s = Fun g S› that by auto
qed

lemma SMP_empty[simp]: "SMP {} = {}"
proof (rule ccontr)
  assume "SMP {} ≠ {}"
  then obtain t where "t ∈ SMP {}" by auto
  thus False by (induct t rule: SMP.induct) auto
qed
```

```
lemma SMP_I:
  assumes "s ∈ M" "wt_subst δ" "t ⊑ s · δ" "⋀v. wf_trm (δ v)"
  shows "t ∈ SMP M"
using SMP.Substitution[OF SMP.MP[OF assms(1)] assms(2)] SMP.Subterm[of "s · δ" M t] assms(3,4)
by (cases "t = s · δ") simp_all

lemma SMP_wf_trm:
  assumes "t ∈ SMP M" "wf_trms M"
  shows "wf_trm t"
using assms(1)
by (induct t rule: SMP.induct)
   (use assms(2) in blast,
    use wf_trm_subtermeq in blast,
    use wf_trm_subst in blast,
    use Ana_keys_wf' in blast)

lemma SMP_ikI[intro]: "t ∈ ik_st S ⟹ t ∈ SMP (trms_st S)" by force

lemma MP_setI[intro]: "x ∈ set S ⟹ trms_stp x ⊆ trms_st S" by force

lemma SMP_setI[intro]: "x ∈ set S ⟹ trms_stp x ⊆ SMP (trms_st S)" by force

lemma SMP_subset_I:
  assumes M: "∀t ∈ M. ∃s δ. s ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"
  shows "SMP M ⊆ SMP N"
proof
  fix t show "t ∈ SMP M ⟹ t ∈ SMP N"
  proof (induction t rule: SMP.induct)
    case (MP t)
    then obtain s δ where s: "s ∈ N" "wt_subst δ" "wf_trms (subst_range δ)" "t = s · δ"
      using M by moura
    show ?case using SMP_I[OF s(1,2), of "s · δ"] s(3,4) wf_trm_subst_range_iff by fast
  qed (auto intro!: SMP.Substitution[of _ N])
qed

lemma SMP_union: "SMP (A ∪ B) = SMP A ∪ SMP B"
proof
  show "SMP (A ∪ B) ⊆ SMP A ∪ SMP B"
  proof
    fix t assume "t ∈ SMP (A ∪ B)"
    thus "t ∈ SMP A ∪ SMP B" by (induct rule: SMP.induct) blast+
  qed

  { fix t assume "t ∈ SMP A" hence "t ∈ SMP (A ∪ B)" by (induct rule: SMP.induct) blast+ }
  moreover { fix t assume "t ∈ SMP B" hence "t ∈ SMP (A ∪ B)" by (induct rule: SMP.induct) blast+
}
  ultimately show "SMP A ∪ SMP B ⊆ SMP (A ∪ B)" by blast
qed

lemma SMP_append[simp]: "SMP (trms_st (S@S')) = SMP (trms_st S) ∪ SMP (trms_st S')" (is "?A = ?B")
using SMP_union by simp

lemma SMP_mono: "A ⊆ B ⟹ SMP A ⊆ SMP B"
proof -
  assume "A ⊆ B"
  then obtain C where "B = A ∪ C" by moura
  thus "SMP A ⊆ SMP B" by (simp add: SMP_union)
qed

lemma SMP_Union: "SMP (⋃m ∈ M. f m) = (⋃m ∈ M. SMP (f m))"
proof
  show "SMP (⋃m∈M. f m) ⊆ (⋃m∈M. SMP (f m))"
```

```
  proof
    fix t assume "t ∈ SMP (⋃m∈M. f m)"
    thus "t ∈ (⋃m∈M. SMP (f m))" by (induct t rule: SMP.induct) force+
  qed
  show "(⋃m∈M. SMP (f m)) ⊆ SMP (⋃m∈M. f m)"
  proof
    fix t assume "t ∈ (⋃m∈M. SMP (f m))"
    then obtain m where "m ∈ M" "t ∈ SMP (f m)" by moura
    thus "t ∈ SMP (⋃m∈M. f m)" using SMP_mono[of "f m" "⋃m∈M. f m"] by auto
  qed
qed

lemma SMP_singleton_ex:
  "t ∈ SMP M ⟹ (∃m ∈ M. t ∈ SMP {m})"
  "m ∈ M ⟹ t ∈ SMP {m} ⟹ t ∈ SMP M"
using SMP_Union[of "λt. {t}" M] by auto

lemma SMP_Cons: "SMP (trms_st (x#S)) = SMP (trms_st [x]) ∪ SMP (trms_st S)"
using SMP_append[of "[x]" S] by auto

lemma SMP_Nil[simp]: "SMP (trms_st []) = {}"
proof -
  { fix t assume "t ∈ SMP (trms_st [])" hence False by induct auto }
  thus ?thesis by blast
qed

lemma SMP_subset_union_eq: assumes "M ⊆ SMP N" shows "SMP N = SMP (M ∪ N)"
proof -
  { fix t assume "t ∈ SMP (M ∪ N)" hence "t ∈ SMP N"
      using assms by (induction rule: SMP.induct) blast+
  }
  thus ?thesis using SMP_union by auto
qed

lemma SMP_subterms_subset: "subterms_set M ⊆ SMP M"
proof
  fix t assume "t ∈ subterms_set M"
  then obtain m where "m ∈ M" "t ⊑ m" by auto
  thus "t ∈ SMP M" using SMP_I[of _ _ Var] by auto
qed

lemma SMP_SMP_subset: "N ⊆ SMP M ⟹ SMP N ⊆ SMP M"
by (metis SMP_mono SMP_subset_union_eq Un_commute Un_upper2)

lemma wt_subst_rm_vars: "wt_subst δ ⟹ wt_subst (rm_vars X δ)"
using rm_vars_dom unfolding wt_subst_def by auto

lemma wt_subst_SMP_subset:
  assumes "trms_st S ⊆ SMP S'" "wt_subst δ" "wf_trms (subst_range δ)"
  shows "trms_st (S ·st δ) ⊆ SMP S'"
proof
  fix t assume *: "t ∈ trms_st (S ·st δ)"
  show "t ∈ SMP S'" using trm_strand_subst_cong(2)[OF *]
  proof
    assume "∃t'. t = t' · δ ∧ t' ∈ trms_st S"
    thus "t ∈ SMP S'" using assms SMP.Substitution by auto
  next
    assume "∃X F. Inequality X F ∈ set S ∧ (∃t'∈trms_pairs F. t = t' · rm_vars (set X) δ)"
    then obtain X F t' where **:
        "Inequality X F ∈ set S" "t'∈trms_pairs F" "t = t' · rm_vars (set X) δ"
      by force
    then obtain s where s: "s ∈ trms_stp (Inequality X F)" "t = s · rm_vars (set X) δ" by moura
    hence "s ∈ SMP (trms_st S)" using **(1) by force
```

```
    hence "t ∈ SMP (trms_st S)"
      using SMP.Substitution[OF _ wt_subst_rm_vars[OF assms(2)] wf_trms_subst_rm_vars'[OF assms(3)]]
      unfolding s(2) by blast
    thus "t ∈ SMP S'" by (metis SMP_union SMP_subset_union_eq UnCI assms(1))
  qed
qed

lemma MP_subset_SMP: "⋃(trms_stp ` set S) ⊆ SMP (trms_st S)" "trms_st S ⊆ SMP (trms_st S)" "M ⊆ SMP M"
by auto

lemma SMP_fun_map_snd_subset: "SMP (trms_st (map Send X)) ⊆ SMP (trms_st [Send (Fun f X)])"
proof
  fix t assume "t ∈ SMP (trms_st (map Send X))" thus "t ∈ SMP (trms_st [Send (Fun f X)])"
  proof (induction t rule: SMP.induct)
    case (MP t)
    hence "t ∈ set X" by auto
    hence "t ⊏ Fun f X" by (metis subtermI')
    thus ?case using SMP.Subterm[of "Fun f X" "trms_st [Send (Fun f X)]" t] using SMP.MP by auto
  qed blast+
qed

lemma SMP_wt_subst_subset:
  assumes "t ∈ SMP (M ·set ℐ)" "wt_subst ℐ" "wf_trms (subst_range ℐ)"
  shows "t ∈ SMP M"
using assms wf_trm_subst_range_iff[of ℐ] by (induct t rule: SMP.induct) blast+

lemma SMP_wt_instances_subset:
  assumes "∀t ∈ M. ∃s ∈ N. ∃δ. t = s · δ ∧ wt_subst δ ∧ wf_trms (subst_range δ)"
    and "t ∈ SMP M"
  shows "t ∈ SMP N"
proof -
  obtain m where m: "m ∈ M" "t ∈ SMP {m}" using SMP_singleton_ex(1)[OF assms(2)] by blast
  then obtain n δ where n: "n ∈ N" "m = n · δ" "wt_subst δ" "wf_trms (subst_range δ)"
    using assms(1) by fast

  have "t ∈ SMP (N ·set δ)" using n(1,2) SMP_singleton_ex(2)[of m "N ·set δ", OF _ m(2)] by fast
  thus ?thesis using SMP_wt_subst_subset[OF _ n(3,4)] by blast
qed

lemma SMP_consts:
  assumes "∀t ∈ M. ∃c. t = Fun c []"
    and "∀t ∈ M. Ana t = ([], [])"
  shows "SMP M = M"
proof
  show "SMP M ⊆ M"
  proof
    fix t show "t ∈ SMP M ⟹ t ∈ M"
      apply (induction t rule: SMP.induct)
      by (use assms in auto)
  qed
qed auto

lemma SMP_subterms_eq:
  "SMP (subterms_set M) = SMP M"
proof
  show "SMP M ⊆ SMP (subterms_set M)" using SMP_mono[of M "subterms_set M"] by blast
  show "SMP (subterms_set M) ⊆ SMP M"
  proof
    fix t show "t ∈ SMP (subterms_set M) ⟹ t ∈ SMP M" by (induction t rule: SMP.induct) blast+
  qed
qed

lemma SMP_funs_term:
```

```
  assumes t: "t ∈ SMP M" "f ∈ funs_term t ∨ (∃x ∈ fv t. f ∈ funs_term (Γ (Var x)))"
    and f: "arity f > 0"
    and M: "wf_trms M"
    and Ana_f: "⋀s K T. Ana s = (K,T) ⟹ f ∈ ⋃(funs_term ` set K) ⟹ f ∈ funs_term s"
  shows "f ∈ ⋃(funs_term ` M) ∨ (∃x ∈ fv_set M. f ∈ funs_term (Γ (Var x)))"
using t
proof (induction t rule: SMP.induct)
  case (Subterm t t')
  thus ?case by (metis UN_I vars_iff_subtermeq funs_term_subterms_eq(1) term.order_trans)
next
  case (Substitution t δ)
  show ?case
    using M SMP_wf_trm[OF Substitution.hyps(1)] wf_trm_subst[of δ t, OF Substitution.hyps(3)]
          funs_term_type_iff[OF _ f] wt_subst_trm''[OF Substitution.hyps(2), of t]
          Substitution.prems Substitution.IH
    by metis
next
  case (Ana t K T t')
  thus ?case
    using Ana_f[OF Ana.hyps(2)] Ana_keys_fv[OF Ana.hyps(2)]
    by fastforce
qed auto

lemma id_type_eq:
  assumes "Γ (Fun f X) = Γ (Fun g Y)"
  shows "f ∈ C ⟹ g ∈ C" "f ∈ Σf ⟹ g ∈ Σf"
using assms const_type' fun_type' id_union_univ(1)
by (metis UNIV_I UnE "term.distinct"(1))+

lemma fun_type_arg_cong:
  assumes "f ∈ Σf" "g ∈ Σf" "Γ (Fun f (x#X)) = Γ (Fun g (y#Y))"
  shows "Γ x = Γ y" "Γ (Fun f X) = Γ (Fun g Y)"
using assms fun_type' by auto

lemma fun_type_arg_cong':
  assumes "f ∈ Σf" "g ∈ Σf" "Γ (Fun f (X@x#X')) = Γ (Fun g (Y@y#Y'))" "length X = length Y"
  shows "Γ x = Γ y"
using assms
proof (induction X arbitrary: Y)
  case Nil thus ?case using fun_type_arg_cong(1)[of f g x X' y Y'] by auto
next
  case (Cons x' X Y'')
  then obtain y' Y where "Y'' = y'#Y" by (metis length_Suc_conv)
  hence "Γ (Fun f (X@x#X')) = Γ (Fun g (Y@y#Y'))" "length X = length Y"
    using Cons.prems(3,4) fun_type_arg_cong(2)[OF Cons.prems(1,2), of x' "X@x#X'"] by auto
  thus ?thesis using Cons.IH[OF Cons.prems(1,2)] by auto
qed

lemma fun_type_param_idx: "Γ (Fun f T) = Fun g S ⟹ i < length T ⟹ Γ (T ! i) = S ! i"
by (metis fun_type fun_type_id_eq fun_type_inv(1) nth_map term.inject(2))

lemma fun_type_param_ex:
  assumes "Γ (Fun f T) = Fun g (map Γ S)" "t ∈ set S"
  shows "∃s ∈ set T. Γ s = Γ t"
using fun_type_length_eq[OF assms(1)] length_map[of Γ S] assms(2)
      fun_type_param_idx[OF assms(1)] nth_map in_set_conv_nth
by metis

lemma tfr_stp_all_split:
  "list_all tfr_stp (x#S) ⟹ list_all tfr_stp [x]"
  "list_all tfr_stp (x#S) ⟹ list_all tfr_stp S"
  "list_all tfr_stp (S@S') ⟹ list_all tfr_stp S"
  "list_all tfr_stp (S@S') ⟹ list_all tfr_stp S'"
```

```
  "list_all tfr_stp (S@x#S') ⟹ list_all tfr_stp (S@S')"
by fastforce+

lemma tfr_stp_all_append:
  assumes "list_all tfr_stp S" "list_all tfr_stp S'"
  shows "list_all tfr_stp (S@S')"
using assms by fastforce

lemma tfr_stp_all_wt_subst_apply:
  assumes "list_all tfr_stp S"
    and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
           "bvars_st S ∩ range_vars ϑ = {}"
  shows "list_all tfr_stp (S ·_st ϑ)"
using assms(1,4)
proof (induction S)
  case (Cons x S)
  hence IH: "list_all tfr_stp (S ·_st ϑ)"
    using tfr_stp_all_split(2)[of x S]
    unfolding range_vars_alt_def by fastforce
  thus ?case
  proof (cases x)
    case (Equality a t t')
    hence "(∃δ. Unifier δ t t') ⟶ Γ t = Γ t'" using Cons.prems by auto
    hence "(∃δ. Unifier δ (t · ϑ) (t' · ϑ)) ⟶ Γ (t · ϑ) = Γ (t' · ϑ)"
      by (metis Unifier_comp' wt_subst_trm'[OF assms(2)])
    moreover have "(x#S) ·_st ϑ = Equality a (t · ϑ) (t' · ϑ)#(S ·_st ϑ)"
      using ‹x = Equality a t t'› by auto
    ultimately show ?thesis using IH by auto
  next
    case (Inequality X F)
    let ?σ = "rm_vars (set X) ϑ"
    let ?G = "F ·_pairs ?σ"

    let ?P = "λF X. ∀x ∈ fv_pairs F - set X. ∃a. Γ (Var x) = TAtom a"
    let ?Q = "λF X.
      ∀f T. Fun f T ∈ subterms_set (trms_pairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)"

    have 0: "set X ∩ range_vars ?σ = {}"
      using Cons.prems(2) Inequality rm_vars_img_subset[of "set X"]
      by (auto simp add: subst_domain_def range_vars_alt_def)

    have 1: "?P F X ∨ ?Q F X" using Inequality Cons.prems by simp

    have 2: "fv_set (?σ ` set X) = set X" by auto

    have "?P ?G X" when "?P F X" using that
    proof (induction F)
      case (Cons g G)
      obtain t t' where g: "g = (t,t')" by (metis surj_pair)

      have "∀x ∈ (fv (t · ?σ) ∪ fv (t' · ?σ)) - set X. ∃a. Γ (Var x) = Var a"
      proof -
        have *: "∀x ∈ fv t - set X. ∃a. Γ (Var x) = Var a"
                "∀x ∈ fv t' - set X. ∃a. Γ (Var x) = Var a"
          using g Cons.prems by simp_all

        have **: "∀x. wf_trm (?σ x)"
          using ϑ(2) wf_trm_subst_range_iff[of ϑ] wf_trm_subst_rm_vars'[of ϑ _ "set X"] by simp

        show ?thesis
          using wt_subst_TAtom_fv[OF wt_subst_rm_vars[OF ϑ(1)] ** *(1)]
                wt_subst_TAtom_fv[OF wt_subst_rm_vars[OF ϑ(1)] ** *(2)]
                wt_subst_trm'[OF wt_subst_rm_vars[OF ϑ(1), of "set X"]] 2
```

```
        by blast
      qed
      moreover have "∀x∈fv_pairs (G ·_pairs ?σ) - set X. ∃a. Γ (Var x) = Var a"
        using Cons by auto
      ultimately show ?case using g by (auto simp add: subst_apply_pairs_def)
    qed (simp add: subst_apply_pairs_def)
    hence "?P ?G X ∨ ?Q ?G X"
      using 1 ineq_subterm_inj_cond_subst[OF 0, of "trms_pairs F"] trms_pairs_subst[of F ?σ]
      by presburger
    moreover have "(x#S) ·_st ϑ = Inequality X (F ·_pairs ?σ)#(S ·_st ϑ)"
      using ‹x = Inequality X F› by auto
    ultimately show ?thesis using IH by simp
  qed auto
qed simp

lemma tfr_stp_all_same_type:
  "list_all tfr_stp (S@Equality a t t'#S') ⟹ Unifier δ t t' ⟹ Γ t = Γ t'"
by force+

lemma tfr_subset:
  "⋀A B. tfr_set (A ∪ B) ⟹ tfr_set A"
  "⋀A B. tfr_set B ⟹ A ⊆ B ⟹ tfr_set A"
  "⋀A B. tfr_set B ⟹ SMP A ⊆ SMP B ⟹ tfr_set A"
proof -
  show 1: "tfr_set (A ∪ B) ⟹ tfr_set A" for A B
    using SMP_union[of A B] unfolding tfr_set_def by simp

  fix A B assume B: "tfr_set B"

  show "A ⊆ B ⟹ tfr_set A"
  proof -
    assume "A ⊆ B"
    then obtain C where "B = A ∪ C" by moura
    thus ?thesis using B 1 by blast
  qed

  show "SMP A ⊆ SMP B ⟹ tfr_set A"
  proof -
    assume "SMP A ⊆ SMP B"
    then obtain C where "SMP B = SMP A ∪ C" by moura
    thus ?thesis using B unfolding tfr_set_def by blast
  qed
qed

lemma tfr_empty[simp]: "tfr_set {}"
unfolding tfr_set_def by simp

lemma tfr_consts_mono:
  assumes "∀t ∈ M. ∃c. t = Fun c []"
    and "∀t ∈ M. Ana t = ([], [])"
    and "tfr_set N"
  shows "tfr_set (N ∪ M)"
proof -
  { fix s t
    assume *: "s ∈ SMP (N ∪ M) - range Var" "t ∈ SMP (N ∪ M) - range Var" "∃δ. Unifier δ s t"
    hence **: "is_Fun s" "is_Fun t" "s ∈ SMP N ∨ s ∈ M" "t ∈ SMP N ∨ t ∈ M"
      using assms(3) SMP_consts[OF assms(1,2)] SMP_union[of N M] by auto
    moreover have "Γ s = Γ t" when "s ∈ SMP N" "t ∈ SMP N"
      using that assms(3) *(3) **(1,2) unfolding tfr_set_def by blast
    moreover have "Γ s = Γ t" when st: "s ∈ M" "t ∈ M"
    proof -
      obtain c d where "s = Fun c []" "t = Fun d []" using st assms(1) by moura
      hence "s = t" using *(3) by fast
```

```
      thus ?thesis by metis
    qed
    moreover have "Γ s = Γ t" when st: "s ∈ SMP N" "t ∈ M"
    proof -
      obtain c where "t = Fun c []" using st assms(1) by moura
      hence "s = t" using *(3) **(1,2) by auto
      thus ?thesis by metis
    qed
    moreover have "Γ s = Γ t" when st: "s ∈ M" "t ∈ SMP N"
    proof -
      obtain c where "s = Fun c []" using st assms(1) by moura
      hence "s = t" using *(3) **(1,2) by auto
      thus ?thesis by metis
    qed
    ultimately have "Γ s = Γ t" by metis
  } thus ?thesis by (metis tfr_set_def)
qed

lemma dual_st_tfr_stp: "list_all tfr_stp S ⟹ list_all tfr_stp (dual_st S)"
proof (induction S)
  case (Cons x S)
  have "list_all tfr_stp S" using Cons.prems by simp
  hence IH: "list_all tfr_stp (dual_st S)" using Cons.IH by metis
  from Cons show ?case
  proof (cases x)
    case (Equality a t t')
    hence "(∃δ. Unifier δ t t') ⟹ Γ t = Γ t'" using Cons by auto
    thus ?thesis using Equality IH by fastforce
  next
    case (Inequality X F)
    have "set (dual_st (x#S)) = insert x (set (dual_st S))" using Inequality by auto
    moreover have "(∀x ∈ fv_pairs F - set X. ∃a. Γ (Var x) = Var a) ∨
            (∀f T. Fun f T ∈ subterms_set (trms_pairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))"

      using Cons.prems Inequality by auto
    ultimately show ?thesis using Inequality IH by auto
  qed auto
qed simp

lemma subst_var_inv_wt:
  assumes "wt_subst δ"
  shows "wt_subst (subst_var_inv δ X)"
using assms f_inv_into_f[of _ δ X]
unfolding wt_subst_def subst_var_inv_def
by presburger

lemma subst_var_inv_wf_trms:
  "wf_trms (subst_range (subst_var_inv δ X))"
using f_inv_into_f[of _ δ X]
unfolding wt_subst_def subst_var_inv_def
by auto

lemma unify_list_wt_if_same_type:
  assumes "Unification.unify E B = Some U" "∀(s,t) ∈ set E. wf_trm s ∧ wf_trm t ∧ Γ s = Γ t"
  and "∀(v,t) ∈ set B. Γ (Var v) = Γ t"
  shows "∀(v,t) ∈ set U. Γ (Var v) = Γ t"
using assms
proof (induction E B arbitrary: U rule: Unification.unify.induct)
  case (2 f X g Y E B U)
  hence "wf_trm (Fun f X)" "wf_trm (Fun g Y)" "Γ (Fun f X) = Γ (Fun g Y)" by auto

  from "2.prems"(1) obtain E' where *: "decompose (Fun f X) (Fun g Y) = Some E'"
    and [simp]: "f = g" "length X = length Y" "E' = zip X Y"
```

```
    and **: "Unification.unify (E'@E) B = Some U"
    by (auto split: option.splits)

  have "∀(s,t) ∈ set E'. wf_trm s ∧ wf_trm t ∧ Γ s = Γ t"
  proof -
    { fix s t assume "(s,t) ∈ set E'"
      then obtain X' X'' Y' Y'' where "X = X'@s#X''" "Y = Y'@t#Y''" "length X' = length Y'"
        using zip_arg_subterm_split[of s t X Y] ‹E' = zip X Y› by metis
      hence "Γ (Fun f (X'@s#X'')) = Γ (Fun g (Y'@t#Y''))" by (metis ‹Γ (Fun f X) = Γ (Fun g Y)›)

      from ‹E' = zip X Y› have "∀(s,t) ∈ set E'. s ⊏ Fun f X ∧ t ⊏ Fun g Y"
        using zip_arg_subterm[of _ _ X Y] by blast
      with ‹(s,t) ∈ set E'› have "wf_trm s" "wf_trm t"
        using wf_trm_subterm ‹wf_trm (Fun f X)› ‹wf_trm (Fun g Y)› by (blast,blast)
      moreover have "f ∈ Σ_f"
      proof (rule ccontr)
        assume "f ∉ Σ_f"
        hence "f ∈ C" "arity f = 0" using const_arity_eq_zero[of f] by simp_all
        thus False using ‹wf_trm (Fun f X)› * ‹(s,t) ∈ set E'› unfolding wf_trm_def by auto
      qed
      hence "Γ s = Γ t"
        using fun_type_arg_cong' ‹f ∈ Σ_f› ‹Γ (Fun f (X'@s#X'')) = Γ (Fun g (Y'@t#Y''))›
              ‹length X' = length Y'› ‹f = g›
        by metis
      ultimately have "wf_trm s" "wf_trm t" "Γ s = Γ t" by metis+
    }
    thus ?thesis by blast
  qed
  moreover have "∀(s,t) ∈ set E. wf_trm s ∧ wf_trm t ∧ Γ s = Γ t" using "2.prems"(2) by auto
  ultimately show ?case using "2.IH"[OF * ** _ "2.prems"(3)] by fastforce
next
  case (3 v t E B U)
  hence "Γ (Var v) = Γ t" "wf_trm t" by auto
  hence "wt_subst (subst v t)"
      and *: "∀(v, t) ∈ set ((v,t)#B). Γ (Var v) = Γ t"
             "⋀t t'. (t,t') ∈ set E ⟹ Γ t = Γ t'"
    using "3.prems"(2,3) unfolding wt_subst_def subst_def by auto

  show ?case
  proof (cases "t = Var v")
    assume "t = Var v" thus ?case using 3 by auto
  next
    assume "t ≠ Var v"
    hence "v ∉ fv t" using "3.prems"(1) by auto
    hence **: "Unification.unify (subst_list (subst v t) E) ((v, t)#B) = Some U"
      using Unification.unify.simps(3)[of v t E B] "3.prems"(1) ‹t ≠ Var v› by auto

    have "∀(s, t) ∈ set (subst_list (subst v t) E). wf_trm s ∧ wf_trm t"
      using wf_trm_subst_singleton[OF _ ‹wf_trm t›] "3.prems"(2)
      unfolding subst_list_def subst_def by auto
    moreover have "∀(s, t) ∈ set (subst_list (subst v t) E). Γ s = Γ t"
      using *(2)[THEN wt_subst_trm'[OF ‹wt_subst (subst v t)›]] by (simp add: subst_list_def)
    ultimately show ?thesis using "3.IH"(2)[OF ‹t ≠ Var v› ‹v ∉ fv t› ** _ *(1)] by auto
  qed
next
  case (4 f X v E B U)
  hence "Γ (Var v) = Γ (Fun f X)" "wf_trm (Fun f X)" by auto
  hence "wt_subst (subst v (Fun f X))"
      and *: "∀(v, t) ∈ set ((v,(Fun f X))#B). Γ (Var v) = Γ t"
             "⋀t t'. (t,t') ∈ set E ⟹ Γ t = Γ t'"
    using "4.prems"(2,3) unfolding wt_subst_def subst_def by auto

  have "v ∉ fv (Fun f X)" using "4.prems"(1) by force
```

```
  hence **: "Unification.unify (subst_list (subst v (Fun f X)) E) ((v, (Fun f X))#B) = Some U"
    using Unification.unify.simps(3)[of v "Fun f X" E B] "4.prems"(1) by auto

  have "∀(s, t) ∈ set (subst_list (subst v (Fun f X)) E). wf_trm s ∧ wf_trm t"
    using wf_trm_subst_singleton[OF _ ‹wf_trm (Fun f X)›] "4.prems"(2)
    unfolding subst_list_def subst_def by auto
  moreover have "∀(s, t) ∈ set (subst_list (subst v (Fun f X)) E). Γ s = Γ t"
    using *(2)[THEN wt_subst_trm'[OF ‹wt_subst (subst v (Fun f X))›]] by (simp add: subst_list_def)
  ultimately show ?case using "4.IH"[OF ‹v ∉ fv (Fun f X)› ** _ *(1)] by auto
qed auto

lemma mgu_wt_if_same_type:
  assumes "mgu s t = Some σ" "wf_trm s" "wf_trm t" "Γ s = Γ t"
  shows "wt_subst σ"
proof -
  let ?fv_disj = "λv t S. ¬(∃(v',t') ∈ S - {(v,t)}. (insert v (fv t)) ∩ (insert v' (fv t')) ≠ {})"

  from assms(1) obtain σ' where "Unification.unify [(s,t)] [] = Some σ'" "subst_of σ' = σ"
    by (auto split: option.splits)
  hence "∀(v,t) ∈ set σ'. Γ (Var v) = Γ t" "distinct (map fst σ')"
    using assms(2,3,4) unify_list_wt_if_same_type unify_list_distinct[of "[(s,t)]"] by auto
  thus "wt_subst σ" using ‹subst_of σ' = σ› unfolding wt_subst_def
  proof (induction σ' arbitrary: σ rule: List.rev_induct)
    case (snoc tt σ' σ)
    then obtain v t where tt: "tt = (v,t)" by (metis surj_pair)
    hence σ: "σ = subst v t ∘s subst_of σ'" using snoc.prems(3) by simp

    have "∀(v,t) ∈ set σ'. Γ (Var v) = Γ t" "distinct (map fst σ')" using snoc.prems(1,2) by auto
    then obtain σ'' where σ'': "subst_of σ' = σ''" "∀v. Γ (Var v) = Γ (σ'' v)" by (metis snoc.IH)
    hence "Γ t = Γ (t · σ'')" for t using wt_subst_trm by blast
    hence "Γ (Var v) = Γ (σ'' v)" "Γ t = Γ (t · σ'')" using σ''(2) by auto
    moreover have "Γ (Var v) = Γ t" using snoc.prems(1) tt by simp
    moreover have σ2: "σ = Var(v := t) ∘s σ'' " using σ σ''(1) unfolding subst_def by simp
    ultimately have "Γ (Var v) = Γ (σ v)" unfolding subst_compose_def by simp

    have "subst_domain (subst v t) ⊆ {v}" unfolding subst_def by (auto simp add: subst_domain_def)
    hence *: "subst_domain σ ⊆ insert v (subst_domain σ'')"
      using tt σ σ''(1) snoc.prems(2) subst_domain_compose[of _ σ'']
      by (auto simp add: subst_domain_def)

    have "v ∉ set (map fst σ')" using tt snoc.prems(2) by auto
    hence "v ∉ subst_domain σ''" using σ''(1) subst_of_dom_subset[of σ'] by auto

    { fix w assume "w ∈ subst_domain σ''"
      hence "σ w = σ'' w" using σ2 σ''(1) ‹v ∉ subst_domain σ''› unfolding subst_compose_def by
auto
      hence "Γ (Var w) = Γ (σ w)" using σ''(2) by simp
    }
    thus ?case using ‹Γ (Var v) = Γ (σ v)› * by force
  qed simp
qed

lemma wt_Unifier_if_Unifier:
  assumes s_t: "wf_trm s" "wf_trm t" "Γ s = Γ t"
    and δ: "Unifier δ s t"
  shows "∃ϑ. Unifier ϑ s t ∧ wt_subst ϑ ∧ wf_trms (subst_range ϑ)"
using mgu_always_unifies[OF δ] mgu_gives_MGU[THEN MGU_is_Unifier[of s _ t]]
      mgu_wt_if_same_type[OF _ s_t] mgu_wf_trm[OF _ s_t(1,2)] wf_trm_subst_range_iff
by fast

end
```

3.3.3 Automatically Proving Type-Flaw Resistance

Definitions: Variable Renaming

```
abbreviation "max_var t ≡ Max (insert 0 (snd ‘ fv t))"
abbreviation "max_var_set X ≡ Max (insert 0 (snd ‘ X))"

definition "var_rename n v ≡ Var (fst v, snd v + Suc n)"
definition "var_rename_inv n v ≡ Var (fst v, snd v - Suc n)"
```

Definitions: Computing a Finite Representation of the Sub-Message Patterns

A sufficient requirement for a term to be a well-typed instance of another term

```
definition is_wt_instance_of_cond where
  "is_wt_instance_of_cond Γ t s ≡ (
    Γ t = Γ s ∧ (case mgu t s of
      None ⇒ False
    | Some δ ⇒ inj_on δ (fv t) ∧ (∀x ∈ fv t. is_Var (δ x))))"

definition has_all_wt_instances_of where
  "has_all_wt_instances_of Γ N M ≡ ∀t ∈ N. ∃s ∈ M. is_wt_instance_of_cond Γ t s"
```

This function computes a finite representation of the set of sub-message patterns

```
definition SMP0 where
  "SMP0 Ana Γ M ≡ let
      f = λt. Fun (the_Fun (Γ t)) (map Var (zip (args (Γ t)) [0..<length (args (Γ t))]));
      g = λM'. map f (filter (λt. is_Var t ∧ is_Fun (Γ t)) M')@
               concat (map (fst ∘ Ana) M')@concat (map subterms_list M');
      h = remdups ∘ g
    in while (λA. set (h A) ≠ set A) h M"
```

These definitions are useful to refine an SMP representation set

```
fun generalize_term where
  "generalize_term _ _ n (Var x) = (Var x, n)"
| "generalize_term Γ p n (Fun f T) = (let τ = Γ (Fun f T)
    in if p τ then (Var (τ, n), Suc n)
       else let (T',n') = foldr (λt (S,m). let (t',m') = generalize_term Γ p m t in (t'#S,m'))
                                T ([],n)
            in (Fun f T', n'))"

definition generalize_terms where
  "generalize_terms Γ p ≡ map (fst ∘ generalize_term Γ p 0)"

definition remove_superfluous_terms where
  "remove_superfluous_terms Γ T ≡
    let
      f = λS t R. ∃s ∈ set S - R. s ≠ t ∧ is_wt_instance_of_cond Γ t s;
      g = λS t (U,R). if f S t R then (U, insert t R) else (t#U, R);
      h = λS. remdups (fst (foldr (g S) S ([],{})))
    in while (λS. h S ≠ S) h T"
```

Definitions: Checking Type-Flaw Resistance

```
definition is_TComp_var_instance_closed where
  "is_TComp_var_instance_closed Γ M ≡ ∀x ∈ fv_set (set M). is_Fun (Γ (Var x)) ⟶
      list_ex (λt. is_Fun t ∧ Γ t = Γ (Var x) ∧ list_all is_Var (args t) ∧ distinct (args t)) M"

definition finite_SMP_representation where
  "finite_SMP_representation arity Ana Γ M ≡
    list_all (wf_trm' arity) M ∧
    has_all_wt_instances_of Γ (subterms_set (set M)) (set M) ∧
    has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ‘ set M)) (set M) ∧
    is_TComp_var_instance_closed Γ M"
```

```
definition comp_tfr_set where
  "comp_tfr_set arity Ana Γ M ≡
    finite_SMP_representation arity Ana Γ M ∧
    (let δ = var_rename (max_var_set (fv_set (set M)))
     in ∀s ∈ set M. ∀t ∈ set M. is_Fun s ∧ is_Fun t ∧ Γ s ≠ Γ t ⟶ mgu s (t · δ) = None)"

fun comp_tfr_stp where
  "comp_tfr_stp Γ (⟨_: t ≐ t'⟩_st) = (mgu t t' ≠ None ⟶ Γ t = Γ t')"
| "comp_tfr_stp Γ (∀X⟨∨≠: F⟩_st) = (
    (∀x ∈ fv_pairs F - set X. is_Var (Γ (Var x))) ∨
    (∀u ∈ subterms_set (trms_pairs F).
      is_Fun u ⟶ (args u = [] ∨ (∃s ∈ set (args u). s ∉ Var ` set X))))"
| "comp_tfr_stp _ _ = True"

definition comp_tfr_st where
  "comp_tfr_st arity Ana Γ M S ≡
    list_all (comp_tfr_stp Γ) S ∧
    list_all (wf_trm' arity) (trms_list_st S) ∧
    has_all_wt_instances_of Γ (trms_st S) (set M) ∧
    comp_tfr_set arity Ana Γ M"
```

Small Lemmata

```
lemma less_Suc_max_var_set:
  assumes z: "z ∈ X"
    and X: "finite X"
  shows "snd z < Suc (max_var_set X)"
proof -
  have "snd z ∈ snd ` X" using z by simp
  hence "snd z ≤ Max (insert 0 (snd ` X))" using X by simp
  thus ?thesis using X by simp
qed

lemma (in typed_model) finite_SMP_representationD:
  assumes "finite_SMP_representation arity Ana Γ M"
  shows "wf_trms (set M)"
    and "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
    and "has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ` set M)) (set M)"
    and "is_TComp_var_instance_closed Γ M"
using assms unfolding finite_SMP_representation_def list_all_iff wf_trm_code by blast+

lemma (in typed_model) is_wt_instance_of_condD:
  assumes t_instance_s: "is_wt_instance_of_cond Γ t s"
  obtains δ where
    "Γ t = Γ s" "mgu t s = Some δ"
    "inj_on δ (fv t)" "δ ` (fv t) ⊆ range Var"
using t_instance_s unfolding is_wt_instance_of_cond_def Let_def by (cases "mgu t s") fastforce+

lemma (in typed_model) is_wt_instance_of_condD':
  assumes t_wf_trm: "wf_trm t"
    and s_wf_trm: "wf_trm s"
    and t_instance_s: "is_wt_instance_of_cond Γ t s"
  shows "∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"
proof -
  obtain δ where s:
      "Γ t = Γ s" "mgu t s = Some δ"
      "inj_on δ (fv t)" "δ ` (fv t) ⊆ range Var"
    by (metis is_wt_instance_of_condD[OF t_instance_s])

  have 0: "wf_trm t" "wf_trm s" using s(1) t_wf_trm s_wf_trm by auto

  note 1 = mgu_wt_if_same_type[OF s(2) 0 s(1)]
```

```
  note 2 = conjunct1[OF mgu_gives_MGU[OF s(2)]]

  show ?thesis
    using s(1) inj_var_ran_unifiable_has_subst_match[OF 2 s(3,4)]
          wt_subst_compose[OF 1 subst_var_inv_wt[OF 1, of "fv t"]]
          wf_trms_subst_compose[OF mgu_wf_trms[OF s(2) 0] subst_var_inv_wf_trms[of δ "fv t"]]
    by auto
qed

lemma (in typed_model) is_wt_instance_of_condD'':
  assumes s_wf_trm: "wf_trm s"
    and t_instance_s: "is_wt_instance_of_cond Γ t s"
    and t_var: "t = Var x"
  shows "∃y. s = Var y ∧ Γ (Var y) = Γ (Var x)"
proof -
  obtain δ where δ: "wt_subst δ" and s: "Var x = s · δ"
    using is_wt_instance_of_condD'[OF _ s_wf_trm t_instance_s] t_var by auto
  obtain y where y: "s = Var y" using s by (cases s) auto
  show ?thesis using wt_subst_trm''[OF δ] s y by metis
qed

lemma (in typed_model) has_all_wt_instances_ofD:
  assumes N_instance_M: "has_all_wt_instances_of Γ N M"
    and t_in_N: "t ∈ N"
  obtains s δ where
    "s ∈ M" "Γ t = Γ s" "mgu t s = Some δ"
    "inj_on δ (fv t)" "δ ` (fv t) ⊆ range Var"
by (metis t_in_N N_instance_M is_wt_instance_of_condD has_all_wt_instances_of_def)

lemma (in typed_model) has_all_wt_instances_ofD':
  assumes N_wf_trms: "wf_trms N"
    and M_wf_trms: "wf_trms M"
    and N_instance_M: "has_all_wt_instances_of Γ N M"
    and t_in_N: "t ∈ N"
  shows "∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t ∈ M ·_set δ"
using assms is_wt_instance_of_condD' unfolding has_all_wt_instances_of_def by fast

lemma (in typed_model) has_all_wt_instances_ofD'':
  assumes N_wf_trms: "wf_trms N"
    and M_wf_trms: "wf_trms M"
    and N_instance_M: "has_all_wt_instances_of Γ N M"
    and t_in_N: "Var x ∈ N"
  shows "∃y. Var y ∈ M ∧ Γ (Var y) = Γ (Var x)"
using assms is_wt_instance_of_condD'' unfolding has_all_wt_instances_of_def by fast

lemma (in typed_model) has_all_instances_of_if_subset:
  assumes "N ⊆ M"
  shows "has_all_wt_instances_of Γ N M"
using assms inj_onI mgu_same_empty
unfolding has_all_wt_instances_of_def is_wt_instance_of_cond_def
by (smt option.case_eq_if option.discI option.sel subsetD term.discI(1) term.inject(1))

lemma (in typed_model) SMP_I':
  assumes N_wf_trms: "wf_trms N"
    and M_wf_trms: "wf_trms M"
    and N_instance_M: "has_all_wt_instances_of Γ N M"
    and t_in_N: "t ∈ N"
  shows "t ∈ SMP M"
using has_all_wt_instances_ofD'[OF N_wf_trms M_wf_trms N_instance_M t_in_N]
      SMP.Substitution[OF SMP.MP[of _ M]]
by blast
```

Lemma: Proving Type-Flaw Resistance

```
locale typed_model' = typed_model arity public Ana Γ
  for arity::"'fun ⇒ nat"
    and public::"'fun ⇒ bool"
    and Ana::"('fun,(('fun,'atom::finite) term_type × nat)) term
              ⇒ (('fun,(('fun,'atom) term_type × nat)) term list
                 × ('fun,(('fun,'atom) term_type × nat)) term list)"
    and Γ::"('fun,(('fun,'atom) term_type × nat)) term ⇒ ('fun,'atom) term_type"
  +
  assumes Γ_Var_fst: "⋀τ n m. Γ (Var (τ,n)) = Γ (Var (τ,m))"
    and Ana_const: "⋀c T. arity c = 0 ⟹ Ana (Fun c T) = ([],[])"
    and Ana_subst'_or_Ana_keys_subterm:
      "(∀f T δ K R. Ana (Fun f T) = (K,R) ⟶ Ana (Fun f T ⋅ δ) = (K ⋅list δ,R ⋅list δ)) ∨
       (∀t K R k. Ana t = (K,R) ⟶ k ∈ set K ⟶ k ⊏ t)"
begin

lemma var_rename_inv_comp: "t ⋅ (var_rename n ∘s var_rename_inv n) = t"
proof (induction t)
  case (Fun f T)
  hence "map (λt. t ⋅ var_rename n ∘s var_rename_inv n) T = T" by (simp add: map_idI)
  thus ?case by (metis subst_apply_term.simps(2))
qed (simp add: var_rename_def var_rename_inv_def)

lemma var_rename_fv_disjoint:
  "fv s ∩ fv (t ⋅ var_rename (max_var s)) = {}"
proof -
  have 1: "∀v ∈ fv s. snd v ≤ max_var s" by simp
  have 2: "∀v ∈ fv (t ⋅ var_rename n). snd v > n" for n unfolding var_rename_def by (induct t) auto
  show ?thesis using 1 2 by force
qed

lemma var_rename_fv_set_disjoint:
  assumes "finite M" "s ∈ M"
  shows "fv s ∩ fv (t ⋅ var_rename (max_var_set (fvset M))) = {}"
proof -
  have 1: "∀v ∈ fv s. snd v ≤ max_var_set (fvset M)" using assms
  proof (induction M rule: finite_induct)
    case (insert t M) thus ?case
    proof (cases "t = s")
      case False
      hence "∀v ∈ fv s. snd v ≤ max_var_set (fvset M)" using insert by simp
      moreover have "max_var_set (fvset M) ≤ max_var_set (fvset (insert t M))"
        using insert.hyps(1) insert.prems
        by force
      ultimately show ?thesis by auto
    qed simp
  qed simp

  have 2: "∀v ∈ fv (t ⋅ var_rename n). snd v > n" for n unfolding var_rename_def by (induct t) auto

  show ?thesis using 1 2 by force
qed

lemma var_rename_fv_set_disjoint':
  assumes "finite M"
  shows "fvset M ∩ fvset (N ⋅set var_rename (max_var_set (fvset M))) = {}"
using var_rename_fv_set_disjoint[OF assms] by auto

lemma var_rename_is_renaming[simp]:
  "subst_range (var_rename n) ⊆ range Var"
  "subst_range (var_rename_inv n) ⊆ range Var"
unfolding var_rename_def var_rename_inv_def by auto
```

```
lemma var_rename_wt[simp]:
  "wt_subst (var_rename n)"
  "wt_subst (var_rename_inv n)"
by (auto simp add: var_rename_def var_rename_inv_def wt_subst_def Γ_Var_fst)

lemma var_rename_wt':
  assumes "wt_subst δ" "s = m · δ"
  shows "wt_subst (var_rename_inv n ∘s δ)" "s = m · var_rename n · var_rename_inv n ∘s δ"
using assms(2) wt_subst_compose[OF var_rename_wt(2)[of n] assms(1)] var_rename_inv_comp[of m n]
by force+

lemma var_rename_wf_trms_range[simp]:
  "wf_trms (subst_range (var_rename n))"
  "wf_trms (subst_range (var_rename_inv n))"
using var_rename_is_renaming by fastforce+

lemma Fun_range_case:
  "(∀f T. Fun f T ∈ M ⟶ P f T) ⟷ (∀u ∈ M. case u of Fun f T ⇒ P f T | _ ⇒ True)"
  "(∀f T. Fun f T ∈ M ⟶ P f T) ⟷ (∀u ∈ M. is_Fun u ⟶ P (the_Fun u) (args u))"
by (auto split: "term.splits")

lemma is_TComp_var_instance_closedD:
  assumes x: "∃y ∈ fv_set (set M). Γ (Var x) = Γ (Var y)" "Γ (Var x) = TComp f T"
    and closed: "is_TComp_var_instance_closed Γ M"
  shows "∃g U. Fun g U ∈ set M ∧ Γ (Fun g U) = Γ (Var x) ∧ (∀u ∈ set U. is_Var u) ∧ distinct U"
using assms unfolding is_TComp_var_instance_closed_def list_all_iff list_ex_iff by fastforce

lemma is_TComp_var_instance_closedD':
  assumes "∃y ∈ fv_set (set M). Γ (Var x) = Γ (Var y)" "TComp f T ⊑ Γ (Var x)"
    and closed: "is_TComp_var_instance_closed Γ M"
    and wf: "wf_trms (set M)"
  shows "∃g U. Fun g U ∈ set M ∧ Γ (Fun g U) = TComp f T ∧ (∀u ∈ set U. is_Var u) ∧ distinct U"
using assms(1,2)
proof (induction "Γ (Var x)" arbitrary: x)
  case (Fun g U)
  note IH = Fun.hyps(1)
  have g: "arity g > 0" "public g" using Fun.hyps(2) fun_type_inv[of "Var x"] Γ_Var_fst by simp_all
  then obtain V where V:
      "Fun g V ∈ set M" "Γ (Fun g V) = Γ (Var x)" "∀v ∈ set V. ∃x. v = Var x"
      "distinct V" "length U = length V"
    using is_TComp_var_instance_closedD[OF Fun.prems(1) Fun.hyps(2)[symmetric] closed(1)]
    by (metis Fun.hyps(2) fun_type_id_eq fun_type_length_eq is_VarE)
  hence U: "U = map Γ V" using fun_type[OF g(1), of V] Fun.hyps(2) by simp
  hence 1: "Γ v ∈ set U" when v: "v ∈ set V" for v using v by simp

  have 2: "∃y ∈ fv_set (set M). Γ (Var z) = Γ (Var y)" when z: "Var z ∈ set V" for z
    using V(1) fv_subset_subterms Fun_param_in_subterms[OF z] by fastforce

  show ?case
  proof (cases "TComp f T = Γ (Var x)")
    case False
    then obtain u where u: "u ∈ set U" "TComp f T ⊑ u"
      using Fun.prems(2) Fun.hyps(2) by moura
    then obtain y where y: "Var y ∈ set V" "Γ (Var y) = u" using U V(3) Γ_Var_fst by auto
    show ?thesis using IH[OF _ 2[OF y(1)]] u y(2) by metis
  qed (use V in fastforce)
qed simp

lemma TComp_var_instance_wt_subst_exists:
  assumes gT: "Γ (Fun g T) = TComp g (map Γ U)" "wf_trm (Fun g T)"
    and U: "∀u ∈ set U. ∃y. u = Var y" "distinct U"
  shows "∃ϑ. wt_subst ϑ ∧ wf_trms (subst_range ϑ) ∧ Fun g T = Fun g U · ϑ"
```

```
proof -
  define the_i where "the_i ≡ λy. THE x. x < length U ∧ U ! x = Var y"
  define ϑ where ϑ: "ϑ ≡ λy. if Var y ∈ set U then T ! the_i y else Var y"

  have g: "arity g > 0" using gT(1,2) fun_type_inv(1) by blast

  have UT: "length U = length T" using fun_type_length_eq gT(1) by fastforce

  have 1: "the_i y < length U ∧ U ! the_i y = Var y" when y: "Var y ∈ set U" for y
    using theI'[OF distinct_Ex1[OF U(2) y]] unfolding the_i_def by simp

  have 2: "wt_subst ϑ"
    using ϑ 1 gT(1) fun_type[OF g] UT
    unfolding wt_subst_def
    by (metis (no_types, lifting) nth_map term.inject(2))

  have "∀i<length T. U ! i · ϑ = T ! i"
    using ϑ 1 U(1) UT distinct_Ex1[OF U(2)] in_set_conv_nth
    by (metis (no_types, lifting) subst_apply_term.simps(1))
  hence "T = map (λt. t · ϑ) U" by (simp add: UT nth_equalityI)
  hence 3: "Fun g T = Fun g U · ϑ" by simp

  have "subst_range ϑ ⊆ set T" using ϑ 1 U(1) UT by (auto simp add: subst_domain_def)
  hence 4: "wf_trms (subst_range ϑ)" using gT(2) wf_trm_param by auto

  show ?thesis by (metis 2 3 4)
qed


lemma TComp_var_instance_closed_has_Var:
  assumes closed: "is_TComp_var_instance_closed Γ M"
    and wf_M: "wf_trms (set M)"
    and wf_δx: "wf_trm (δ x)"
    and y_ex: "∃y ∈ fv_set (set M). Γ (Var x) = Γ (Var y)"
    and t: "t ⊑ δ x"
    and δ_wt: "wt_subst δ"
  shows "∃y ∈ fv_set (set M). Γ (Var y) = Γ t"
proof (cases "Γ (Var x)")
  case (Var a)
  hence "t = δ x"
    using t wf_δx δ_wt
    by (metis (full_types) const_type_inv_wf fun_if_subterm subtermeq_Var_const(2) wt_subst_def)
  thus ?thesis using y_ex wt_subst_trm''[OF δ_wt, of "Var x"] by fastforce
next
  case (Fun f T)
  hence Γ_δx: "Γ (δ x) = TComp f T" using wt_subst_trm''[OF δ_wt, of "Var x"] by auto

  show ?thesis
  proof (cases "t = δ x")
    case False
    hence t_subt_δx: "t ⊏ δ x" using t(1) Γ_δx by fastforce

    obtain T' where T': "δ x = Fun f T'" using Γ_δx t_subt_δx fun_type_id_eq by (cases "δ x") auto

    obtain g S where gS: "Fun g S ⊑ δ x" "t ∈ set S" using Fun_ex_if_subterm[OF t_subt_δx] by blast

    have gS_wf: "wf_trm (Fun g S)" by (rule wf_trm_subtermeq[OF wf_δx gS(1)])
    hence "arity g > 0" using gS(2) by (metis length_pos_if_in_set wf_trm_arity)
    hence gS_Γ: "Γ (Fun g S) = TComp g (map Γ S)" using fun_type by blast

    obtain h U where hU:
        "Fun h U ∈ set M" "Γ (Fun h U) = Fun g (map Γ S)" "∀u ∈ set U. is_Var u"
      using is_TComp_var_instance_closedD'[OF y_ex _ closed wf_M]
            subtermeq_imp_subtermtypeeq[OF wf_δx] gS Γ_δx Fun gS_Γ
```

```
      by metis

    obtain y where y: "Var y ∈ set U" "Γ (Var y) = Γ t"
      using hU(3) fun_type_param_ex[OF hU(2) gS(2)] by fast

    have "y ∈ fv_set (set M)" using hU(1) y(1) by force
    thus ?thesis using y(2) closed by metis
  qed (metis y_ex Fun Γ_δx)
qed

lemma TComp_var_instance_closed_has_Fun:
  assumes closed: "is_TComp_var_instance_closed Γ M"
    and wf_M: "wf_trms (set M)"
    and wf_δx: "wf_trm (δ x)"
    and y_ex: "∃y ∈ fv_set (set M). Γ (Var x) = Γ (Var y)"
    and t: "t ⊑ δ x"
    and δ_wt: "wt_subst δ"
    and t_Γ: "Γ t = TComp g T"
    and t_fun: "is_Fun t"
  shows "∃m ∈ set M. ∃ϑ. wt_subst ϑ ∧ wf_trms (subst_range ϑ) ∧ t = m · ϑ ∧ is_Fun m"
proof -
  obtain T'' where T'': "t = Fun g T''" using t_Γ t_fun fun_type_id_eq by blast

  have g: "arity g > 0" using t_Γ fun_type_inv[of t] by simp_all

  have "TComp g T ⊑ Γ (Var x)" using δ_wt t t_Γ
    by (metis wf_δx subtermeq_imp_subtermtypeeq wt_subst_def)
  then obtain U where U:
      "Fun g U ∈ set M" "Γ (Fun g U) = TComp g T" "∀u ∈ set U. ∃y. u = Var y"
      "distinct U" "length T'' = length U"
    using is_TComp_var_instance_closedD'[OF y_ex _ closed wf_M]
    by (metis t_Γ T'' fun_type_id_eq fun_type_length_eq is_VarE)
  hence UT': "T = map Γ U" using fun_type[OF g, of U] by simp

  show ?thesis
    using TComp_var_instance_wt_subst_exists UT' T'' U(1,3,4) t t_Γ wf_δx wf_trm_subtermeq
    by (metis term.disc(2))
qed

lemma TComp_var_and_subterm_instance_closed_has_subterms_instances:
  assumes M_var_inst_cl: "is_TComp_var_instance_closed Γ M"
    and M_subterms_cl: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
    and M_wf: "wf_trms (set M)"
    and t: "t ⊑_set set M"
    and s: "s ⊑ t · δ"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)"
  shows "∃m ∈ set M. ∃ϑ. wt_subst ϑ ∧ wf_trms (subst_range ϑ) ∧ s = m · ϑ"
using subterm_subst_unfold[OF s]
proof
  assume "∃s'. s' ⊑ t ∧ s = s' · δ"
  then obtain s' where s': "s' ⊑ t" "s = s' · δ" by blast
  then obtain ϑ where ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" "s' ∈ set M ·_set ϑ"
    using t has_all_wt_instances_ofD'[OF wf_trms_subterms[OF M_wf] M_wf M_subterms_cl]
          term.order_trans[of s' t]
    by blast
  then obtain m where m: "m ∈ set M" "s' = m · ϑ" by blast

  have "s = m · (ϑ ∘_s δ)" using s'(2) m(2) by simp
  thus ?thesis
    using m(1) wt_subst_compose[OF ϑ(1) δ(1)] wf_trms_subst_compose[OF ϑ(2) δ(2)] by blast
next
  assume "∃x ∈ fv t. s ⊏ δ x"
  then obtain x where x: "x ∈ fv t" "s ⊏ δ x" "s ⊑ δ x" by blast
```

```
  note 0 = TComp_var_instance_closed_has_Var[OF M_var_inst_cl M_wf]
  note 1 = has_all_wt_instances_ofD''[OF wf_trms_subterms[OF M_wf] M_wf M_subterms_cl]

  have δx_wf: "wf_{trm} (δ x)" and s_wf_trm: "wf_{trm} s"
    using δ(2) wf_trm_subterm[OF _ x(2)] by fastforce+

  have x_fv_ex: "∃y ∈ fv_{set} (set M). Γ (Var x) = Γ (Var y)"
    using x(1) s fv_subset_subterms[OF t] by auto

  obtain y where y: "y ∈ fv_{set} (set M)" "Γ (Var y) = Γ s"
    using 0[of δ x s, OF δx_wf x_fv_ex x(3) δ(1)] by metis
  then obtain z where z: "Var z ∈ set M" "Γ (Var z) = Γ s"
    using 1[of y] vars_iff_subtermeq_set[of y "set M"] by metis

  define ϑ where "ϑ ≡ Var(z := s)::('fun, ('fun, 'atom) term × nat) subst"

  have "wt_{subst} ϑ" "wf_{trms} (subst_range ϑ)" "s = Var z · ϑ"
    using z(2) s_wf_trm unfolding ϑ_def wt_{subst}_def by force+
  thus ?thesis using z(1) by blast
qed

context
begin
private lemma SMP_D_aux1:
  assumes "t ∈ SMP (set M)"
    and closed: "has_all_wt_instances_of Γ (subterms_{set} (set M)) (set M)"
                "is_TComp_var_instance_closed Γ M"
    and wf_M: "wf_{trms} (set M)"
  shows "∀x ∈ fv t. ∃y ∈ fv_{set} (set M). Γ (Var y) = Γ (Var x)"
using assms(1)
proof (induction t rule: SMP.induct)
  case (MP t) show ?case
  proof
    fix x assume x: "x ∈ fv t"
    hence "Var x ∈ subterms_{set} (set M)" using MP.hyps vars_iff_subtermeq by fastforce
    then obtain δ s where δ: "wt_{subst} δ" "wf_{trms} (subst_range δ)"
        and s: "s ∈ set M" "Var x = s · δ"
      using has_all_wt_instances_ofD'[OF wf_trms_subterms[OF wf_M] wf_M closed(1)] by blast
    then obtain y where y: "s = Var y" by (cases s) auto
    thus "∃y ∈ fv_{set} (set M). Γ (Var y) = Γ (Var x)"
      using s wt_subst_trm''[OF δ(1), of "Var y"] by force
  qed
next
  case (Subterm t t')
  hence "fv t' ⊆ fv t" using subtermeq_vars_subset by auto
  thus ?case using Subterm.IH by auto
next
  case (Substitution t δ)
  note IH = Substitution.IH
  show ?case
  proof
    fix x assume x: "x ∈ fv (t · δ)"
    then obtain y where y: "y ∈ fv t" "Γ (Var x) ⊑ Γ (Var y)"
      using Substitution.hyps(2,3)
      by (metis subst_apply_img_var subtermeqI' subtermeq_imp_subtermtypeeq
                vars_iff_subtermeq wt_{subst}_def wf_trm_subst_rangeD)
    let ?P = "λx. ∃y ∈ fv_{set} (set M). Γ (Var y) = Γ (Var x)"
    show "?P x" using y IH
    proof (induction "Γ (Var y)" arbitrary: y t)
      case (Var a)
      hence "Γ (Var x) = Γ (Var y)" by auto
      thus ?case using Var(2,4) by auto
```

```
    next
      case (Fun f T)
      obtain z where z: "∃w ∈ fv_set (set M). Γ (Var z) = Γ (Var w)" "Γ (Var z) = Γ (Var y)"
        using Fun.prems(1,3) by blast
      show ?case
      proof (cases "Γ (Var x) = Γ (Var y)")
        case True thus ?thesis using Fun.prems by auto
      next
        case False
        then obtain τ where τ: "τ ∈ set T" "Γ (Var x) ⊑ τ" using Fun.prems(2) Fun.hyps(2) by auto
        then obtain U where U:
            "Fun f U ∈ set M" "Γ (Fun f U) = Γ (Var z)" "∀u ∈ set U. ∃v. u = Var v" "distinct U"
          using is_TComp_var_instance_closedD'[OF z(1) _ closed(2) wf_M] Fun.hyps(2) z(2)
          by (metis fun_type_id_eq subtermeqI' is_VarE)
        hence 1: "∀x ∈ fv (Fun f U). ∃y ∈ fv_set (set M). Γ (Var y) = Γ (Var x)" by force

        have "arity f > 0" using U(2) z(2) Fun.hyps(2) fun_type_inv(1) by metis
        hence "Γ (Fun f U) = TComp f (map Γ U)" using fun_type by auto
        then obtain u where u: "Var u ∈ set U" "Γ (Var u) = τ"
          using τ(1) U(2,3) z(2) Fun.hyps(2) by auto
        show ?thesis
          using Fun.hyps(1)[of u "Fun f U"] u τ 1
          by force
      qed
    qed
  qed
next
  case (Ana t K T k)
  have "fv k ⊆ fv t" using Ana_keys_fv[OF Ana.hyps(2)] Ana.hyps(3) by auto
  thus ?case using Ana.IH by auto
qed

private lemma SMP_D_aux2:
  fixes t::"('fun, ('fun, 'atom) term × nat) term"
  assumes t_SMP: "t ∈ SMP (set M)"
    and t_Var: "∃x. t = Var x"
    and M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
  shows "∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = m · δ"
proof -
  have M_wf: "wf_trms (set M)"
      and M_var_inst_cl: "is_TComp_var_instance_closed Γ M"
      and M_subterms_cl: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
      and M_Ana_cl: "has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ` set M)) (set M)"
    using finite_SMP_representationD[OF M_SMP_repr] by blast+

  have M_Ana_wf: "wf_trms (⋃ ((set ∘ fst ∘ Ana) ` set M))"
  proof
    fix k assume "k ∈ ⋃((set ∘ fst ∘ Ana) ` set M)"
    then obtain m where m: "m ∈ set M" "k ∈ set (fst (Ana m))" by force
    thus "wf_trm k" using M_wf Ana_keys_wf'[of m "fst (Ana m)" _ k] surjective_pairing by blast
  qed

  note 0 = has_all_wt_instances_ofD'[OF wf_trms_subterms[OF M_wf] M_wf M_subterms_cl]
  note 1 = has_all_wt_instances_ofD'[OF M_Ana_wf M_wf M_Ana_cl]

  obtain x y where x: "t = Var x" and y: "y ∈ fv_set (set M)" "Γ (Var y) = Γ (Var x)"
    using t_Var SMP_D_aux1[OF t_SMP M_subterms_cl M_var_inst_cl M_wf] by fastforce
  then obtain m δ where m: "m ∈ set M" "m · δ = Var y" and δ: "wt_subst δ"
    using 0[of "Var y"] vars_iff_subtermeq_set[of y "set M"] by fastforce
  obtain z where z: "m = Var z" using m(2) by (cases m) auto

  define ϑ where "ϑ ≡ Var(z := Var x)::('fun, ('fun, 'atom) term × nat) subst"
```

```
  have "Γ (Var z) = Γ (Var x)" using y(2) m(2) z wt_subst_trm''[OF δ, of m] by argo
  hence "wt_subst ϑ" "wf_trms (subst_range ϑ)" unfolding ϑ_def wt_subst_def by force+
  moreover have "t = m · ϑ" using x z unfolding ϑ_def by simp
  ultimately show ?thesis using m(1) by blast
qed

private lemma SMP_D_aux3:
  assumes hyps: "t' ⊑ t" and wf_t: "wf_trm t" and prems: "is_Fun t'"
    and IH:
      "((∃f. t = Fun f []) ∧ (∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = m · δ)) ∨
       (∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = m · δ ∧ is_Fun m)"
    and M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
  shows "((∃f. t' = Fun f []) ∧ (∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t' = m · δ))
∨
         (∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t' = m · δ ∧ is_Fun m)"
proof (cases "∃f. t = Fun f [] ∨ t' = Fun f []")
  case True
  have M_wf: "wf_trms (set M)"
    and M_var_inst_cl: "is_TComp_var_instance_closed Γ M"
    and M_subterms_cl: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
    and M_Ana_cl: "has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ` set M)) (set M)"
  using finite_SMP_representationD[OF M_SMP_repr] by blast+

  note 0 = has_all_wt_instances_ofD'[OF wf_trms_subterms[OF M_wf] M_wf M_subterms_cl]
  note 1 = TComp_var_instance_closed_has_Fun[OF M_var_inst_cl M_wf]
  note 2 = TComp_var_and_subterm_instance_closed_has_subterms_instances[
             OF M_var_inst_cl M_subterms_cl M_wf]

  have wf_t': "wf_trm t'" using hyps wf_t wf_trm_subterm by blast

  obtain c where "t = Fun c [] ∨ t' = Fun c []" using True by moura
  thus ?thesis
  proof
    assume c: "t' = Fun c []"
    show ?thesis
    proof (cases "∃f. t = Fun f []")
      case True
      hence "t = t'" using c hyps by force
      thus ?thesis using IH by fast
    next
      case False
      note F = this
      then obtain m δ where m: "m ∈ set M" "t = m · δ"
          and δ: "wt_subst δ" "wf_trms (subst_range δ)"
        using IH by blast

      show ?thesis using subterm_subst_unfold[OF hyps[unfolded m(2)]]
      proof
        assume "∃m'. m' ⊑ m ∧ t' = m' · δ"
        then obtain m' where m': "m' ⊑ m" "t' = m' · δ" by moura
        obtain n ϑ where n: "n ∈ set M" "m' = n · ϑ" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
          using 0[of m'] m(1) m'(1) by blast
        have "t' = n · (ϑ ∘s δ)" using m'(2) n(2) by auto
        thus ?thesis
          using c n(1) wt_subst_compose[OF ϑ(1) δ(1)] wf_trms_subst_compose[OF ϑ(2) δ(2)] by blast
      next
        assume "∃x ∈ fv m. t' ⊏ δ x"
        then obtain x where x: "x ∈ fv m" "t' ⊏ δ x" "t' ⊑ δ x" by moura
        have δx_wf: "wf_trm (δ x)" using δ(2) by fastforce

        have x_fv_ex: "∃y ∈ fv_set (set M). Γ (Var x) = Γ (Var y)" using x m by auto

        show ?thesis
```

```
        proof (cases "Γ t'")
          case (Var a)
          show ?thesis
            using c m 2[OF _ hyps[unfolded m(2)] δ]
            by fast
        next
          case (Fun g S)
          show ?thesis
            using c 1[of δ x t', OF δx_wf x_fv_ex x(3) δ(1) Fun]
            by blast
        qed
      qed
    qed
  qed (use IH hyps in simp)
next
  case False
  note F = False
  then obtain m δ where m:
      "m ∈ set M" "wt_subst δ" "t = m · δ" "is_Fun m" "wf_trms (subst_range δ)"
    using IH by moura
  obtain f T where fT: "t' = Fun f T" "arity f > 0" "Γ t' = TComp f (map Γ T)"
    using F prems fun_type wf_trm_subtermeq[OF wf_t hyps]
    by (metis is_FunE length_greater_0_conv subtermeqI' wf_trm_def)

  have closed: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
               "is_TComp_var_instance_closed Γ M"
    using M_SMP_repr unfolding finite_SMP_representation_def by metis+

  have M_wf: "wf_trms (set M)"
    using finite_SMP_representationD[OF M_SMP_repr] by blast

  show ?thesis
  proof (cases "∃x ∈ fv m. t' ⊑ δ x")
    case True
    then obtain x where x: "x ∈ fv m" "t' ⊑ δ x" by moura
    have 1: "x ∈ fv_set (set M)" using m(1) x(1) by auto
    have 2: "is_Fun (δ x)" using prems x(2) by auto
    have 3: "wf_trm (δ x)" using m(5) by (simp add: wf_trm_subst_rangeD)
    have "¬(∃f. δ x = Fun f [])" using F x(2) by auto
    hence "∃f T. Γ (Var x) = TComp f T" using 2 3 m(2)
      by (metis (no_types) fun_type is_FunE length_greater_0_conv subtermeqI' wf_trm_def wt_subst_def)
    moreover have "∃f T. Γ t' = Fun f T"
      using False prems wf_trm_subtermeq[OF wf_t hyps]
      by (metis (no_types) fun_type is_FunE length_greater_0_conv subtermeqI' wf_trm_def)
    ultimately show ?thesis
      using TComp_var_instance_closed_has_Fun 1 x(2) m(2) prems closed 3 M_wf
      by metis
  next
    case False
    then obtain m' where m': "m' ⊑ m" "t' = m' · δ" "is_Fun m'"
      using hyps m(3) subterm_subst_not_img_subterm
      by blast
    then obtain ϑ m'' where ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" "m'' ∈ set M" "m' = m'' · ϑ"
      using m(1) has_all_wt_instances_ofD'[OF wf_trms_subterms[OF M_wf] M_wf closed(1)] by blast
    hence t'_m'': "t' = m'' · ϑ ∘s δ" using m'(2) by fastforce

    note ϑδ = wt_subst_compose[OF ϑ(1) m(2)] wf_trms_subst_compose[OF ϑ(2) m(5)]

    show ?thesis
    proof (cases "is_Fun m''")
      case True thus ?thesis using ϑ(3,4) m'(2,3) m(4) fT t'_m'' ϑδ by blast
    next
      case False
```

```
      then obtain x where x: "m'' = Var x" by moura
      hence "∃y ∈ fv_set (set M). Γ (Var x) = Γ (Var y)" "t' ⊑ (ϑ ∘s δ) x"
            "Γ (Var x) = Fun f (map Γ T)" "wf_trm ((ϑ ∘s δ) x)"
        using ϑδ t'_m'' ϑ(3) fv_subset[OF ϑ(3)] fT(3) subst_apply_term.simps(1)[of x "ϑ ∘s δ"]
              wt_subst_trm''[OF ϑδ(1), of "Var x"]
        by (fastforce, blast, argo, fastforce)
      thus ?thesis
        using x TComp_var_instance_closed_has_Fun[
                of M "ϑ ∘s δ" x t' f "map Γ T", OF closed(2) M_wf _ _ _ ϑδ(1) fT(3) prems]
        by blast
    qed
  qed
qed


lemma SMP_D:
  assumes "t ∈ SMP (set M)" "is_Fun t"
    and M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
  shows "((∃f. t = Fun f []) ∧ (∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = m · δ)) ∨
         (∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = m · δ ∧ is_Fun m)"
proof -
  have wf_M: "wf_trms (set M)"
      and closed: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
                  "has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ` set M)) (set M)"
                  "is_TComp_var_instance_closed Γ M"
    using finite_SMP_representationD[OF M_SMP_repr] by blast+

  show ?thesis using assms(1,2)
  proof (induction t rule: SMP.induct)
    case (MP t)
    moreover have "wt_subst Var" "wf_trms (subst_range Var)" "t = t · Var" by simp_all
    ultimately show ?case by blast
  next
    case (Subterm t t')
    hence t_fun: "is_Fun t" by auto
    note * = Subterm.hyps(2) SMP_wf_trm[OF Subterm.hyps(1) wf_M(1)]
             Subterm.prems Subterm.IH[OF t_fun] M_SMP_repr
    show ?case by (rule SMP_D_aux3[OF *])
  next
    case (Substitution t δ)
    have wf: "wf_trm t" by (metis Substitution.hyps(1) wf_M(1) SMP_wf_trm)
    hence wf': "wf_trm (t · δ)" using Substitution.hyps(3) wf_trm_subst by blast
    show ?case
    proof (cases "Γ t")
      case (Var a)
      hence 1: "Γ (t · δ) = TAtom a" using Substitution.hyps(2) by (metis wt_subst_trm'')
      then obtain c where c: "t · δ = Fun c []"
        using TAtom_term_cases[OF wf' 1] Substitution.prems by fastforce
      hence "(∃x. t = Var x) ∨ t = t · δ" by (cases t) auto
      thus ?thesis
      proof
        assume t_Var: "∃x. t = Var x"
        then obtain x where x: "t = Var x" "δ x = Fun c []" "Γ (Var x) = TAtom a"
          using c 1 wt_subst_trm''[OF Substitution.hyps(2), of t] by force

        obtain m ϑ where m: "m ∈ set M" "t = m · ϑ" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
          using SMP_D_aux2[OF Substitution.hyps(1) t_Var M_SMP_repr] by moura

        have "m · (ϑ ∘s δ) = Fun c []" using c m(2) by auto
        thus ?thesis
          using c m(1) wt_subst_compose[OF ϑ(1) Substitution.hyps(2)]
                wf_trms_subst_compose[OF ϑ(2) Substitution.hyps(3)]
          by metis
      qed (use c Substitution.IH in auto)
```

```
  next
    case (Fun f T)
    hence 1: "Γ (t · δ) = TComp f T" using Substitution.hyps(2) by (metis wt_subst_trm'')
    have 2: "¬(∃f. t = Fun f [])" using Fun TComp_term_cases[OF wf] by auto
    obtain T'' where T'': "t · δ = Fun f T''"
      using 1 2 fun_type_id_eq Fun Substitution.prems
      by fastforce
    have f: "arity f > 0" "public f" using fun_type_inv[OF 1] by metis+

    show ?thesis
    proof (cases t)
      case (Fun g U)
      then obtain m ϑ where m:
          "m ∈ set M" "wt_subst ϑ" "t = m · ϑ" "is_Fun m" "wf_trms (subst_range ϑ)"
        using Substitution.IH Fun 2 by moura
      have "wt_subst (ϑ ∘s δ)" "t · δ = m · (ϑ ∘s δ)" "wf_trms (subst_range (ϑ ∘s δ))"
        using wt_subst_compose[OF m(2) Substitution.hyps(2)] m(3)
              wf_trms_subst_compose[OF m(5) Substitution.hyps(3)]
        by auto
      thus ?thesis using m(1,4) by metis
    next
      case (Var x)
      then obtain y where y: "y ∈ fv_set (set M)" "Γ (Var y) = Γ (Var x)"
        using SMP_D_aux1[OF Substitution.hyps(1) closed(1,3) wf_M] Fun
        by moura
      hence 3: "Γ (Var y) = TComp f T" using Var Fun Γ_Var_fst by simp

      obtain h V where V:
          "Fun h V ∈ set M" "Γ (Fun h V) = Γ (Var y)" "∀u ∈ set V. ∃z. u = Var z" "distinct V"
        by (metis is_VarE is_TComp_var_instance_closedD[OF _ 3 closed(3)] y(1))
      moreover have "length T'' = length V" using 3 V(2) fun_type_length_eq 1 T'' by metis
      ultimately have TV: "T = map Γ V"
        by (metis fun_type[OF f(1)] 3 fun_type_id_eq term.inject(2))

      obtain ϑ where ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" "t · δ = Fun h V · ϑ"
        using TComp_var_instance_wt_subst_exists 1 3 T'' TV V(2,3,4) wf'
        by (metis fun_type_id_eq)

      have 9: "Γ (Fun h V) = Γ (δ x)" using y(2) Substitution.hyps(2) V(2) 1 3 Var by auto

      show ?thesis using Var ϑ 9 V(1) by force
    qed
  qed
next
  case (Ana t K T k)
  have 1: "is_Fun t" using Ana.hyps(2,3) by auto
  then obtain f U where U: "t = Fun f U" by moura

  have 2: "fv k ⊆ fv t" using Ana_keys_fv[OF Ana.hyps(2)] Ana.hyps(3) by auto

  have wf_t: "wf_trm t"
    using SMP_wf_trm[OF Ana.hyps(1)] wf_trm_code wf_M
    by auto
  hence wf_k: "wf_trm k"
    using Ana_keys_wf'[OF Ana.hyps(2)] wf_trm_code Ana.hyps(3)
    by auto

  have wf_M_keys: "wf_trms (⋃((set ∘ fst ∘ Ana) ` set M))"
  proof
    fix t assume "t ∈ (⋃((set ∘ fst ∘ Ana) ` set M))"
    then obtain s where s: "s ∈ set M" "t ∈ (set ∘ fst ∘ Ana) s" by blast
    obtain K R where KR: "Ana s = (K,R)" by (metis surj_pair)
    hence "t ∈ set K" using s(2) by simp
```

```
      thus "wf_trm t" using Ana_keys_wf'[OF KR] wf_M s(1) by blast
    qed

    show ?case using Ana_subst'_or_Ana_keys_subterm
    proof
      assume "∀t K T k. Ana t = (K, T) ⟶ k ∈ set K ⟶ k ⊏ t"
      hence *: "k ⊑ t" using Ana.hyps(2,3) by auto
      show ?thesis by (rule SMP_D_aux3[OF * wf_t Ana.prems Ana.IH[OF 1] M_SMP_repr])
    next
      assume Ana_subst':
          "∀f T δ K M. Ana (Fun f T) = (K, M) ⟶ Ana (Fun f T · δ) = (K ·_list δ, M ·_list δ)"

      have "arity f > 0" using Ana_const[of f U] U Ana.hyps(2,3) by fastforce
      hence "U ≠ []" using wf_t U unfolding wf_trm_def by force
      then obtain m δ where m: "m ∈ set M" "wt_subst δ" "wf_trms (subst_range δ)" "t = m · δ" "is_Fun
m"
        using Ana.IH[OF 1] U by auto
      hence "Ana (t · δ) = (K ·_list δ,T ·_list δ)" using Ana_subst' U Ana.hyps(2) by auto
      obtain Km Tm where Ana_m: "Ana m = (Km,Tm)" by moura
      hence "Ana (m · δ) = (Km ·_list δ,Tm ·_list δ)"
        using Ana_subst' U m(4) is_FunE[OF m(5)] Ana.hyps(2)
        by metis
      then obtain km where km: "km ∈ set Km" "k = km · δ" using Ana.hyps(2,3) m(4) by auto
      then obtain ϑ km' where ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
          and km': "km' ∈ set M" "km = km' · ϑ"
        using Ana_m m(1) has_all_wt_instances_ofD'[OF wf_M_keys wf_M closed(2), of km] by force

      have kϑδ: "k = km' · ϑ ∘s δ" "wt_subst (ϑ ∘s δ)" "wf_trms (subst_range (ϑ ∘s δ))"
        using km(2) km' wt_subst_compose[OF ϑ(1) m(2)] wf_trms_subst_compose[OF ϑ(2) m(3)]
        by auto

      show ?case
      proof (cases "is_Fun km'")
        case True thus ?thesis using kϑδ km'(1) by blast
      next
        case False
        note F = False
        then obtain x where x: "km' = Var x" by auto
        hence 3: "x ∈ fv_set (set M)" using fv_subset[OF km'(1)] by auto
        obtain kf kT where kf: "k = Fun kf kT" using Ana.prems by auto
        show ?thesis
        proof (cases "kT = []")
          case True thus ?thesis using kϑδ(1) kϑδ(2) kϑδ(3) kf km'(1) by blast
        next
          case False
          hence 4: "arity kf > 0" using wf_k kf TAtom_term_cases const_type by fastforce
          then obtain kT' where kT': "Γ k = TComp kf kT'" by (simp add: fun_type kf)
          then obtain V where V:
              "Fun kf V ∈ set M" "Γ (Fun kf V) = Γ (Var x)" "∀u ∈ set V. ∃v. u = Var v"
              "distinct V" "is_Fun (Fun kf V)"
            using is_TComp_var_instance_closedD[OF _ _ closed(3), of x]
                  x m(2) kϑδ(1) 3 wt_subst_trm''[OF kϑδ(2)]
            by (metis fun_type_id_eq term.disc(2) is_VarE)
          have 5: "kT' = map Γ V"
            using fun_type[OF 4] x kT' kϑδ m(2) V(2)
            by (metis term.inject(2) wt_subst_trm'')
          thus ?thesis
            using TComp_var_instance_wt_subst_exists wf_k kf 4 V(3,4) kT' V(1,5)
            by metis
        qed
      qed
    qed
  qed
```

```
qed

lemma SMP_D':
  fixes M
  defines "δ ≡ var_rename (max_var_set (fv_set (set M)))"
  assumes M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
    and s: "s ∈ SMP (set M)" "is_Fun s" "∄f. s = Fun f []"
    and t: "t ∈ SMP (set M)" "is_Fun t" "∄f. t = Fun f []"
  obtains σ s0 ϑ t0
  where "wt_subst σ" "wf_trms (subst_range σ)" "s0 ∈ set M" "is_Fun s0" "s = s0 · σ" "Γ s = Γ s0"
    and "wt_subst ϑ" "wf_trms (subst_range ϑ)" "t0 ∈ set M" "is_Fun t0" "t = t0 · δ · ϑ" "Γ t = Γ t0"
proof -
  obtain σ s0 where
      s0: "wt_subst σ" "wf_trms (subst_range σ)" "s0 ∈ set M" "s = s0 · σ" "is_Fun s0"
    using s(3) SMP_D[OF s(1,2) M_SMP_repr] unfolding δ_def by metis

  obtain ϑ t0 where t0:
      "wt_subst ϑ" "wf_trms (subst_range ϑ)" "t0 ∈ set M" "t = t0 · δ · ϑ" "is_Fun t0"
    using t(3) SMP_D[OF t(1,2) M_SMP_repr] var_rename_wt'[of _ t]
          wf_trms_subst_compose_Var_range(1)[OF _ var_rename_is_renaming(2)]
    unfolding δ_def by metis

  have "Γ s = Γ s0" "Γ t = Γ (t0 · δ)" "Γ (t0 · δ) = Γ t0"
    using s0 t0 wt_subst_trm'' by (metis, metis, metis δ_def var_rename_wt(1))
  thus ?thesis using s0 t0 that by simp
qed

lemma SMP_D'':
  fixes t::"('fun, ('fun, 'atom) term × nat) term"
  assumes t_SMP: "t ∈ SMP (set M)"
    and M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
  shows "∃m ∈ set M. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = m · δ"
proof (cases "(∃x. t = Var x) ∨ (∃c. t = Fun c [])")
  case True
  have M_wf: "wf_trms (set M)"
      and M_var_inst_cl: "is_TComp_var_instance_closed Γ M"
      and M_subterms_cl: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
      and M_Ana_cl: "has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ` set M)) (set M)"
    using finite_SMP_representationD[OF M_SMP_repr] by blast+

  have M_Ana_wf: "wf_trms (⋃ ((set ∘ fst ∘ Ana) ` set M))"
  proof
    fix k assume "k ∈ ⋃((set ∘ fst ∘ Ana) ` set M)"
    then obtain m where m: "m ∈ set M" "k ∈ set (fst (Ana m))" by force
    thus "wf_trm k" using M_wf Ana_keys_wf'[of m "fst (Ana m)" _ k] surjective_pairing by blast
  qed

  show ?thesis using True
  proof
    assume "∃x. t = Var x"
    then obtain x y where x: "t = Var x" and y: "y ∈ fv_set (set M)" "Γ (Var y) = Γ (Var x)"
      using SMP_D_aux1[OF t_SMP M_subterms_cl M_var_inst_cl M_wf] by fastforce
    then obtain m δ where m: "m ∈ set M" "m · δ = Var y" and δ: "wt_subst δ"
      using has_all_wt_instances_ofD'[OF wf_trms_subterms[OF M_wf] M_wf M_subterms_cl, of "Var y"]
            vars_iff_subtermeq_set[of y "set M"]
      by fastforce

    obtain z where z: "m = Var z" using m(2) by (cases m) auto

    define ϑ where "ϑ ≡ Var(z := Var x)::('fun, ('fun, 'atom) term × nat) subst"

    have "Γ (Var z) = Γ (Var x)" using y(2) m(2) z wt_subst_trm''[OF δ, of m] by argo
    hence "wt_subst ϑ" "wf_trms (subst_range ϑ)" unfolding ϑ_def wt_subst_def by force+
```

```
    moreover have "t = m · ϑ" using x z unfolding ϑ_def by simp
    ultimately show ?thesis using m(1) by blast
  qed (use SMP_D[OF t_SMP _ M_SMP_repr] in blast)
qed (use SMP_D[OF t_SMP _ M_SMP_repr] in blast)
end

lemma tfr_set_if_comp_tfr_set:
  assumes "comp_tfr_set arity Ana Γ M"
  shows "tfr_set (set M)"
proof -
  let ?δ = "var_rename (max_var_set (fv_set (set M)))"
  have M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
    by (metis comp_tfr_set_def assms)

  have M_finite: "finite (set M)"
    using assms card_gt_0_iff unfolding comp_tfr_set_def by blast

  show ?thesis
  proof (unfold tfr_set_def; intro ballI impI)
    fix s t assume "s ∈ SMP (set M) - Var`𝒱" "t ∈ SMP (set M) - Var`𝒱"
    hence st: "s ∈ SMP (set M)" "is_Fun s" "t ∈ SMP (set M)" "is_Fun t" by auto
    have "¬(∃δ. Unifier δ s t)" when st_type_neq: "Γ s ≠ Γ t"
    proof (cases "∃f. s = Fun f [] ∨ t = Fun f []")
      case False
      then obtain σ s0 ϑ t0 where
            s0: "s0 ∈ set M" "is_Fun s0" "s = s0 · σ" "Γ s = Γ s0"
        and t0: "t0 ∈ set M" "is_Fun t0" "t = t0 · ?δ · ϑ" "Γ t = Γ t0"
        using SMP_D'[OF M_SMP_repr st(1,2) _ st(3,4)] by metis
      hence "¬(∃δ. Unifier δ s0 (t0 · ?δ))"
        using assms mgu_None_is_subst_neq st_type_neq wt_subst_trm''[OF var_rename_wt(1)]
        unfolding comp_tfr_set_def Let_def by metis
      thus ?thesis
        using vars_term_disjoint_imp_unifier[OF var_rename_fv_set_disjoint[OF M_finite]] s0(1) t0(1)
        unfolding s0(3) t0(3) by (metis (no_types, hide_lams) subst_subst_compose)
    qed (use st_type_neq st(2,4) in auto)
    thus "Γ s = Γ t" when "∃δ. Unifier δ s t" by (metis that)
  qed
qed

lemma tfr_set_if_comp_tfr_set':
  assumes "let N = SMP0 Ana Γ M in set M ⊆ set N ∧ comp_tfr_set arity Ana Γ N"
  shows "tfr_set (set M)"
by (rule tfr_subset(2)[
          OF tfr_set_if_comp_tfr_set[OF conjunct2[OF assms[unfolded Let_def]]]
             conjunct1[OF assms[unfolded Let_def]]])

lemma tfr_stp_is_comp_tfr_stp: "tfr_stp a = comp_tfr_stp Γ a"
proof (cases a)
  case (Equality ac t t')
  thus ?thesis
    using mgu_always_unifies[of t _ t'] mgu_gives_MGU[of t t']
    by auto
next
  case (Inequality X F)
  thus ?thesis
    using tfr_stp.simps(2)[of X F]
          comp_tfr_stp.simps(2)[of Γ X F]
          Fun_range_case(2)[of "subterms_set (trms_pairs F)"]
    unfolding is_Var_def
    by auto
qed auto

lemma tfr_st_if_comp_tfr_st:
```

```
  assumes "comp_tfr_st arity Ana Γ M S"
  shows "tfr_st S"
unfolding tfr_st_def
proof
  have comp_tfr_set_M: "comp_tfr_set arity Ana Γ M"
    using assms unfolding comp_tfr_st_def by blast

  have wf_trms_M: "wf_trms (set M)"
      and wf_trms_S: "wf_trms (trms_st S)"
      and S_trms_instance_M: "has_all_wt_instances_of Γ (trms_st S) (set M)"
    using assms wf_trm_code trms_list_st_is_trms_st
    unfolding comp_tfr_st_def comp_tfr_set_def finite_SMP_representation_def list_all_iff
    by blast+

  show "tfr_set (trms_st S)"
    using tfr_subset(3)[OF tfr_set_if_comp_tfr_set[OF comp_tfr_set_M] SMP_SMP_subset]
          SMP_I'[OF wf_trms_S wf_trms_M S_trms_instance_M]
    by blast

  have "list_all (comp_tfr_stp Γ) S" by (metis assms comp_tfr_st_def)
  thus "list_all tfr_stp S" by (induct S) (simp_all add: tfr_stp_is_comp_tfr_stp)
qed

lemma tfr_st_if_comp_tfr_st':
  assumes "comp_tfr_st arity Ana Γ (SMP0 Ana Γ (trms_list_st S)) S"
  shows "tfr_st S"
by (rule tfr_st_if_comp_tfr_st[OF assms])
```

Lemmata for Checking Ground SMP (GSMP) Disjointness

```
context
begin
private lemma ground_SMP_disjointI_aux1:
  fixes M::"('fun, ('fun, 'atom) term × nat) term set"
  assumes f_def: "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) =
{}}"
    and g_def: "g ≡ λM. {t ∈ M. fv t = {}}"
  shows "f (SMP M) = g (SMP M)"
proof
  have "t ∈ f (SMP M)" when t: "t ∈ SMP M" "fv t = {}" for t
  proof -
    define δ where "δ ≡ Var::('fun, ('fun, 'atom) term × nat) subst"
    have "wt_subst δ" "wf_trms (subst_range δ)" "t = t · δ"
      using subst_apply_term_empty[of t] that(2) wt_subst_Var wf_trm_subst_range_Var
      unfolding δ_def by auto
    thus ?thesis using SMP.Substitution[OF t(1), of δ] t(2) unfolding f_def by fastforce
  qed
  thus "g (SMP M) ⊆ f (SMP M)" unfolding g_def by blast
qed (use f_def g_def in blast)

private lemma ground_SMP_disjointI_aux2:
  fixes M::"('fun, ('fun, 'atom) term × nat) term list"
  assumes f_def: "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) =
{}}"
    and M_SMP_repr: "finite_SMP_representation arity Ana Γ M"
  shows "f (set M) = f (SMP (set M))"
proof
  have M_wf: "wf_trms (set M)"
      and M_var_inst_cl: "is_TComp_var_instance_closed Γ M"
      and M_subterms_cl: "has_all_wt_instances_of Γ (subterms_set (set M)) (set M)"
      and M_Ana_cl: "has_all_wt_instances_of Γ (⋃((set ∘ fst ∘ Ana) ` set M)) (set M)"
    using finite_SMP_representationD[OF M_SMP_repr] by blast+
```

```
  show "f (SMP (set M)) ⊆ f (set M)"
  proof
    fix t assume "t ∈ f (SMP (set M))"
    then obtain s δ where s: "t = s · δ" "s ∈ SMP (set M)" "fv (s · δ) = {}"
        and δ: "wt_subst δ" "wf_trms (subst_range δ)"
      unfolding f_def by blast

    have t_wf: "wf_trm t" using SMP_wf_trm[OF s(2) M_wf] s(1) wf_trm_subst[OF δ(2)] by blast

    obtain m ϑ where m: "m ∈ set M" "s = m · ϑ" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
      using SMP_D''[OF s(2) M_SMP_repr] by blast

    have "t = m · (ϑ ∘s δ)" "fv (m · (ϑ ∘s δ)) = {}" using s(1,3) m(2) by simp_all
    thus "t ∈ f (set M)"
      using m(1) wt_subst_compose[OF ϑ(1) δ(1)] wf_trms_subst_compose[OF ϑ(2) δ(2)]
      unfolding f_def by blast
  qed
qed (auto simp add: f_def)

private lemma ground_SMP_disjointI_aux3:
  fixes A B C::"('fun, ('fun, 'atom) term × nat) term set"
  defines "P ≡ λt s. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ Unifier δ t s"
  assumes f_def: "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) =
{}}"
    and Q_def: "Q ≡ λt. intruder_synth' public arity {} t"
    and R_def: "R ≡ λt. ∃u ∈ C. is_wt_instance_of_cond Γ t u"
    and AB: "wf_trms A" "wf_trms B" "fv_set A ∩ fv_set B = {}"
    and C: "wf_trms C"
    and ABC: "∀t ∈ A. ∀s ∈ B. P t s ⟶ (Q t ∧ Q s) ∨ (R t ∧ R s)"
  shows "f A ∩ f B ⊆ f C ∪ {m. {} ⊢c m}"
proof
  fix t assume "t ∈ f A ∩ f B"
  then obtain ta tb δa δb where
          ta: "t = ta · δa" "ta ∈ A" "wt_subst δa" "wf_trms (subst_range δa)" "fv (ta · δa) = {}"
      and tb: "t = tb · δb" "tb ∈ B" "wt_subst δb" "wf_trms (subst_range δb)" "fv (tb · δb) = {}"
    unfolding f_def by blast

  have ta_tb_wf: "wf_trm ta" "wf_trm tb" "fv ta ∩ fv tb = {}" "Γ ta = Γ tb"
    using ta(1,2) tb(1,2) AB fv_subset_subterms
          wt_subst_trm''[OF ta(3), of ta] wt_subst_trm''[OF tb(3), of tb]
    by (fast, fast, blast, simp)

  obtain ϑ where ϑ: "Unifier ϑ ta tb" "wt_subst ϑ" "wf_trms (subst_range ϑ)"
    using vars_term_disjoint_imp_unifier[OF ta_tb_wf(3), of δa δb]
          ta(1) tb(1) wt_Unifier_if_Unifier[OF ta_tb_wf(1,2,4)]
    by blast
  hence "(Q ta ∧ Q tb) ∨ (R ta ∧ R tb)" using ABC ta(2) tb(2) unfolding P_def by blast+
  thus "t ∈ f C ∪ {m. {} ⊢c m}"
  proof
    show "Q ta ∧ Q tb ⟹ ?thesis"
      using ta(1) pgwt_ground[of ta] pgwt_is_empty_synth[of ta] subst_ground_ident[of ta δa]
      unfolding Q_def f_def intruder_synth_code[symmetric] by simp
  next
    assume "R ta ∧ R tb"
    then obtain ua σa where ua: "ta = ua · σa" "ua ∈ C" "wt_subst σa" "wf_trms (subst_range σa)"
      using ϑ ABC ta_tb_wf(1,2) ta(2) tb(2) C is_wt_instance_of_condD'
      unfolding P_def R_def by metis

    have "t = ua · (σa ∘s δa)" "fv t = {}"
      using ua(1) ta(1,5) tb(1,5) by auto
    thus ?thesis
      using ua(2) wt_subst_compose[OF ua(3) ta(3)] wf_trms_subst_compose[OF ua(4) ta(4)]
      unfolding f_def by blast
```

```
  qed
qed

lemma ground_SMP_disjointI:
  fixes A B::"('fun, ('fun, 'atom) term × nat) term list" and C
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "g ≡ λM. {t ∈ M. fv t = {}}"
    and "Q ≡ λt. intruder_synth' public arity {} t"
    and "R ≡ λt. ∃u ∈ C. is_wt_instance_of_cond Γ t u"
  assumes AB_fv_disj: "fv_set (set A) ∩ fv_set (set B) = {}"
    and A_SMP_repr: "finite_SMP_representation arity Ana Γ A"
    and B_SMP_repr: "finite_SMP_representation arity Ana Γ B"
    and C_wf: "wf_trms C"
    and ABC: "∀t ∈ set A. ∀s ∈ set B. Γ t = Γ s ∧ mgu t s ≠ None ⟶ (Q t ∧ Q s) ∨ (R t ∧ R s)"
  shows "g (SMP (set A)) ∩ g (SMP (set B)) ⊆ f C ∪ {m. {} ⊢c m}"
proof -
  have AB_wf: "wf_trms (set A)" "wf_trms (set B)"
    using A_SMP_repr B_SMP_repr
    unfolding finite_SMP_representation_def wf_trm_code list_all_iff
    by blast+

  let ?P = "λt s. ∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ Unifier δ t s"
  have ABC': "∀t ∈ set A. ∀s ∈ set B. ?P t s ⟶ (Q t ∧ Q s) ∨ (R t ∧ R s)"
    by (metis (no_types) ABC mgu_None_is_subst_neq wt_subst_trm'')

  show ?thesis
    using ground_SMP_disjointI_aux1[OF f_def g_def, of "set A"]
          ground_SMP_disjointI_aux1[OF f_def g_def, of "set B"]
          ground_SMP_disjointI_aux2[OF f_def A_SMP_repr]
          ground_SMP_disjointI_aux2[OF f_def B_SMP_repr]
          ground_SMP_disjointI_aux3[OF f_def Q_def R_def AB_wf AB_fv_disj C_wf ABC']
    by argo
qed

end

end

end
```

3.4 The Typing Result (Typing_Result)

```
theory Typing_Result
imports Typed_Model
begin
```

3.4.1 The Typing Result for the Composition-Only Intruder

```
context typed_model
begin
```

Well-typedness and Type-Flaw Resistance Preservation

```
context
begin

private lemma LI_preserves_tfr_stp_all_single:
  assumes "(S,ϑ) ⇝ (S',ϑ')" "wf_constr S ϑ" "wt_subst ϑ"
  and "list_all tfr_stp S" "tfr_set (trms_st S)" "wf_trms (trms_st S)"
  shows "list_all tfr_stp S'"
using assms
proof (induction rule: LI_rel.induct)
```

```
  case (Compose S X f S' ϑ)
  hence "list_all tfr_stp S" "list_all tfr_stp S'" by simp_all
  moreover have "list_all tfr_stp (map Send X)" by (induct X) auto
  ultimately show ?case by simp
next
  case (Unify S f Y δ X S' ϑ)
  hence "list_all tfr_stp (S@S')" by simp

  have "fv_st (S@Send (Fun f X)#S') ∩ bvars_st (S@S') = {}"
    using Unify.prems(1) by (auto simp add: wf_constr_def)
  moreover have "fv (Fun f X) ⊆ fv_st (S@Send (Fun f X)#S')" by auto
  moreover have "fv (Fun f Y) ⊆ fv_st (S@Send (Fun f X)#S')"
    using Unify.hyps(2) fv_subset_if_in_strand_ik'[of "Fun f Y" S] by force
  ultimately have bvars_disj:
      "bvars_st (S@S') ∩ fv (Fun f X) = {}" "bvars_st (S@S') ∩ fv (Fun f Y) = {}"
    by blast+

  have "wf_trm (Fun f X)" using Unify.prems(5) by simp
  moreover have "wf_trm (Fun f Y)"
  proof -
    obtain x where "x ∈ set S" "Fun f Y ∈ subterms_set (trms_stp x)" "wf_trms (trms_stp x)"
      using Unify.hyps(2) Unify.prems(5) by force+
    thus ?thesis using wf_trm_subterm by auto
  qed
  moreover have
      "Fun f X ∈ SMP (trms_st (S@Send (Fun f X)#S'))" "Fun f Y ∈ SMP (trms_st (S@Send (Fun f X)#S'))"
    using SMP_append[of S "Send (Fun f X)#S'"] SMP_Cons[of "Send (Fun f X)" S']
          SMP_ikI[OF Unify.hyps(2)]
    by auto
  hence "Γ (Fun f X) = Γ (Fun f Y)"
    using Unify.prems(4) mgu_gives_MGU[OF Unify.hyps(3)[symmetric]]
    unfolding tfr_set_def by blast
  ultimately have "wt_subst δ" using mgu_wt_if_same_type[OF Unify.hyps(3)[symmetric]] by metis
  moreover have "wf_trms (subst_range δ)"
    using mgu_wf_trm[OF Unify.hyps(3)[symmetric] ‹wf_trm (Fun f X)› ‹wf_trm (Fun f Y)›]
    by (metis wf_trm_subst_range_iff)
  moreover have "bvars_st (S@S') ∩ range_vars δ = {}"
    using mgu_vars_bounded[OF Unify.hyps(3)[symmetric]] bvars_disj by fast
  ultimately show ?case using tfr_stp_all_wt_subst_apply[OF ‹list_all tfr_stp (S@S')›] by metis
next
  case (Equality S δ t t' a S' ϑ)
  have "list_all tfr_stp (S@S')" "Γ t = Γ t'"
    using tfr_stp_all_same_type[of S a t t' S']
          tfr_stp_all_split(5)[of S _ S']
          MGU_is_Unifier[OF mgu_gives_MGU[OF Equality.hyps(2)[symmetric]]]
          Equality.prems(3)
    by blast+
  moreover have "wf_trm t" "wf_trm t'" using Equality.prems(5) by auto
  ultimately have "wt_subst δ"
    using mgu_wt_if_same_type[OF Equality.hyps(2)[symmetric]]
    by metis
  moreover have "wf_trms (subst_range δ)"
    using mgu_wf_trm[OF Equality.hyps(2)[symmetric] ‹wf_trm t› ‹wf_trm t'›]
    by (metis wf_trm_subst_range_iff)
  moreover have "fv_st (S@Equality a t t'#S') ∩ bvars_st (S@Equality a t t'#S') = {}"
    using Equality.prems(1) by (auto simp add: wf_constr_def)
  hence "bvars_st (S@S') ∩ fv t = {}" "bvars_st (S@S') ∩ fv t' = {}" by auto
  hence "bvars_st (S@S') ∩ range_vars δ = {}"
    using mgu_vars_bounded[OF Equality.hyps(2)[symmetric]] by fast
  ultimately show ?case using tfr_stp_all_wt_subst_apply[OF ‹list_all tfr_stp (S@S')›] by metis
qed

private lemma LI_in_SMP_subset_single:
```

```
  assumes "(S,ϑ) ⇝ (S',ϑ')" "wf_constr S ϑ" "wt_subst ϑ"
          "tfr_set (trms_st S)" "wf_trms (trms_st S)" "list_all tfr_stp S"
  and "trms_st S ⊆ SMP M"
  shows "trms_st S' ⊆ SMP M"
using assms
proof (induction rule: LI_rel.induct)
  case (Compose S X f S' ϑ)
  hence "SMP (trms_st [Send (Fun f X)]) ⊆ SMP M"
  proof -
    have "SMP (trms_st [Send (Fun f X)]) ⊆ SMP (trms_st (S@Send (Fun f X)#S'))"
      using trms_st_append SMP_mono by auto
    thus ?thesis
      using SMP_union[of "trms_st (S@Send (Fun f X)#S')" M]
            SMP_subset_union_eq[OF Compose.prems(6)]
      by auto
  qed
  thus ?case using Compose.prems(6) by auto
next
  case (Unify S f Y δ X S' ϑ)
  have "Fun f X ∈ SMP (trms_st (S@Send (Fun f X)#S'))" by auto
  moreover have "MGU δ (Fun f X) (Fun f Y)"
    by (metis mgu_gives_MGU[OF Unify.hyps(3)[symmetric]])
  moreover have
       "⋀x. x ∈ set S ⟹ wf_trms (trms_stp x)" "wf_trm (Fun f X)"
    using Unify.prems(4) by force+
  moreover have "Fun f Y ∈ SMP (trms_st (S@Send (Fun f X)#S'))"
    by (meson SMP_ikI Unify.hyps(2) contra_subsetD ik_append_subset(1))
  ultimately have "wf_trm (Fun f Y)" "Γ (Fun f X) = Γ (Fun f Y)"
    using ik_st_subterm_exD[OF ‹Fun f Y ∈ ik_st S›] ‹tfr_set (trms_st (S@Send (Fun f X)#S'))›
    unfolding tfr_set_def by (metis (full_types) SMP_wf_trm Unify.prems(4), blast)
  hence "wt_subst δ" by (metis mgu_wt_if_same_type[OF Unify.hyps(3)[symmetric] ‹wf_trm (Fun f X)›])
  moreover have "wf_trms (subst_range δ)"
    using mgu_wf_trm[OF Unify.hyps(3)[symmetric] ‹wf_trm (Fun f X)› ‹wf_trm (Fun f Y)›] by simp
  ultimately have "trms_st ((S@Send (Fun f X)#S') ·st δ) ⊆ SMP M"
    using SMP.Substitution Unify.prems(6) wt_subst_SMP_subset by metis
  thus ?case by auto
next
  case (Equality S δ t t' a S' ϑ)
  hence "Γ t = Γ t'"
    using tfr_stp_all_same_type MGU_is_Unifier[OF mgu_gives_MGU[OF Equality.hyps(2)[symmetric]]]
    by metis
  moreover have "t ∈ SMP (trms_st (S@Equality a t t'#S'))" "t' ∈ SMP (trms_st (S@Equality a t t'#S'))"
    using Equality.prems(1) by auto
  moreover have "MGU δ t t'" using mgu_gives_MGU[OF Equality.hyps(2)[symmetric]] by metis
  moreover have "⋀x. x ∈ set S ⟹ wf_trms (trms_stp x)" "wf_trm t" "wf_trm t'"
    using Equality.prems(4) by force+
  ultimately have "wt_subst δ" by (metis mgu_wt_if_same_type[OF Equality.hyps(2)[symmetric] ‹wf_trm t›])
  moreover have "wf_trms (subst_range δ)"
    using mgu_wf_trm[OF Equality.hyps(2)[symmetric] ‹wf_trm t› ‹wf_trm t'›] by simp
  ultimately have "trms_st ((S@Equality a t t'#S') ·st δ) ⊆ SMP M"
    using SMP.Substitution Equality.prems wt_subst_SMP_subset by metis
  thus ?case by auto
qed

private lemma LI_preserves_tfr_single:
  assumes "(S,ϑ) ⇝ (S',ϑ')" "wf_constr S ϑ" "wt_subst ϑ" "wf_trms (subst_range ϑ)"
          "tfr_set (trms_st S)" "wf_trms (trms_st S)"
          "list_all tfr_stp S"
  shows "tfr_set (trms_st S') ∧ wf_trms (trms_st S')"
using assms
proof (induction rule: LI_rel.induct)
  case (Compose S X f S' ϑ)
  let ?SMPmap = "SMP (trms_st (S@map Send X@S')) - (Var`𝒱)"
```

```
  have "?SMPmap ⊆ SMP (trms_st (S@Send (Fun f X)#S')) - (Var`𝒱)"
    using SMP_fun_map_snd_subset[of X f]
          SMP_append[of "map Send X" S'] SMP_Cons[of "Send (Fun f X)" S']
          SMP_append[of S "Send (Fun f X)#S'"] SMP_append[of S "map Send X@S'"]
    by auto
  hence "∀s ∈ ?SMPmap. ∀t ∈ ?SMPmap. (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
    using Compose unfolding tfr_set_def by (meson subsetCE)
  thus ?case
    using LI_preserves_trm_wf[OF r_into_rtrancl[OF LI_rel.Compose[OF Compose.hyps]], of S']
          Compose.prems(5)
    unfolding tfr_set_def by blast
next
  case (Unify S f Y δ X S' ϑ)
  let ?SMPδ = "SMP (trms_st (S@S' ·_st δ)) - (Var`𝒱)"

  have "SMP (trms_st (S@S' ·_st δ)) ⊆ SMP (trms_st (S@Send (Fun f X)#S'))"
  proof
    fix s assume "s ∈ SMP (trms_st (S@S' ·_st δ))" thus "s ∈ SMP (trms_st (S@Send (Fun f X)#S'))"
      using LI_in_SMP_subset_single[
              OF LI_rel.Unify[OF Unify.hyps] Unify.prems(1,2,4,5,6)
                 MP_subset_SMP(2)[of "S@Send (Fun f X)#S'"]]
      by (metis SMP_union SMP_subset_union_eq Un_iff)
  qed
  hence "∀s ∈ ?SMPδ. ∀t ∈ ?SMPδ. (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
    using Unify.prems(4) unfolding tfr_set_def by (meson Diff_iff subsetCE)
  thus ?case
    using LI_preserves_trm_wf[OF r_into_rtrancl[OF LI_rel.Unify[OF Unify.hyps]], of S']
          Unify.prems(5)
    unfolding tfr_set_def by blast
next
  case (Equality S δ t t' a S' ϑ)
  let ?SMPδ = "SMP (trms_st (S@S' ·_st δ)) - (Var`𝒱)"

  have "SMP (trms_st (S@S' ·_st δ)) ⊆ SMP (trms_st (S@Equality a t t'#S'))"
  proof
    fix s assume "s ∈ SMP (trms_st (S@S' ·_st δ))" thus "s ∈ SMP (trms_st (S@Equality a t t'#S'))"
      using LI_in_SMP_subset_single[
              OF LI_rel.Equality[OF Equality.hyps] Equality.prems(1,2,4,5,6)
                 MP_subset_SMP(2)[of "S@Equality a t t'#S'"]]
      by (metis SMP_union SMP_subset_union_eq Un_iff)
  qed
  hence "∀s ∈ ?SMPδ. ∀t ∈ ?SMPδ. (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
    using Equality.prems unfolding tfr_set_def by (meson Diff_iff subsetCE)
  thus ?case
    using LI_preserves_trm_wf[OF r_into_rtrancl[OF LI_rel.Equality[OF Equality.hyps]], of _ S']
          Equality.prems
    unfolding tfr_set_def by blast
qed

private lemma LI_preserves_welltypedness_single:
  assumes "(S,ϑ) ⇝ (S',ϑ')" "wf_constr S ϑ" "wt_subst ϑ" "wf_trms (subst_range ϑ)"
  and "tfr_set (trms_st S)" "wf_trms (trms_st S)" "list_all tfr_stp S"
  shows "wt_subst ϑ' ∧ wf_trms (subst_range ϑ')"
using assms
proof (induction rule: LI_rel.induct)
  case (Unify S f Y δ X S' ϑ)
  have "wf_trm (Fun f X)" using Unify.prems(5) unfolding tfr_set_def by simp
  moreover have "wf_trm (Fun f Y)"
  proof -
    obtain x where "x ∈ set S" "Fun f Y ∈ subterms_set (trms_stp x)" "wf_trms (trms_stp x)"
      using Unify.hyps(2) Unify.prems(5) unfolding tfr_set_def by force
    thus ?thesis using wf_trm_subterm by auto
  qed
```

```
  moreover have
      "Fun f X ∈ SMP (trms_st (S@Send (Fun f X)#S'))" "Fun f Y ∈ SMP (trms_st (S@Send (Fun f X)#S'))"
    using SMP_append[of S "Send (Fun f X)#S'"] SMP_Cons[of "Send (Fun f X)" S']
          SMP_ikI[OF Unify.hyps(2)]
    by auto
  hence "Γ (Fun f X) = Γ (Fun f Y)"
    using Unify.prems(4) mgu_gives_MGU[OF Unify.hyps(3)[symmetric]]
    unfolding tfr_set_def by blast
  ultimately have "wt_subst δ" using mgu_wt_if_same_type[OF Unify.hyps(3)[symmetric]] by metis

  have "wf_trms (subst_range δ)"
    by (meson mgu_wf_trm[OF Unify.hyps(3)[symmetric] ‹wf_trm (Fun f X)› ‹wf_trm (Fun f Y)›]
              wf_trm_subst_range_iff)
  hence "wf_trms (subst_range (ϑ ∘s δ))"
    using wf_trm_subst_range_iff wf_trm_subst ‹wf_trms (subst_range ϑ)›
    unfolding subst_compose_def
    by (metis (no_types, lifting))
  thus ?case by (metis wt_subst_compose[OF ‹wt_subst ϑ› ‹wt_subst δ›])
next
  case (Equality S δ t t' a S' ϑ)
  have "wf_trm t" "wf_trm t'" using Equality.prems(5) by simp_all
  moreover have "Γ t = Γ t'"
    using ‹list_all tfr_stp (S@Equality a t t'#S')›
          MGU_is_Unifier[OF mgu_gives_MGU[OF Equality.hyps(2)[symmetric]]]
    by auto
  ultimately have "wt_subst δ" using mgu_wt_if_same_type[OF Equality.hyps(2)[symmetric]] by metis

  have "wf_trms (subst_range δ)"
    by (meson mgu_wf_trm[OF Equality.hyps(2)[symmetric] ‹wf_trm t› ‹wf_trm t'›] wf_trm_subst_range_iff)
  hence "wf_trms (subst_range (ϑ ∘s δ))"
    using wf_trm_subst_range_iff wf_trm_subst ‹wf_trms (subst_range ϑ)›
    unfolding subst_compose_def
    by (metis (no_types, lifting))
  thus ?case by (metis wt_subst_compose[OF ‹wt_subst ϑ› ‹wt_subst δ›])
qed metis

lemma LI_preserves_welltypedness:
  assumes "(S,ϑ) ⇝* (S',ϑ')" "wf_constr S ϑ" "wt_subst ϑ" "wf_trms (subst_range ϑ)"
    and "tfr_set (trms_st S)" "wf_trms (trms_st S)" "list_all tfr_stp S"
  shows "wt_subst ϑ'" (is "?A ϑ'")
    and "wf_trms (subst_range ϑ')" (is "?B ϑ'")
proof -
  have "?A ϑ' ∧ ?B ϑ'" using assms
  proof (induction S ϑ rule: converse_rtrancl_induct2)
    case (step S1 ϑ1 S2 ϑ2)
    hence "?A ϑ2 ∧ ?B ϑ2" using LI_preserves_welltypedness_single by presburger
    moreover have "wf_constr S2 ϑ2"
      by (fact LI_preserves_wellformedness[OF r_into_rtrancl[OF step.hyps(1)] step.prems(1)])
    moreover have "tfr_set (trms_st S2)" "wf_trms (trms_st S2)"
      using LI_preserves_tfr_single[OF step.hyps(1)] step.prems by presburger+
    moreover have "list_all tfr_stp S2"
      using LI_preserves_tfr_stp_all_single[OF step.hyps(1)] step.prems by fastforce
    ultimately show ?case using step.IH by presburger
  qed simp
  thus "?A ϑ'" "?B ϑ'" by simp_all
qed

lemma LI_preserves_tfr:
  assumes "(S,ϑ) ⇝* (S',ϑ')" "wf_constr S ϑ" "wt_subst ϑ" "wf_trms (subst_range ϑ)"
    and "tfr_set (trms_st S)" "wf_trms (trms_st S)" "list_all tfr_stp S"
  shows "tfr_set (trms_st S')" (is "?A S'")
    and "wf_trms (trms_st S')" (is "?B S'")
    and "list_all tfr_stp S'" (is "?C S'")
```

```
proof -
  have "?A S' ∧ ?B S' ∧ ?C S'" using assms
  proof (induction S ϑ rule: converse_rtrancl_induct2)
    case (step S1 ϑ1 S2 ϑ2)
    have "wf_constr S2 ϑ2" "tfr_set (trms_st S2)" "wf_trms (trms_st S2)" "list_all tfr_stp S2"
      using LI_preserves_wellformedness[OF r_into_rtrancl[OF step.hyps(1)] step.prems(1)]
            LI_preserves_tfr_single[OF step.hyps(1) step.prems(1,2)]
            LI_preserves_tfr_stp_all_single[OF step.hyps(1) step.prems(1,2)]
            step.prems(3,4,5,6)
      by metis+
    moreover have "wt_subst ϑ2" "wf_trms (subst_range ϑ2)"
      using LI_preserves_welltypedness[OF r_into_rtrancl[OF step.hyps(1)] step.prems]
      by simp_all
    ultimately show ?case using step.IH by presburger
  qed blast
  thus "?A S'" "?B S'" "?C S'" by simp_all
qed
end
```

Simple Constraints are Well-typed Satisfiable

Proving the existence of a well-typed interpretation

```
context
begin
lemma wt_interpretation_exists:
  obtains ℐ::"('fun,'var) subst"
  where "interpretation_subst ℐ" "wt_subst ℐ" "subst_range ℐ ⊆ public_ground_wf_terms"
proof
  define ℐ where "ℐ = (λx. (SOME t. Γ (Var x) = Γ t ∧ public_ground_wf_term t))"

  { fix x t assume "ℐ x = t"
    hence "Γ (Var x) = Γ t ∧ public_ground_wf_term t"
      using someI_ex[of "λt. Γ (Var x) = Γ t ∧ public_ground_wf_term t",
                     OF type_pgwt_inhabited[of "Var x"]]
      unfolding ℐ_def wf_trm_def by simp
  } hence props: "ℐ v = t ⟹ Γ (Var v) = Γ t ∧ public_ground_wf_term t" for v t by metis

  have "ℐ v ≠ Var v" for v using props pgwt_ground by force
  hence "subst_domain ℐ = UNIV" by auto
  moreover have "ground (subst_range ℐ)" by (simp add: props pgwt_ground)
  ultimately show "interpretation_subst ℐ" by metis
  show "wt_subst ℐ" unfolding wt_subst_def using props by simp
  show "subst_range ℐ ⊆ public_ground_wf_terms" by (auto simp add: props)
qed

lemma wt_grounding_subst_exists:
  "∃ϑ. wt_subst ϑ ∧ wf_trms (subst_range ϑ) ∧ fv (t · ϑ) = {}"
proof -
  obtain ϑ where ϑ: "interpretation_subst ϑ" "wt_subst ϑ" "subst_range ϑ ⊆ public_ground_wf_terms"
    using wt_interpretation_exists by blast
  show ?thesis using pgwt_wellformed interpretation_grounds[OF ϑ(1)] ϑ(2,3) by blast
qed

private fun fresh_pgwt::"'fun set ⇒ ('fun,'atom) term_type ⇒ ('fun,'var) term" where
  "fresh_pgwt S (TAtom a) =
    Fun (SOME c. c ∉ S ∧ Γ (Fun c []) = TAtom a ∧ public c) []"
| "fresh_pgwt S (TComp f T) = Fun f (map (fresh_pgwt S) T)"

private lemma fresh_pgwt_same_type:
  assumes "finite S" "wf_trm t"
  shows "Γ (fresh_pgwt S (Γ t)) = Γ t"
proof -
  let ?P = "λτ::('fun,'atom) term_type. wf_trm τ ∧ (∀f T. TComp f T ⊑ τ ⟶ 0 < arity f)"
```

```
  { fix τ assume "?P τ" hence "Γ (fresh_pgwt S τ) = τ"
    proof (induction τ)
      case (Var a)
      let ?P = "λc. c ∉ S ∧ Γ (Fun c []) = Var a ∧ public c"
      let ?Q = "λc. Γ (Fun c []) = Var a ∧ public c"
      have " {c. ?Q c} - S = {c. ?P c}" by auto
      hence "infinite {c. ?P c}"
        using Diff_infinite_finite[OF assms(1) infinite_typed_consts[of a]]
        by metis
      hence "∃c. ?P c" using not_finite_existsD by blast
      thus ?case using someI_ex[of ?P] by auto
    next
      case (Fun f T)
      have f: "0 < arity f" using Fun.prems fun_type_inv by auto
      have "⋀t. t ∈ set T ⟹ ?P t"
        using Fun.prems wf_trm_subtermeq term.le_less_trans Fun_param_is_subterm
        by metis
      hence "⋀t. t ∈ set T ⟹ Γ (fresh_pgwt S t) = t" using Fun.prems Fun.IH by auto
      hence "map Γ (map (fresh_pgwt S) T) = T"  by (induct T) auto
      thus ?case using fun_type[OF f] by simp
    qed
  } thus ?thesis using assms(1) Γ_wf'[OF assms(2)] Γ_wf(1) by auto
qed

private lemma fresh_pgwt_empty_synth:
  assumes "finite S" "wf_trm t"
  shows "{} ⊢c fresh_pgwt S (Γ t)"
proof -
  let ?P = "λτ::('fun,'atom) term_type. wf_trm τ ∧ (∀f T. TComp f T ⊑ τ ⟶ 0 < arity f)"
  { fix τ assume "?P τ" hence "{} ⊢c fresh_pgwt S τ"
    proof (induction τ)
      case (Var a)
      let ?P = "λc. c ∉ S ∧ Γ (Fun c []) = Var a ∧ public c"
      let ?Q = "λc. Γ (Fun c []) = Var a ∧ public c"
      have " {c. ?Q c} - S = {c. ?P c}" by auto
      hence "infinite {c. ?P c}"
        using Diff_infinite_finite[OF assms(1) infinite_typed_consts[of a]]
        by metis
      hence "∃c. ?P c" using not_finite_existsD by blast
      thus ?case
        using someI_ex[of ?P] intruder_synth.ComposeC[of "[]" _ "{}"] const_type_inv
        by auto
    next
      case (Fun f T)
      have f: "0 < arity f" "length T = arity f" "public f"
        using Fun.prems fun_type_inv unfolding wf_trm_def by auto
      have "⋀t. t ∈ set T ⟹ ?P t"
        using Fun.prems wf_trm_subtermeq term.le_less_trans Fun_param_is_subterm
        by metis
      hence "⋀t. t ∈ set T ⟹ {} ⊢c fresh_pgwt S t" using Fun.prems Fun.IH by auto
      moreover have "length (map (fresh_pgwt S) T) = arity f" using f(2) by auto
      ultimately show ?case using intruder_synth.ComposeC[of "map (fresh_pgwt S) T" f] f by auto
    qed
  } thus ?thesis using assms(1) Γ_wf'[OF assms(2)] Γ_wf(1) by auto
qed

private lemma fresh_pgwt_has_fresh_const:
  assumes "finite S" "wf_trm t"
  obtains c where "Fun c [] ⊑ fresh_pgwt S (Γ t)" "c ∉ S"
proof -
  let ?P = "λτ::('fun,'atom) term_type. wf_trm τ ∧ (∀f T. TComp f T ⊑ τ ⟶ 0 < arity f)"
  { fix τ assume "?P τ" hence "∃c. Fun c [] ⊑ fresh_pgwt S τ ∧ c ∉ S"
    proof (induction τ)
```

```
      case (Var a)
      let ?P = "λc. c ∉ S ∧ Γ (Fun c []) = Var a ∧ public c"
      let ?Q = "λc. Γ (Fun c []) = Var a ∧ public c"
      have " {c. ?Q c} - S = {c. ?P c}" by auto
      hence "infinite {c. ?P c}"
        using Diff_infinite_finite[OF assms(1) infinite_typed_consts[of a]]
        by metis
      hence "∃c. ?P c" using not_finite_existsD by blast
      thus ?case using someI_ex[of ?P] by auto
    next
      case (Fun f T)
      have f: "0 < arity f" "length T = arity f" "public f" "T ≠ []"
        using Fun.prems fun_type_inv unfolding wf_trm_def by auto
      obtain t' where t': "t' ∈ set T" by (meson all_not_in_conv f(4) set_empty)
      have "⋀t. t ∈ set T ⟹ ?P t"
        using Fun.prems wf_trm_subtermeq term.le_less_trans Fun_param_is_subterm
        by metis
      hence "⋀t. t ∈ set T ⟹ ∃c. Fun c [] ⊑ fresh_pgwt S t ∧ c ∉ S"
        using Fun.prems Fun.IH by auto
      then obtain c where c: "Fun c [] ⊑ fresh_pgwt S t'" "c ∉ S" using t' by metis
      thus ?case using t' by auto
    qed
  } thus ?thesis using that assms Γ_wf'[OF assms(2)] Γ_wf(1) by blast
qed

private lemma fresh_pgwt_subterm_fresh:
  assumes "finite S" "wf_trm t" "wf_trm s" "funs_term s ⊆ S"
  shows "s ∉ subterms (fresh_pgwt S (Γ t))"
proof -
  let ?P = "λτ::('fun,'atom) term_type. wf_trm τ ∧ (∀f T. TComp f T ⊑ τ ⟶ 0 < arity f)"
  { fix τ assume "?P τ" hence "s ∉ subterms (fresh_pgwt S τ)"
    proof (induction τ)
      case (Var a)
      let ?P = "λc. c ∉ S ∧ Γ (Fun c []) = Var a ∧ public c"
      let ?Q = "λc. Γ (Fun c []) = Var a ∧ public c"
      have " {c. ?Q c} - S = {c. ?P c}" by auto
      hence "infinite {c. ?P c}"
        using Diff_infinite_finite[OF assms(1) infinite_typed_consts[of a]]
        by metis
      hence "∃c. ?P c" using not_finite_existsD by blast
      thus ?case using someI_ex[of ?P] assms(4) by auto
    next
      case (Fun f T)
      have f: "0 < arity f" "length T = arity f" "public f"
        using Fun.prems fun_type_inv unfolding wf_trm_def by auto
      have "⋀t. t ∈ set T ⟹ ?P t"
        using Fun.prems wf_trm_subtermeq term.le_less_trans Fun_param_is_subterm
        by metis
      hence "⋀t. t ∈ set T ⟹ s ∉ subterms (fresh_pgwt S t)" using Fun.prems Fun.IH by auto
      moreover have "s ≠ fresh_pgwt S (Fun f T)"
      proof -
        obtain c where c: "Fun c [] ⊑ fresh_pgwt S (Fun f T)" "c ∉ S"
          using fresh_pgwt_has_fresh_const[OF assms(1)] type_wfttype_inhabited Fun.prems
          by metis
        hence "¬Fun c [] ⊑ s" using assms(4) subtermeq_imp_funs_term_subset by force
        thus ?thesis using c(1) by auto
      qed
      ultimately show ?case by auto
    qed
  } thus ?thesis using assms(1) Γ_wf'[OF assms(2)] Γ_wf(1) by auto
qed

private lemma wt_fresh_pgwt_term_exists:
```

```
  assumes "finite T" "wf_trm s" "wf_trms T"
  obtains t where "Γ t = Γ s" "{} ⊢c t" "∀s ∈ T. ∀u ∈ subterms s. u ∉ subterms t"
proof -
  have finite_S: "finite (⋃(funs_term ` T))" using assms(1) by auto

  have 1: "Γ (fresh_pgwt (⋃(funs_term ` T)) (Γ s)) = Γ s"
    using fresh_pgwt_same_type[OF finite_S assms(2)] by auto

  have 2: "{} ⊢c fresh_pgwt (⋃(funs_term ` T)) (Γ s)"
    using fresh_pgwt_empty_synth[OF finite_S assms(2)] by auto

  have 3: "∀v ∈ T. ∀u ∈ subterms v. u ∉ subterms (fresh_pgwt (⋃(funs_term ` T)) (Γ s))"
    using fresh_pgwt_subterm_fresh[OF finite_S assms(2)] assms(3)
          wf_trm_subtermeq subtermeq_imp_funs_term_subset
    by force

  show ?thesis by (rule that[OF 1 2 3])
qed

lemma wt_bij_finite_subst_exists:
  assumes "finite (S::'var set)" "finite (T::('fun,'var) terms)" "wf_trms T"
  shows "∃σ::('fun,'var) subst.
              subst_domain σ = S
            ∧ bij_betw σ (subst_domain σ) (subst_range σ)
            ∧ subterms_set (subst_range σ) ⊆ {t. {} ⊢c t} - T
            ∧ (∀s ∈ subst_range σ. ∀u ∈ subst_range σ. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u)
            ∧ wt_subst σ
            ∧ wf_trms (subst_range σ)"
using assms
proof (induction rule: finite_induct)
  case empty
  have "subst_domain Var = {}"
       "bij_betw Var (subst_domain Var) (subst_range Var)"
       "subterms_set (subst_range Var) ⊆ {t. {} ⊢c t} - T"
       "∀s ∈ subst_range Var. ∀u ∈ subst_range Var. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u"
       "wt_subst Var"
       "wf_trms (subst_range Var)"
    unfolding bij_betw_def
    by auto
  thus ?case by (force simp add: subst_domain_def)
next
  case (insert x S)
  then obtain σ where σ:
      "subst_domain σ = S" "bij_betw σ (subst_domain σ) (subst_range σ)"
      "subterms_set (subst_range σ) ⊆ {t. {} ⊢c t} - T"
      "∀s ∈ subst_range σ. ∀u ∈ subst_range σ. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u"
      "wt_subst σ" "wf_trms (subst_range σ)"
    by (auto simp del: subst_range.simps)

  have *: "finite (T ∪ subst_range σ)"
    using insert.prems(1) insert.hyps(1) σ(1) by simp
  have **: "wf_trm (Var x)" by simp
  have ***: "wf_trms (T ∪ subst_range σ)" using assms(3) σ(6) by blast
  obtain t where t:
      "Γ t = Γ (Var x)" "{} ⊢c t"
      "∀s ∈ T ∪ subst_range σ. ∀u ∈ subterms s. u ∉ subterms t"
    using wt_fresh_pgwt_term_exists[OF * ** ***] by auto

  obtain ϑ where ϑ: "ϑ ≡ λy. if x = y then t else σ y" by simp

  have t_ground: "fv t = {}" using t(2) pgwt_ground[of t] pgwt_is_empty_synth[of t] by auto
  hence x_dom: "x ∉ subst_domain σ" "x ∈ subst_domain ϑ" using insert.hyps(2) σ(1) ϑ by auto
  moreover have "subst_range σ ⊆ subterms_set (subst_range σ)" by auto
```

```
hence ground_imgs: "ground (subst_range σ)"
  using σ(3) pgwt_ground pgwt_is_empty_synth
  by force
ultimately have x_img: "σ x ∉ subst_range σ"
  using ground_subst_dom_iff_img
  by (auto simp add: subst_domain_def)

have "ground (insert t (subst_range σ))"
  using ground_imgs x_dom t_ground
  by auto
have ϑ_dom: "subst_domain ϑ = insert x (subst_domain σ)"
  using ϑ t_ground by (auto simp add: subst_domain_def)
have ϑ_img: "subst_range ϑ = insert t (subst_range σ)"
proof
  show "subst_range ϑ ⊆ insert t (subst_range σ)"
  proof
    fix t' assume "t' ∈ subst_range ϑ"
    then obtain y where "y ∈ subst_domain ϑ" "t' = ϑ y" by auto
    thus "t' ∈ insert t (subst_range σ)" using ϑ by (auto simp add: subst_domain_def)
  qed
  show "insert t (subst_range σ) ⊆ subst_range ϑ"
  proof
    fix t' assume t': "t' ∈ insert t (subst_range σ)"
    hence "fv t' = {}" using ground_imgs x_img t_ground by auto
    hence "t' ≠ Var x" by auto
    show "t' ∈ subst_range ϑ"
    proof (cases "t' = t")
      case False
      hence "t' ∈ subst_range σ" using t' by auto
      then obtain y where "σ y ∈ subst_range σ" "t' = σ y" by auto
      hence "y ∈ subst_domain σ" "t' ≠ Var y"
        using ground_subst_dom_iff_img[OF ground_imgs(1)]
        by (auto simp add: subst_domain_def simp del: subst_range.simps)
      hence "x ≠ y" using x_dom by auto
      hence "ϑ y = σ y" unfolding ϑ by auto
      thus ?thesis using ‹t' ≠ Var y› ‹t' = σ y› subst_imgI[of ϑ y] by auto
    qed (metis subst_imgI ϑ ‹t' ≠ Var x›)
  qed
qed
hence ϑ_ground_img: "ground (subst_range ϑ)"
  using ground_imgs t_ground
  by auto

have "subst_domain ϑ = insert x S" using ϑ_dom σ(1) by auto
moreover have "bij_betw ϑ (subst_domain ϑ) (subst_range ϑ)"
proof (intro bij_betwI')
  fix y z assume *: "y ∈ subst_domain ϑ" "z ∈ subst_domain ϑ"
  hence "fv (ϑ y) = {}" "fv (ϑ z) = {}" using ϑ_ground_img by auto
  { assume "ϑ y = ϑ z" hence "y = z"
    proof (cases "ϑ y ∈ subst_range σ ∧ ϑ z ∈ subst_range σ")
      case True
      hence **: "y ∈ subst_domain σ" "z ∈ subst_domain σ"
        using ϑ ϑ_dom True * t(3) by (metis Un_iff term.order_refl insertE)+
      hence "y ≠ x" "z ≠ x" using x_dom by auto
      hence "ϑ y = σ y" "ϑ z = σ z" using ϑ by auto
      thus ?thesis using ‹ϑ y = ϑ z› σ(2) ** unfolding bij_betw_def inj_on_def by auto
    qed (metis ϑ * ‹ϑ y = ϑ z› ϑ_dom ground_imgs(1) ground_subst_dom_iff_img insertE)
  }
  thus "(ϑ y = ϑ z) = (y = z)" by auto
next
  fix y assume "y ∈ subst_domain ϑ" thus "ϑ y ∈ subst_range ϑ" by auto
next
  fix t assume "t ∈ subst_range ϑ" thus "∃z ∈ subst_domain ϑ. t = ϑ z" by auto
```

```
  qed
  moreover have "subterms_set (subst_range ϑ) ⊆ {t. {} ⊢c t} - T"
  proof -
    { fix s assume "s ⊑ t"
      hence "s ∈ {t. {} ⊢c t} - T"
        using t(2,3)
        by (metis Diff_eq_empty_iff Diff_iff Un_upper1 term.order_refl
                  deduct_synth_subterm mem_Collect_eq)
    } thus ?thesis using σ(3) ϑ ϑ_img by auto
  qed
  moreover have "wt_subst ϑ" using ϑ t(1) σ(5) unfolding wt_subst_def by auto
  moreover have "wf_trms (subst_range ϑ)"
    using ϑ σ(6) t(2) pgwt_is_empty_synth pgwt_wellformed
          wf_trm_subst_range_iff[of σ] wf_trm_subst_range_iff[of ϑ]
    by metis
  moreover have "∀s∈subst_range ϑ. ∀u∈subst_range ϑ. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u"
    using σ(4) ϑ_img t(3) by (auto simp del: subst_range.simps)
  ultimately show ?case by blast
qed

private lemma wt_bij_finite_tatom_subst_exists_single:
  assumes "finite (S::'var set)" "finite (T::('fun,'var) terms)"
  and "⋀x. x ∈ S ⟹ Γ (Var x) = TAtom a"
  shows "∃σ::('fun,'var) subst. subst_domain σ = S
                     ∧ bij_betw σ (subst_domain σ) (subst_range σ)
                     ∧ subst_range σ ⊆ ((λc. Fun c []) ` {c. Γ (Fun c []) = TAtom a ∧
                                                          public c ∧ arity c = 0}) - T
                     ∧ wt_subst σ
                     ∧ wf_trms (subst_range σ)"
proof -
  let ?U = "{c. Γ (Fun c []) = TAtom a ∧ public c ∧ arity c = 0}"

  obtain σ where σ:
      "subst_domain σ = S" "bij_betw σ (subst_domain σ) (subst_range σ)"
      "subst_range σ ⊆ ((λc. Fun c []) ` ?U) - T"
    using bij_finite_const_subst_exists'[OF assms(1,2) infinite_typed_consts'[of a]]
    by auto

  { fix x assume "x ∉ subst_domain σ" hence "Γ (Var x) = Γ (σ x)" by auto }
  moreover
  { fix x assume "x ∈ subst_domain σ"
    hence "∃c ∈ ?U. σ x = Fun c [] ∧ arity c = 0" using σ by auto
    hence "Γ (σ x) = TAtom a" "wf_trm (σ x)" using assms(3) const_type wf_trmI[of "[]"] by auto
    hence "Γ (Var x) = Γ (σ x)" "wf_trm (σ x)" using assms(3) σ(1) by force+
  }
  ultimately have "wt_subst σ" "wf_trms (subst_range σ)"
    using wf_trm_subst_range_iff[of σ]
    unfolding wt_subst_def
    by force+
  thus ?thesis using σ by auto
qed

lemma wt_bij_finite_tatom_subst_exists:
  assumes "finite (S::'var set)" "finite (T::('fun,'var) terms)"
  and "⋀x. x ∈ S ⟹ ∃a. Γ (Var x) = TAtom a"
  shows "∃σ::('fun,'var) subst. subst_domain σ = S
                     ∧ bij_betw σ (subst_domain σ) (subst_range σ)
                     ∧ subst_range σ ⊆ ((λc. Fun c []) ` C_pub) - T
                     ∧ wt_subst σ
                     ∧ wf_trms (subst_range σ)"
using assms
proof (induction rule: finite_induct)
  case empty
```

```
  have "subst_domain Var = {}"
       "bij_betw Var (subst_domain Var) (subst_range Var)"
       "subst_range Var ⊆ ((λc. Fun c []) ` 𝒞pub) - T"
       "wtsubst Var"
       "wftrms (subst_range Var)"
    unfolding bij_betw_def
    by auto
  thus ?case by (auto simp add: subst_domain_def)
next
  case (insert x S)
  then obtain a where a: "Γ (Var x) = TAtom a" by fastforce

  from insert obtain σ where σ:
      "subst_domain σ = S" "bij_betw σ (subst_domain σ) (subst_range σ)"
      "subst_range σ ⊆ ((λc. Fun c []) ` 𝒞pub) - T" "wtsubst σ"
      "wftrms (subst_range σ)"
    by auto

  let ?S' = "{y ∈ S. Γ (Var y) = TAtom a}"
  let ?T' = "T ∪ subst_range σ"

  have *: "finite (insert x ?S')" using insert by simp
  have **: "finite ?T'" using insert.prems(1) insert.hyps(1) σ(1) by simp
  have ***: "⋀y. y ∈ insert x ?S' ⟹ Γ (Var y) = TAtom a" using a by auto

  obtain δ where δ:
      "subst_domain δ = insert x ?S'" "bij_betw δ (subst_domain δ) (subst_range δ)"
      "subst_range δ ⊆ ((λc. Fun c []) ` 𝒞pub) - ?T'" "wtsubst δ" "wftrms (subst_range δ)"
    using wt_bij_finite_tatom_subst_exists_single[OF * ** ***] const_type_inv[of _ "[]" a]
    by blast

  obtain ϑ where ϑ: "ϑ ≡ λy. if x = y then δ y else σ y" by simp

  have x_dom: "x ∉ subst_domain σ" "x ∈ subst_domain δ" "x ∈ subst_domain ϑ"
    using insert.hyps(2) σ(1) δ(1) ϑ by (auto simp add: subst_domain_def)
  moreover have ground_imgs: "ground (subst_range σ)" "ground (subst_range δ)"
    using pgwt_ground σ(3) δ(3) by auto
  ultimately have x_img: "σ x ∉ subst_range σ" "δ x ∈ subst_range δ"
    using ground_subst_dom_iff_img by (auto simp add: subst_domain_def)

  have "ground (insert (δ x) (subst_range σ))" using ground_imgs x_dom by auto
  have ϑ_dom: "subst_domain ϑ = insert x (subst_domain σ)"
    using δ(1) ϑ by (auto simp add: subst_domain_def)
  have ϑ_img: "subst_range ϑ = insert (δ x) (subst_range σ)"
  proof
    show "subst_range ϑ ⊆ insert (δ x) (subst_range σ)"
    proof
      fix t assume "t ∈ subst_range ϑ"
      then obtain y where "y ∈ subst_domain ϑ" "t = ϑ y" by auto
      thus "t ∈ insert (δ x) (subst_range σ)" using ϑ by (auto simp add: subst_domain_def)
    qed
    show "insert (δ x) (subst_range σ) ⊆ subst_range ϑ"
    proof
      fix t assume t: "t ∈ insert (δ x) (subst_range σ)"
      hence "fv t = {}" using ground_imgs x_img(2) by auto
      hence "t ≠ Var x" by auto
      show "t ∈ subst_range ϑ"
      proof (cases "t = δ x")
        case True thus ?thesis using subst_imgI ϑ ‹t ≠ Var x› by metis
      next
        case False
        hence "t ∈ subst_range σ" using t by auto
        then obtain y where "σ y ∈ subst_range σ" "t = σ y" by auto
```

```
        hence "y ∈ subst_domain σ" "t ≠ Var y"
          using ground_subst_dom_iff_img[OF ground_imgs(1)]
          by (auto simp add: subst_domain_def simp del: subst_range.simps)
        hence "x ≠ y" using x_dom by auto
        hence "ϑ y = σ y" unfolding ϑ by auto
        thus ?thesis using ‹t ≠ Var y› ‹t = σ y› subst_imgI[of ϑ y] by auto
      qed
    qed
  qed
  hence ϑ_ground_img: "ground (subst_range ϑ)" using ground_imgs x_img by auto

  have "subst_domain ϑ = insert x S" using ϑ_dom σ(1) by auto
  moreover have "bij_betw ϑ (subst_domain ϑ) (subst_range ϑ)"
  proof (intro bij_betwI')
    fix y z assume *: "y ∈ subst_domain ϑ" "z ∈ subst_domain ϑ"
    hence "fv (ϑ y) = {}" "fv (ϑ z) = {}" using ϑ_ground_img by auto
    { assume "ϑ y = ϑ z" hence "y = z"
      proof (cases "ϑ y ∈ subst_range σ ∧ ϑ z ∈ subst_range σ")
        case True
        hence **: "y ∈ subst_domain σ" "z ∈ subst_domain σ"
          using ϑ ϑ_dom x_img(2) δ(3) True
          by (metis (no_types) *(1) DiffE Un_upper2 insertE subsetCE,
              metis (no_types) *(2) DiffE Un_upper2 insertE subsetCE)
        hence "y ≠ x" "z ≠ x" using x_dom by auto
        hence "ϑ y = σ y" "ϑ z = σ z" using ϑ by auto
        thus ?thesis using ‹ϑ y = ϑ z› σ(2) ** unfolding bij_betw_def inj_on_def by auto
      qed (metis ϑ * ‹ϑ y = ϑ z› ϑ_dom ground_imgs(1) ground_subst_dom_iff_img insertE)
    }
    thus "(ϑ y = ϑ z) = (y = z)" by auto
  next
    fix y assume "y ∈ subst_domain ϑ" thus "ϑ y ∈ subst_range ϑ" by auto
  next
    fix t assume "t ∈ subst_range ϑ" thus "∃z ∈ subst_domain ϑ. t = ϑ z" by auto
  qed
  moreover have "subst_range ϑ ⊆ (λc. Fun c []) ` C_pub - T"
    using σ(3) δ(3) ϑ by (auto simp add: subst_domain_def)
  moreover have "wt_subst ϑ" using σ(4) δ(4) ϑ unfolding wt_subst_def by auto
  moreover have "wf_trms (subst_range ϑ)"
    using ϑ σ(5) δ(5) wf_trm_subst_range_iff[of δ]
          wf_trm_subst_range_iff[of σ] wf_trm_subst_range_iff[of ϑ]
    by presburger
  ultimately show ?case by blast
qed

theorem wt_sat_if_simple:
  assumes "simple S" "wf_constr S ϑ" "wt_subst ϑ" "wf_trms (subst_range ϑ)" "wf_trms (trms_st S)"
  and ℐ': "∀X F. Inequality X F ∈ set S ⟶ ineq_model ℐ' X F"
          "ground (subst_range ℐ')"
          "subst_domain ℐ' = {x ∈ vars_st S. ∃X F. Inequality X F ∈ set S ∧ x ∈ fv_pairs F - set X}"
  and tfr_stp_all: "list_all tfr_stp S"
  shows "∃ℐ. interpretation_subst ℐ ∧ (ℐ ⊨_c ⟨S, ϑ⟩) ∧ wt_subst ℐ ∧ wf_trms (subst_range ℐ)"
proof -
  from ‹wf_constr S ϑ› have "wf_st {} S" "subst_idem ϑ" and S_ϑ_disj: "∀v ∈ vars_st S. ϑ v = Var v"
    using subst_idemI[of ϑ] unfolding wf_constr_def wf_subst_def by force+

  obtain ℐ::"('fun,'var) subst"
    where ℐ: "interpretation_subst ℐ" "wt_subst ℐ" "subst_range ℐ ⊆ public_ground_wf_terms"
    using wt_interpretation_exists by blast
  hence ℐ_deduct: "⋀x M. M ⊢_c ℐ x" and ℐ_wf_trm: "wf_trms (subst_range ℐ)"
    using pgwt_deducible pgwt_wellformed by fastforce+

  let ?P = "λδ X. subst_domain δ = set X ∧ ground (subst_range δ)"
  let ?Sineqsvars = "{x ∈ vars_st S. ∃X F. Inequality X F ∈ set S ∧ x ∈ fv_pairs F ∧ x ∉ set X}"
```

```
let ?Strms = "subterms_set (trms_st S)"

have finite_vars: "finite ?Sineqsvars" "finite ?Strms" "wf_trms ?Strms"
  using wf_trm_subtermeq assms(5) by fastforce+

define Q1 where "Q1 = (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
  ∀x ∈ fv_pairs F - set X. ∃a. Γ (Var x) = TAtom a)"

define Q2 where "Q2 = (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
  ∀f T. Fun f T ∈ subterms_set (trms_pairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))"

define Q1' where "Q1' = (λ(t::('fun,'var) term) (t'::('fun,'var) term) X.
  ∀x ∈ (fv t ∪ fv t') - set X. ∃a. Γ (Var x) = TAtom a)"

define Q2' where "Q2' = (λ(t::('fun,'var) term) (t'::('fun,'var) term) X.
  ∀f T. Fun f T ∈ subterms t ∪ subterms t' ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))"

have ex_P: "∀X. ∃δ. ?P δ X" using interpretation_subst_exists' by blast

have tfr_ineq: "∀X F. Inequality X F ∈ set S ⟶ Q1 F X ∨ Q2 F X"
  using tfr_stp_all Q1_def Q2_def tfr_stp_list_all_alt_def[of S] by blast

have S_fv_bvars_disj: "fv_st S ∩ bvars_st S = {}" using ‹wf_constr S ϑ› unfolding wf_constr_def by metis
hence ineqs_vars_not_bound: "∀X F x. Inequality X F ∈ set S ⟶ x ∈ ?Sineqsvars ⟶ x ∉ set X"
  using strand_fv_bvars_disjoint_unfold by blast

have ϑ_vars_S_bvars_disj: "(subst_domain ϑ ∪ range_vars ϑ) ∩ set X = {}"
  when "Inequality X F ∈ set S" for F X
  using wf_constr_bvars_disj[OF ‹wf_constr S ϑ›]
        strand_fv_bvars_disjointD(1)[OF S_fv_bvars_disj that]
  by blast

obtain σ::"('fun,'var) subst"
  where σ_fv_dom: "subst_domain σ = ?Sineqsvars"
  and σ_subterm_inj: "subterm_inj_on σ (subst_domain σ)"
  and σ_fresh_pub_img: "subterms_set (subst_range σ) ⊆ {t. {} ⊢_c t} - ?Strms"
  and σ_wt: "wt_subst σ"
  and σ_wf_trm: "wf_trms (subst_range σ)"
  using wt_bij_finite_subst_exists[OF finite_vars]
        subst_inj_on_is_bij_betw subterm_inj_on_alt_def'
  by moura

have σ_bij_dom_img: "bij_betw σ (subst_domain σ) (subst_range σ)"
  by (metis σ_subterm_inj subst_inj_on_is_bij_betw subterm_inj_on_alt_def)

have "finite (subst_domain σ)" by(metis σ_fv_dom finite_vars(1))
hence σ_finite_img: "finite (subst_range σ)" using σ_bij_dom_img bij_betw_finite by blast

have σ_img_subterms: "∀s ∈ subst_range σ. ∀u ∈ subst_range σ. (∃v. v ⊑ s ∧ v ⊑ u) ⟶ s = u"
  by (metis σ_subterm_inj subterm_inj_on_alt_def')

have "subst_range σ ⊆ subterms_set (subst_range σ)" by auto
hence "subst_range σ ⊆ public_ground_wf_terms - ?Strms"
    and σ_pgwt_img:
      "subst_range σ ⊆ public_ground_wf_terms"
      "subterms_set (subst_range σ) ⊆ public_ground_wf_terms"
  using σ_fresh_pub_img pgwt_is_empty_synth by blast+

have σ_img_ground: "ground (subst_range σ)"
  using σ_pgwt_img pgwt_ground by auto
hence σ_inj: "inj σ"
  using σ_bij_dom_img subst_inj_is_bij_betw_dom_img_if_ground_img by auto
```

```
have σ_ineqs_fv_dom: "⋀X F. Inequality X F ∈ set S ⟹ fv_pairs F - set X ⊆ subst_domain σ"
  using σ_fv_dom by fastforce

have σ_dom_bvars_disj: "∀X F. Inequality X F ∈ set S ⟶ subst_domain σ ∩ set X = {}"
  using ineqs_vars_not_bound σ_fv_dom by fastforce

have ℐ'1: "∀X F δ. Inequality X F ∈ set S ⟶ fv_pairs F - set X ⊆ subst_domain ℐ'"
  using ℐ'(3) ineqs_vars_not_bound by fastforce

have ℐ'2: "∀X F. Inequality X F ∈ set S ⟶ subst_domain ℐ' ∩ set X = {}"
  using ℐ'(3) ineqs_vars_not_bound by blast

have doms_eq: "subst_domain ℐ' = subst_domain σ" using ℐ'(3) σ_fv_dom by simp

have σ_ineqs_neq: "ineq_model σ X F" when "Inequality X F ∈ set S" for X F
proof -
  obtain a::"'fun" where a: "a ∉ ⋃(funs_term ` subterms_set (subst_range σ))"
    using exists_fun_notin_funs_terms[OF subterms_union_finite[OF σ_finite_img]]
    by moura
  hence a': "⋀T. Fun a T ∉ subterms_set (subst_range σ)"
            "⋀S. Fun a [] ∈ set (Fun a []#S)" "Fun a [] ∉ Var ` set X"
    by (meson a UN_I term.set_intros(1), auto)

  define t where "t ≡ Fun a (Fun a []#map fst F)"
  define t' where "t' ≡ Fun a (Fun a []#map snd F)"

  note F_in = that

  have t_fv: "fv t ∪ fv t' ⊆ fv_pairs F"
    unfolding t_def t'_def by force

  have t_subterms: "subterms t ∪ subterms t' ⊆ subterms_set (trms_pairs F) ∪ {t, t', Fun a []}"
    unfolding t_def t'_def by force

  have "t · δ · σ ≠ t' · δ · σ" when "?P δ X" for δ
  proof -
    have tfr_assms: "Q1 F X ∨ Q2 F X" using tfr_ineq F_in by metis

    have "Q1 F X ⟹ ∀x ∈ fv_pairs F - set X. ∃c. σ x = Fun c []"
    proof
      fix x assume "Q1 F X" and x: "x ∈ fv_pairs F - set X"
      then obtain a where "Γ (Var x) = TAtom a" unfolding Q1_def by moura
      hence a: "Γ (σ x) = TAtom a" using σ_wt unfolding wt_subst_def by simp

      have "x ∈ subst_domain σ" using σ_ineqs_fv_dom x F_in by auto
      then obtain f T where fT: "σ x = Fun f T" by (meson σ_img_ground ground_img_obtain_fun)
      hence "T = []" using σ_wf_trm a TAtom_term_cases by fastforce
      thus "∃c. σ x = Fun c []" using fT by metis
    qed
    hence 1: "Q1 F X ⟹ ∀x ∈ (fv t ∪ fv t') - set X. ∃c. σ x = Fun c []"
      using t_fv by auto

    have 2: "¬Q1 F X ⟹ Q2 F X" by (metis tfr_assms)

    have 3: "subst_domain σ ∩ set X = {}" using σ_dom_bvars_disj F_in by auto

    have 4: "subterms_set (subst_range σ) ∩ (subterms t ∪ subterms t') = {}"
    proof -
      define M1 where "M1 ≡ {t, t', Fun a []}"
      define M2 where "M2 ≡ ?Strms"

      have "subterms_set (trms_pairs F) ⊆ M2"
        using F_in unfolding M2_def by force
```

```
        moreover have "subterms t ∪ subterms t' ⊆ subterms_set (trms_pairs F) ∪ M1"
          using t_subterms unfolding M1_def by blast
        ultimately have *: "subterms t ∪ subterms t' ⊆ M2 ∪ M1"
          by auto

        have "subterms_set (subst_range σ) ∩ M1 = {}"
             "subterms_set (subst_range σ) ∩ M2 = {}"
          using a' σ_fresh_pub_img
          unfolding t_def t'_def M1_def M2_def
          by blast+
        thus ?thesis using * by blast
      qed

      have 5: "(fv t ∪ fv t') - subst_domain σ ⊆ set X"
        using σ_ineqs_fv_dom[OF F_in] t_fv
        by auto

      have 6: "∀δ. ?P δ X ⟶ t · δ · ℐ' ≠ t' · δ · ℐ'"
        by (metis t_def t'_def ℐ'(1) F_in ineq_model_singleE ineq_model_single_iff)

      have 7: "fv t ∪ fv t' - set X ⊆ subst_domain ℐ'" using ℐ'1 F_in t_fv by force

      have 8: "subst_domain ℐ' ∩ set X = {}" using ℐ'2 F_in by auto

      have 9: "Q1' t t' X" when "Q1 F X"
        using that t_fv
        unfolding Q1_def Q1'_def t_def t'_def
        by blast

      have 10: "Q2' t t' X" when "Q2 F X" unfolding Q2'_def
      proof (intro allI impI)
        fix f T assume "Fun f T ∈ subterms t ∪ subterms t'"
        moreover {
          assume "Fun f T ∈ subterms_set (trms_pairs F)"
          hence "T = [] ∨ (∃s∈set T. s ∉ Var ` set X)" by (metis Q2_def that)
        } moreover {
          assume "Fun f T = t" hence "T = [] ∨ (∃s∈set T. s ∉ Var ` set X)"
            unfolding t_def using a'(2,3) by simp
        } moreover {
          assume "Fun f T = t'" hence "T = [] ∨ (∃s∈set T. s ∉ Var ` set X)"
            unfolding t'_def using a'(2,3) by simp
        } moreover {
          assume "Fun f T = Fun a []" hence "T = [] ∨ (∃s∈set T. s ∉ Var ` set X)" by simp
        } ultimately show "T = [] ∨ (∃s∈set T. s ∉ Var ` set X)" using t_subterms by blast
      qed

      note 11 = σ_subterm_inj σ_img_ground 3 4 5

      note 12 = 6 7 8 ℐ'(2) doms_eq

      show "t · δ · σ ≠ t' · δ · σ"
        using 1 2 9 10 that sat_ineq_subterm_inj_subst[OF 11 _ 12]
        unfolding Q1'_def Q2'_def by metis
    qed
    thus ?thesis by (metis t_def t'_def ineq_model_singleI ineq_model_single_iff)
  qed

  have σ_ineqs_fv_dom': "fv_pairs (F ·pairs δ) ⊆ subst_domain σ"
    when "Inequality X F ∈ set S" and "?P δ X" for F δ X
    using σ_ineqs_fv_dom[OF that(1)]
  proof (induction F)
    case (Cons g G)
    obtain t t' where g: "g = (t,t')" by (metis surj_pair)
```

```
    hence "fv_pairs (g#G ·_pairs δ)  = fv (t · δ) ∪ fv (t' · δ) ∪ fv_pairs (G ·_pairs δ)"
          "fv_pairs (g#G) = fv t ∪ fv t' ∪ fv_pairs G"
      by (simp_all add: subst_apply_pairs_def)
    moreover have "fv (t · δ) = fv t - subst_domain δ" "fv (t' · δ) = fv t' - subst_domain δ"
      using g that(2) by (simp_all add: subst_fv_unfold_ground_img range_vars_alt_def)
    moreover have "fv_pairs (G ·_pairs δ) ⊆ subst_domain σ" using Cons by auto
    ultimately show ?case using Cons.prems that(2) by auto
  qed (simp add: subst_apply_pairs_def)

  have σ_ineqs_ground: "fv_pairs ((F ·_pairs δ) ·_pairs σ) = {}"
    when "Inequality X F ∈ set S" and "?P δ X" for F δ X
    using σ_ineqs_fv_dom'[OF that]
  proof (induction F)
    case (Cons g G)
    obtain t t' where g: "g = (t,t')" by (metis surj_pair)
    hence "fv (t · δ) ⊆ subst_domain σ" "fv (t' · δ) ⊆ subst_domain σ"
      using Cons.prems by (auto simp add: subst_apply_pairs_def)
    hence "fv (t · δ · σ) = {}" "fv (t' · δ · σ) = {}"
      using subst_fv_dom_ground_if_ground_img[OF _ σ_img_ground] by metis+
    thus ?case using g Cons by (auto simp add: subst_apply_pairs_def)
  qed (simp add: subst_apply_pairs_def)

  from σ_pgwt_img σ_ineqs_neq have σ_deduct: "M ⊢_c σ x" when "x ∈ subst_domain σ" for x M
    using that pgwt_deducible by fastforce

  { fix M::"('fun,'var) terms"
    have "⟦M; S⟧_c (ϑ ∘_s σ ∘_s ℐ)"
      using ‹wf_st {} S› ‹simple S› S_ϑ_disj σ_ineqs_neq σ_ineqs_fv_dom' ϑ_vars_S_bvars_disj
    proof (induction S arbitrary: M rule: wf_st_simple_induct)
      case (ConsSnd v S)
      hence S_sat: "⟦M; S⟧_c (ϑ ∘_s σ ∘_s ℐ)" and "ϑ v = Var v" by auto
      hence "⋀M. M ⊢_c Var v · (ϑ ∘_s σ ∘_s ℐ)"
        using ℐ_deduct σ_deduct
        by (metis ideduct_synth_subst_apply subst_apply_term.simps(1)
                  subst_subst_compose trm_subst_ident')
      thus ?case using strand_sem_append(1)[OF S_sat] by (metis strand_sem_c.simps(1,2))
    next
      case (ConsIneq X F S)
      have dom_disj: "subst_domain ϑ ∩ fv_pairs F = {}"
        using ConsIneq.prems(1) subst_dom_vars_in_subst
        by force
      hence *: "F ·_pairs ϑ = F" by blast

      have **: "ineq_model σ X F" by (meson ConsIneq.prems(2) in_set_conv_decomp)

      have "⋀x. x ∈ vars_st S ⟹ x ∈ vars_st (S@[Inequality X F])"
           "⋀x. x ∈ set S ⟹ x ∈ set (S@[Inequality X F])" by auto
      hence IH: "⟦M; S⟧_c (ϑ ∘_s σ ∘_s ℐ)" by (metis ConsIneq.IH ConsIneq.prems(1,2,3,4))

      have "ineq_model (σ ∘_s ℐ) X F"
      proof -
        have "fv_pairs (F ·_pairs δ) ⊆ subst_domain σ" when "?P δ X" for δ
          using ConsIneq.prems(3)[OF _ that] by simp
        hence "fv_pairs F - set X ⊆ subst_domain σ"
          using fv_pairs_subst_subset ex_P
          by (metis Diff_subset_conv Un_commute)
        thus ?thesis by (metis ineq_model_ground_subst[OF _ σ_img_ground **])
      qed
      hence "ineq_model (ϑ ∘_s σ ∘_s ℐ) X F"
        using * ineq_model_subst' subst_compose_assoc ConsIneq.prems(4)
        by (metis UnCI list.set_intros(1) set_append)
      thus ?case using IH by (auto simp add: ineq_model_def)
    qed auto
```

```
  }
  moreover have "wt_subst (ϑ ∘s σ ∘s ℐ)" "wf_trms (subst_range (ϑ ∘s σ ∘s ℐ))"
    by (metis wt_subst_compose ‹wt_subst ϑ› ‹wt_subst σ› ‹wt_subst ℐ›,
        metis assms(4) ℐ_wf_trm σ_wf_trm wf_trm_subst subst_img_comp_subset')
  ultimately show ?thesis
    using interpretation_comp(1)[OF ‹interpretation_subst ℐ›, of "ϑ ∘s σ"]
          subst_idem_support[OF ‹subst_idem ϑ›, of "σ ∘s ℐ"] subst_compose_assoc
    unfolding constr_sem_c_def by metis
qed
end
```

Theorem: Type-flaw resistant constraints are well-typed satisfiable (composition-only)

There exists well-typed models of satisfiable type-flaw resistant constraints in the semantics where the intruder is limited to composition only (i.e., he cannot perform decomposition/analysis of deducible messages).

```
theorem wt_attack_if_tfr_attack:
  assumes "interpretation_subst ℐ"
    and "ℐ ⊨c ⟨S, ϑ⟩"
    and "wf_constr S ϑ"
    and "wt_subst ϑ"
    and "tfr_st S"
    and "wf_trms (trms_st S)"
    and "wf_trms (subst_range ϑ)"
  obtains ℐτ where "interpretation_subst ℐτ"
    and "ℐτ ⊨c ⟨S, ϑ⟩"
    and "wt_subst ℐτ"
    and "wf_trms (subst_range ℐτ)"
proof -
  have tfr: "tfr_set (trms_st S)" "wf_trms (trms_st S)" "list_all tfr_stp S"
    using assms(5,6) unfolding tfr_st_def by metis+
  obtain S' ϑ' where *: "simple S'" "(S,ϑ) ⇝* (S',ϑ')" "⟦{}; S'⟧c ℐ"
    using LI_completeness[OF assms(3,2)] unfolding constr_sem_c_def
    by (meson term.order_refl)
  have **: "wf_constr S' ϑ'" "wt_subst ϑ'" "list_all tfr_stp S'" "wf_trms (trms_st S')" "wf_trms (subst_range
ϑ')"
    using LI_preserves_welltypedness[OF *(2) assms(3,4,7) tfr]
          LI_preserves_wellformedness[OF *(2) assms(3)]
          LI_preserves_tfr[OF *(2) assms(3,4,7) tfr]
    by metis+

  define A where "A ≡ {x ∈ vars_st S'. ∃X F. Inequality X F ∈ set S' ∧ x ∈ fv_pairs F ∧ x ∉ set X}"
  define B where "B ≡ UNIV - A"

  let ?ℐ = "rm_vars B ℐ"

  have grℐ: "ground (subst_range ℐ)" "ground (subst_range ?ℐ)"
    using assms(1) rm_vars_img_subset[of B ℐ] by (auto simp add: subst_domain_def)

  { fix X F
    assume "Inequality X F ∈ set S'"
    hence *: "ineq_model ℐ X F"
      using strand_sem_c_imp_ineq_model[OF *(3)]
      by (auto simp del: subst_range.simps)
    hence "ineq_model ?ℐ X F"
    proof -
      { fix δ
        assume 1: "subst_domain δ = set X" "ground (subst_range δ)"
            and 2: "list_ex (λf. fst f · δ ∘s ℐ ≠ snd f · δ ∘s ℐ) F"
        have "list_ex (λf. fst f · δ ∘s rm_vars B ℐ ≠ snd f · δ ∘s rm_vars B ℐ) F" using 2
        proof (induction F)
          case (Cons g G)
          obtain t t' where g: "g = (t,t')" by (metis surj_pair)
```

```
        thus ?case
          using Cons Unifier_ground_rm_vars[OF grℐ(1), of "t · δ" B "t' · δ"]
          by auto
      qed simp
    } thus ?thesis using * unfolding ineq_model_def by simp
  qed
} moreover have "subst_domain ℐ = UNIV" using assms(1) by metis
hence "subst_domain ?ℐ = A" using rm_vars_dom[of B ℐ] B_def by blast
ultimately obtain ℐτ where
    "interpretation_subst ℐτ" "ℐτ ⊨c ⟨S', ϑ'⟩" "wt_subst ℐτ" "wf_trms (subst_range ℐτ)"
  using wt_sat_if_simple[OF *(1) **(1,2,5,4) _ grℐ(2) _ **(3)] A_def
  by (auto simp del: subst_range.simps)
thus ?thesis using that LI_soundness[OF assms(3) *(2)] by metis
qed
```

Contra-positive version: if a type-flaw resistant constraint does not have a well-typed model then it is unsatisfiable

```
corollary secure_if_wt_secure:
  assumes "¬(∃ℐτ. interpretation_subst ℐτ ∧ (ℐτ ⊨c ⟨S, ϑ⟩) ∧ wt_subst ℐτ)"
  and     "wf_constr S ϑ" "wt_subst ϑ" "tfr_st S"
  and     "wf_trms (trms_st S)" "wf_trms (subst_range ϑ)"
  shows "¬(∃ℐ. interpretation_subst ℐ ∧ (ℐ ⊨c ⟨S, ϑ⟩))"
using wt_attack_if_tfr_attack[OF _ _ assms(2,3,4,5,6)] assms(1) by metis

end
```

3.4.2 Lifting the Composition-Only Typing Result to the Full Intruder Model

```
context typed_model
begin
```

Analysis Invariance

```
definition (in typed_model) Ana_invar_subst where
  "Ana_invar_subst ℳ ≡
    (∀f T K M δ. Fun f T ∈ (subterms_set ℳ) ⟶
                Ana (Fun f T) = (K, M) ⟶ Ana (Fun f T · δ) = (K ·list δ, M ·list δ))"

lemma (in typed_model) Ana_invar_subst_subset:
  assumes "Ana_invar_subst M" "N ⊆ M"
  shows "Ana_invar_subst N"
using assms unfolding Ana_invar_subst_def by blast

lemma (in typed_model) Ana_invar_substD:
  assumes "Ana_invar_subst ℳ"
  and "Fun f T ∈ subterms_set ℳ" "Ana (Fun f T) = (K, M)"
  shows "Ana (Fun f T · ℐ) = (K ·list ℐ, M ·list ℐ)"
using assms Ana_invar_subst_def by blast

end
```

Preliminary Definitions

Strands extended with "decomposition steps"

```
datatype (funs_estp: 'a, vars_estp: 'b) extstrand_step =
  Step   "('a,'b) strand_step"
| Decomp "('a,'b) term"

context typed_model
begin

context
begin
```

```
private fun trms_estp where
  "trms_estp (Step x) = trms_stp x"
| "trms_estp (Decomp t) = {t}"

private abbreviation trms_est where "trms_est S ≡ ⋃(trms_estp ` set S)"

private type_synonym ('a,'b) extstrand = "('a,'b) extstrand_step list"
private type_synonym ('a,'b) extstrands = "('a,'b) extstrand set"

private definition decomp::"('fun,'var) term ⇒ ('fun,'var) strand" where
  "decomp t ≡ (case (Ana t) of (K,T) ⇒ send⟨t⟩_st#map Send K@map Receive T)"

private fun to_st where
  "to_st [] = []"
| "to_st (Step x#S) = x#(to_st S)"
| "to_st (Decomp t#S) = (decomp t)@(to_st S)"

private fun to_est where
  "to_est [] = []"
| "to_est (x#S) = Step x#to_est S"

private abbreviation "ik_est A ≡ ik_st (to_st A)"
private abbreviation "wf_est V A ≡ wf_st V (to_st A)"
private abbreviation "assignment_rhs_est A ≡ assignment_rhs_st (to_st A)"
private abbreviation "vars_est A ≡ vars_st (to_st A)"
private abbreviation "wfrestrictedvars_est A ≡ wfrestrictedvars_st (to_st A)"
private abbreviation "bvars_est A ≡ bvars_st (to_st A)"
private abbreviation "fv_est A ≡ fv_st (to_st A)"
private abbreviation "funs_est A ≡ funs_st (to_st A)"

private definition wf_sts'::"('fun,'var) strands ⇒ ('fun,'var) extstrand ⇒ bool" where
  "wf_sts' 𝒮 𝒜 ≡ (∀S ∈ 𝒮. wf_st (wfrestrictedvars_est 𝒜) (dual_st S)) ∧
                  (∀S ∈ 𝒮. ∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}) ∧
                  (∀S ∈ 𝒮. fv_st S ∩ bvars_est 𝒜 = {}) ∧
                  (∀S ∈ 𝒮. fv_st (to_st 𝒜) ∩ bvars_st S = {})"

private definition wf_sts::"('fun,'var) strands ⇒ bool" where
  "wf_sts 𝒮 ≡ (∀S ∈ 𝒮. wf_st {} (dual_st S)) ∧ (∀S ∈ 𝒮. ∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {})"

private inductive well_analyzed::"('fun,'var) extstrand ⇒ bool" where
  Nil[simp]: "well_analyzed []"
| Step: "well_analyzed A ⟹ well_analyzed (A@[Step x])"
| Decomp: "⟦well_analyzed A; t ∈ subterms_set (ik_est A ∪ assignment_rhs_est A) - (Var ` 𝒱)⟧
    ⟹ well_analyzed (A@[Decomp t])"

private fun subst_apply_extstrandstep (infix "·_estp" 51) where
  "subst_apply_extstrandstep (Step x) ϑ = Step (x ·_stp ϑ)"
| "subst_apply_extstrandstep (Decomp t) ϑ = Decomp (t · ϑ)"

private lemma subst_apply_extstrandstep'_simps[simp]:
  "(Step (send⟨t⟩_st)) ·_estp ϑ = Step (send⟨t · ϑ⟩_st)"
  "(Step (receive⟨t⟩_st)) ·_estp ϑ = Step (receive⟨t · ϑ⟩_st)"
  "(Step (⟨a: t ≐ t'⟩_st)) ·_estp ϑ = Step (⟨a: (t · ϑ) ≐ (t' · ϑ)⟩_st)"
  "(Step (∀X⟨∨≠: F⟩_st)) ·_estp ϑ = Step (∀X⟨∨≠: (F ·_pairs rm_vars (set X) ϑ)⟩_st)"
by simp_all

private lemma vars_estp_subst_apply_simps[simp]:
  "vars_estp ((Step (send⟨t⟩_st)) ·_estp ϑ) = fv (t · ϑ)"
  "vars_estp ((Step (receive⟨t⟩_st)) ·_estp ϑ) = fv (t · ϑ)"
  "vars_estp ((Step (⟨a: t ≐ t'⟩_st)) ·_estp ϑ) = fv (t · ϑ) ∪ fv (t' · ϑ)"
  "vars_estp ((Step (∀X⟨∨≠: F⟩_st)) ·_estp ϑ) = set X ∪ fv_pairs (F ·_pairs rm_vars (set X) ϑ)"
by auto
```

private definition *subst_apply_extstrand* (**infix** "$\cdot_{est}$" 51) **where** "S $\cdot_{est}$ ϑ ≡ map (λx. x $\cdot_{estp}$ ϑ) S"

private abbreviation update$_{st}$::"('fun,'var) strands ⇒ ('fun,'var) strand ⇒ ('fun,'var) strands"
where
"update$_{st}$ $\mathcal{S}$ S ≡ (case S of Nil ⇒ $\mathcal{S}$ - {S} | Cons _ S' ⇒ insert S' ($\mathcal{S}$ - {S}))"

private inductive_set decomps$_{est}$::
"('fun,'var) terms ⇒ ('fun,'var) terms ⇒ ('fun,'var) subst ⇒ ('fun,'var) extstrands"

for $\mathcal{M}$ **and** $\mathcal{N}$ **and** $\mathcal{I}$ **where**
Nil: "[] ∈ decomps$_{est}$ $\mathcal{M}$ $\mathcal{N}$ $\mathcal{I}$"
| Decomp: "⟦$\mathcal{D}$ ∈ decomps$_{est}$ $\mathcal{M}$ $\mathcal{N}$ $\mathcal{I}$; Fun f T ∈ subterms$_{set}$ ($\mathcal{M}$ ∪ $\mathcal{N}$);
Ana (Fun f T) = (K,M); M ≠ [];
($\mathcal{M}$ ∪ ik$_{est}$ $\mathcal{D}$) $\cdot_{set}$ $\mathcal{I}$ $\vdash_c$ Fun f T · $\mathcal{I}$;
⋀k. k ∈ set K ⟹ ($\mathcal{M}$ ∪ ik$_{est}$ $\mathcal{D}$) $\cdot_{set}$ $\mathcal{I}$ $\vdash_c$ k · $\mathcal{I}$⟧
⟹ $\mathcal{D}$@[Decomp (Fun f T)] ∈ decomps$_{est}$ $\mathcal{M}$ $\mathcal{N}$ $\mathcal{I}$"

private fun decomp_rm$_{est}$::"('fun,'var) extstrand ⇒ ('fun,'var) extstrand" **where**
"decomp_rm$_{est}$ [] = []"
| "decomp_rm$_{est}$ (Decomp t#S) = decomp_rm$_{est}$ S"
| "decomp_rm$_{est}$ (Step x#S) = Step x#(decomp_rm$_{est}$ S)"

private inductive sem$_{est}$_d::"('fun,'var) terms ⇒ ('fun,'var) subst ⇒ ('fun,'var) extstrand ⇒ bool"
where
Nil[simp]: "sem$_{est}$_d M_0 $\mathcal{I}$ []"
| Send: "sem$_{est}$_d M_0 $\mathcal{I}$ S ⟹ (ik$_{est}$ S ∪ M_0) $\cdot_{set}$ $\mathcal{I}$ ⊢ t · $\mathcal{I}$ ⟹ sem$_{est}$_d M_0 $\mathcal{I}$ (S@[Step (send⟨t⟩$_{st}$)])"
| Receive: "sem$_{est}$_d M_0 $\mathcal{I}$ S ⟹ sem$_{est}$_d M_0 $\mathcal{I}$ (S@[Step (receive⟨t⟩$_{st}$)])"
| Equality: "sem$_{est}$_d M_0 $\mathcal{I}$ S ⟹ t · $\mathcal{I}$ = t' · $\mathcal{I}$ ⟹ sem$_{est}$_d M_0 $\mathcal{I}$ (S@[Step (⟨a: t ≐ t'⟩$_{st}$)])"
| Inequality: "sem$_{est}$_d M_0 $\mathcal{I}$ S
⟹ ineq_model $\mathcal{I}$ X F
⟹ sem$_{est}$_d M_0 $\mathcal{I}$ (S@[Step (∀X⟨∨≠: F⟩$_{st}$)])"
| Decompose: "sem$_{est}$_d M_0 $\mathcal{I}$ S ⟹ (ik$_{est}$ S ∪ M_0) $\cdot_{set}$ $\mathcal{I}$ ⊢ t · $\mathcal{I}$ ⟹ Ana t = (K, M)
⟹ (⋀k. k ∈ set K ⟹ (ik$_{est}$ S ∪ M_0) $\cdot_{set}$ $\mathcal{I}$ ⊢ k · $\mathcal{I}$) ⟹ sem$_{est}$_d M_0 $\mathcal{I}$ (S@[Decomp t])"

private inductive sem$_{est}$_c::"('fun,'var) terms ⇒ ('fun,'var) subst ⇒ ('fun,'var) extstrand ⇒ bool"
where
Nil[simp]: "sem$_{est}$_c M_0 $\mathcal{I}$ []"
| Send: "sem$_{est}$_c M_0 $\mathcal{I}$ S ⟹ (ik$_{est}$ S ∪ M_0) $\cdot_{set}$ $\mathcal{I}$ $\vdash_c$ t · $\mathcal{I}$ ⟹ sem$_{est}$_c M_0 $\mathcal{I}$ (S@[Step (send⟨t⟩$_{st}$)])"
| Receive: "sem$_{est}$_c M_0 $\mathcal{I}$ S ⟹ sem$_{est}$_c M_0 $\mathcal{I}$ (S@[Step (receive⟨t⟩$_{st}$)])"
| Equality: "sem$_{est}$_c M_0 $\mathcal{I}$ S ⟹ t · $\mathcal{I}$ = t' · $\mathcal{I}$ ⟹ sem$_{est}$_c M_0 $\mathcal{I}$ (S@[Step (⟨a: t ≐ t'⟩$_{st}$)])"
| Inequality: "sem$_{est}$_c M_0 $\mathcal{I}$ S
⟹ ineq_model $\mathcal{I}$ X F
⟹ sem$_{est}$_c M_0 $\mathcal{I}$ (S@[Step (∀X⟨∨≠: F⟩$_{st}$)])"
| Decompose: "sem$_{est}$_c M_0 $\mathcal{I}$ S ⟹ (ik$_{est}$ S ∪ M_0) $\cdot_{set}$ $\mathcal{I}$ $\vdash_c$ t · $\mathcal{I}$ ⟹ Ana t = (K, M)
⟹ (⋀k. k ∈ set K ⟹ (ik$_{est}$ S ∪ M_0) $\cdot_{set}$ $\mathcal{I}$ $\vdash_c$ k · $\mathcal{I}$) ⟹ sem$_{est}$_c M_0 $\mathcal{I}$ (S@[Decomp t])"

Preliminary Lemmata

private lemma wf$_{sts}$_wf$_{sts}$':
"wf$_{sts}$ $\mathcal{S}$ = wf$_{sts}$' $\mathcal{S}$ []"
by (simp add: wf$_{sts}$_def wf$_{sts}$'_def)

private lemma decomp_ik:
assumes "Ana t = (K,M)"
shows "ik$_{st}$ (decomp t) = set M"
using ik_rcv_map[of _ M] ik_rcv_map'[of _ M]
by (auto simp add: decomp_def inv_def assms)

private lemma decomp_assignment_rhs_empty:
assumes "Ana t = (K,M)"
shows "assignment_rhs$_{st}$ (decomp t) = {}"
by (auto simp add: decomp_def inv_def assms)

```
private lemma decomp_tfrstp:
  "list_all tfrstp (decomp t)"
by (auto simp add: decomp_def list_all_def)

private lemma trmsest_ikI:
  "t ∈ ikest A ⟹ t ∈ subtermsset (trmsest A)"
proof (induction A rule: to_st.induct)
  case (2 x S) thus ?case by (cases x) auto
next
  case (3 t' A)
  obtain K M where Ana: "Ana t' = (K,M)" by (metis surj_pair)
  show ?case using 3 decomp_ik[OF Ana] Ana_subterm[OF Ana] by auto
qed simp

private lemma trmsest_ik_assignment_rhsI:
  "t ∈ ikest A ∪ assignment_rhsest A ⟹ t ∈ subtermsset (trmsest A)"
proof (induction A rule: to_st.induct)
  case (2 x S) thus ?case
  proof (cases x)
    case (Equality ac t t') thus ?thesis using 2 by (cases ac) auto
  qed auto
next
  case (3 t' A)
  obtain K M where Ana: "Ana t' = (K,M)" by (metis surj_pair)
  show ?case
    using 3 decomp_ik[OF Ana] decomp_assignment_rhs_empty[OF Ana] Ana_subterm[OF Ana]
    by auto
qed simp

private lemma trmsest_ik_subtermsI:
  assumes "t ∈ subtermsset (ikest A)"
  shows "t ∈ subtermsset (trmsest A)"
proof -
  obtain t' where "t' ∈ ikest A" "t ⊑ t'" using trmsest_ikI assms by auto
  thus ?thesis by (meson contra_subsetD in_subterms_subset_Union trmsest_ikI)
qed

private lemma trmsestD:
  assumes "t ∈ trmsest A"
  shows "t ∈ trmsst (to_st A)"
using assms
proof (induction A)
  case (Cons a A)
  obtain K M where Ana: "Ana t = (K,M)" by (metis surj_pair)
  hence "t ∈ trmsst (decomp t)" unfolding decomp_def by force
  thus ?case using Cons.IH Cons.prems by (cases a) auto
qed simp

private lemma subst_apply_extstrand_nil[simp]:
  "[] ·est ϑ = []"
by (simp add: subst_apply_extstrand_def)

private lemma subst_apply_extstrand_singleton[simp]:
  "[Step (receive⟨t⟩st)] ·est ϑ = [Step (Receive (t · ϑ))]"
  "[Step (send⟨t⟩st)] ·est ϑ = [Step (Send (t · ϑ))]"
  "[Step (⟨a: t ≐ t'⟩st)] ·est ϑ = [Step (Equality a (t · ϑ) (t' · ϑ))]"
  "[Decomp t] ·est ϑ = [Decomp (t · ϑ)]"
unfolding subst_apply_extstrand_def by auto

private lemma extstrand_subst_hom:
  "(S@S') ·est ϑ = (S ·est ϑ)@(S' ·est ϑ)" "(x#S) ·est ϑ = (x ·estp ϑ)#(S ·est ϑ)"
unfolding subst_apply_extstrand_def by auto
```

```
private lemma decomp_vars:
  "wfrestrictedvars_st (decomp t) = fv t" "vars_st (decomp t) = fv t" "bvars_st (decomp t) = {}"
  "fv_st (decomp t) = fv t"
proof -
  obtain K M where Ana: "Ana t = (K,M)" by (metis surj_pair)
  hence "decomp t = send⟨t⟩_st#map Send K@map Receive M"
    unfolding decomp_def by simp
  moreover have "⋃(set (map fv K)) = fv_set (set K)" "⋃(set (map fv M)) = fv_set (set M)" by auto
  moreover have "fv_set (set K) ⊆ fv t" "fv_set (set M) ⊆ fv t"
    using Ana_subterm[OF Ana(1)] Ana_keys_fv[OF Ana(1)]
    by (simp_all add: UN_least psubsetD subtermeq_vars_subset)
  ultimately show
      "wfrestrictedvars_st (decomp t) = fv t" "vars_st (decomp t) = fv t" "bvars_st (decomp t) = {}"
      "fv_st (decomp t) = fv t"
    by auto
qed

private lemma bvars_est_cons: "bvars_est (x#X) = bvars_est [x] ∪ bvars_est X"
by (cases x) auto

private lemma bvars_est_append: "bvars_est (A@B) = bvars_est A ∪ bvars_est B"
proof (induction A)
  case (Cons x A) thus ?case using bvars_est_cons[of x "A@B"] bvars_est_cons[of x A] by force
qed simp

private lemma fv_est_cons: "fv_est (x#X) = fv_est [x] ∪ fv_est X"
by (cases x) auto

private lemma fv_est_append: "fv_est (A@B) = fv_est A ∪ fv_est B"
proof (induction A)
  case (Cons x A) thus ?case using fv_est_cons[of x "A@B"] fv_est_cons[of x A] by auto
qed simp

private lemma bvars_decomp: "bvars_est (A@[Decomp t]) = bvars_est A" "bvars_est (Decomp t#A) = bvars_est
A"
using bvars_est_append decomp_vars(3) by fastforce+

private lemma bvars_decomp_rm: "bvars_est (decomp_rm_est A) = bvars_est A"
using bvars_decomp by (induct A rule: decomp_rm_est.induct) simp_all+

private lemma fv_decomp_rm: "fv_est (decomp_rm_est A) ⊆ fv_est A"
by (induct A rule: decomp_rm_est.induct) auto

private lemma ik_assignment_rhs_decomp_fv:
  assumes "t ∈ subterms_set (ik_est A ∪ assignment_rhs_est A)"
  shows "fv_est (A@[Decomp t]) = fv_est A"
proof -
  have "fv_est (A@[Decomp t]) = fv_est A ∪ fv t" using fv_est_append decomp_vars by simp
  moreover have "fv_set (ik_est A ∪ assignment_rhs_est A) ⊆ fv_est A" by force
  moreover have "fv t ⊆ fv_set (ik_est A ∪ assignment_rhs_est A)"
    using fv_subset_subterms[OF assms(1)] by simp
  ultimately show ?thesis by blast
qed

private lemma wfrestrictedvars_est_decomp_rm_est_subset:
  "wfrestrictedvars_est (decomp_rm_est A) ⊆ wfrestrictedvars_est A"
by (induct A rule: decomp_rm_est.induct) auto+

private lemma wfrestrictedvars_est_eq_wfrestrictedvars_st:
  "wfrestrictedvars_est A = wfrestrictedvars_st (to_st A)"
by simp

private lemma decomp_set_unfold:
```

```
  assumes "Ana t = (K, M)"
  shows "set (decomp t) = {send⟨t⟩_st} ∪ (Send ` set K) ∪ (Receive ` set M)"
using assms unfolding decomp_def by auto

private lemma ik_est_finite: "finite (ik_est A)"
by (rule finite_ik_st)

private lemma assignment_rhs_est_finite: "finite (assignment_rhs_est A)"
by (rule finite_assignment_rhs_st)

private lemma to_est_append: "to_est (A@B) = to_est A@to_est B"
by (induct A rule: to_est.induct) auto

private lemma to_st_to_est_inv: "to_st (to_est A) = A"
by (induct A rule: to_est.induct) auto

private lemma to_st_append: "to_st (A@B) = (to_st A)@(to_st B)"
by (induct A rule: to_st.induct) auto

private lemma to_st_cons: "to_st (a#B) = (to_st [a])@(to_st B)"
using to_st_append[of "[a]" B] by simp

private lemma wfrestrictedvars_est_split:
  "wfrestrictedvars_est (x#S) = wfrestrictedvars_est [x] ∪ wfrestrictedvars_est S"
  "wfrestrictedvars_est (S@S') = wfrestrictedvars_est S ∪ wfrestrictedvars_est S'"
using to_st_cons[of x S] to_st_append[of S S'] by auto

private lemma ik_est_append: "ik_est (A@B) = ik_est A ∪ ik_est B"
by (metis ik_append to_st_append)

private lemma assignment_rhs_est_append:
  "assignment_rhs_est (A@B) = assignment_rhs_est A ∪ assignment_rhs_est B"
by (metis assignment_rhs_append to_st_append)

private lemma ik_est_cons: "ik_est (a#A) = ik_est [a] ∪ ik_est A"
by (metis ik_append to_st_cons)

private lemma ik_est_append_subst:
  "ik_est (A@B ·est ϑ) = ik_est (A ·est ϑ) ∪ ik_est (B ·est ϑ)"
  "ik_est (A@B) ·set ϑ = (ik_est A ·set ϑ) ∪ (ik_est B ·set ϑ)"
by (metis ik_est_append extstrand_subst_hom(1), simp add: image_Un to_st_append)

private lemma assignment_rhs_est_append_subst:
  "assignment_rhs_est (A@B ·est ϑ) = assignment_rhs_est (A ·est ϑ) ∪ assignment_rhs_est (B ·est ϑ)"
  "assignment_rhs_est (A@B) ·set ϑ = (assignment_rhs_est A ·set ϑ) ∪ (assignment_rhs_est B ·set ϑ)"
by (metis assignment_rhs_est_append extstrand_subst_hom(1), use assignment_rhs_est_append in blast)

private lemma ik_est_cons_subst:
  "ik_est (a#A ·est ϑ) = ik_est ([a ·estp ϑ]) ∪ ik_est (A ·est ϑ)"
  "ik_est (a#A) ·set ϑ = (ik_est [a] ·set ϑ) ∪ (ik_est A ·set ϑ)"
by (metis ik_est_cons extstrand_subst_hom(2), metis image_Un ik_est_cons)

private lemma decomp_rm_est_append: "decomp_rm_est (S@S') = (decomp_rm_est S)@(decomp_rm_est S')"
by (induct S rule: decomp_rm_est.induct) auto

private lemma decomp_rm_est_single[simp]:
  "decomp_rm_est [Step (send⟨t⟩_st)] = [Step (send⟨t⟩_st)]"
  "decomp_rm_est [Step (receive⟨t⟩_st)] = [Step (receive⟨t⟩_st)]"
  "decomp_rm_est [Decomp t] = []"
by auto

private lemma decomp_rm_est_ik_subset: "ik_est (decomp_rm_est S) ⊆ ik_est S"
proof (induction S rule: decomp_rm_est.induct)
```

```
  case (3 x S) thus ?case by (cases x) auto
qed auto

private lemma decomps_est_ik_subset: "D ∈ decomps_est M N 𝓘 ⟹ ik_est D ⊆ subterms_set (M ∪ N)"
proof (induction D rule: decomps_est.induct)
  case (Decomp D f T K M')
  have "ik_st (decomp (Fun f T)) ⊆ subterms (Fun f T)"
       "ik_st (decomp (Fun f T)) = ik_est [Decomp (Fun f T)]"
    using decomp_ik[OF Decomp.hyps(3)] Ana_subterm[OF Decomp.hyps(3)]
    by auto
  hence "ik_st (to_st [Decomp (Fun f T)]) ⊆ subterms_set (M ∪ N)"
    using in_subterms_subset_Union[OF Decomp.hyps(2)]
    by blast
  thus ?case using ik_est_append[of D "[Decomp (Fun f T)]"] using Decomp.IH by auto
qed simp

private lemma decomps_est_decomp_rm_est_empty: "D ∈ decomps_est M N 𝓘 ⟹ decomp_rm_est D = []"
by (induct D rule: decomps_est.induct) (auto simp add: decomp_rm_est_append)

private lemma decomps_est_append:
  assumes "A ∈ decomps_est S N 𝓘" "B ∈ decomps_est S N 𝓘"
  shows "A@B ∈ decomps_est S N 𝓘"
using assms(2)
proof (induction B rule: decomps_est.induct)
  case Nil show ?case using assms(1) by simp
next
  case (Decomp B f X K T)
  hence "S ∪ ik_est B ·_set 𝓘 ⊆ S ∪ ik_est (A@B) ·_set 𝓘" using ik_est_append by auto
  thus ?case
    using decomps_est.Decomp[OF Decomp.IH(1) Decomp.hyps(2,3,4)]
          ideduct_synth_mono[OF Decomp.hyps(5)]
          ideduct_synth_mono[OF Decomp.hyps(6)]
    by auto
qed

private lemma decomps_est_subterms:
  assumes "A' ∈ decomps_est M N 𝓘"
  shows "subterms_set (ik_est A') ⊆ subterms_set (M ∪ N)"
using assms
proof (induction A' rule: decomps_est.induct)
  case (Decomp D f X K T)
  hence "Fun f X ∈ subterms_set (M ∪ N)" by auto
  hence "subterms_set (set X) ⊆ subterms_set (M ∪ N)"
    using in_subterms_subset_Union[of "Fun f X" "M ∪ N"] params_subterms_Union[of X f]
    by blast
  moreover have "ik_st (to_st [Decomp (Fun f X)]) = set T" using Decomp.hyps(3) decomp_ik by simp
  hence "subterms_set (ik_st (to_st [Decomp (Fun f X)])) ⊆ subterms_set (set X)"
    using Ana_fun_subterm[OF Decomp.hyps(3)] by auto
  ultimately show ?case
    using ik_est_append[of D "[Decomp (Fun f X)]"] Decomp.IH
    by auto
qed simp

private lemma decomps_est_assignment_rhs_empty:
  assumes "A' ∈ decomps_est M N 𝓘"
  shows "assignment_rhs_est A' = {}"
using assms
by (induction A' rule: decomps_est.induct)
   (simp_all add: decomp_assignment_rhs_empty assignment_rhs_est_append)

private lemma decomps_est_finite_ik_append:
  assumes "finite M" "M ⊆ decomps_est A N 𝓘"
  shows "∃D ∈ decomps_est A N 𝓘. ik_est D = (⋃m ∈ M. ik_est m)"
```

```
using assms
proof (induction M rule: finite_induct)
  case empty
  moreover have "[] ∈ decomps_est A N 𝓘" "ik_st (to_st []) = {}" using decomps_est.Nil by auto
  ultimately show ?case by blast
next
  case (insert m M)
  then obtain D where "D ∈ decomps_est A N 𝓘" "ik_est D = (⋃m∈M. ik_st (to_st m))" by moura
  moreover have "m ∈ decomps_est A N 𝓘" using insert.prems(1) by blast
  ultimately show ?case using decomps_est_append[of D A N 𝓘 m] ik_est_append[of D m] by blast
qed

private lemma decomp_snd_exists[simp]: "∃D. decomp t = send⟨t⟩_st#D"
by (metis (mono_tags, lifting) decomp_def prod.case surj_pair)

private lemma decomp_nonnil[simp]: "decomp t ≠ []"
using decomp_snd_exists[of t] by fastforce

private lemma to_st_nil_inv[dest]: "to_st A = [] ⟹ A = []"
by (induct A rule: to_st.induct) auto

private lemma well_analyzedD:
  assumes "well_analyzed A" "Decomp t ∈ set A"
  shows "∃f T. t = Fun f T"
using assms
proof (induction A rule: well_analyzed.induct)
  case (Decomp A t')
  hence "∃f T. t' = Fun f T" by (cases t') auto
  moreover have "Decomp t ∈ set A ∨ t = t'" using Decomp by auto
  ultimately show ?case using Decomp.IH by auto
qed auto

private lemma well_analyzed_inv:
  assumes "well_analyzed (A@[Decomp t])"
  shows "t ∈ subterms_set (ik_est A ∪ assignment_rhs_est A) - (Var ` 𝒱)"
using assms well_analyzed.cases[of "A@[Decomp t]"] by fastforce

private lemma well_analyzed_split_left_single: "well_analyzed (A@[a]) ⟹ well_analyzed A"
by (induction "A@[a]" rule: well_analyzed.induct) auto

private lemma well_analyzed_split_left: "well_analyzed (A@B) ⟹ well_analyzed A"
proof (induction B rule: List.rev_induct)
  case (snoc b B) thus ?case using well_analyzed_split_left_single[of "A@B" b] by simp
qed simp

private lemma well_analyzed_append:
  assumes "well_analyzed A" "well_analyzed B"
  shows "well_analyzed (A@B)"
using assms(2,1)
proof (induction B rule: well_analyzed.induct)
  case (Step B x) show ?case using well_analyzed.Step[OF Step.IH[OF Step.prems]] by simp
next
  case (Decomp B t) thus ?case
    using well_analyzed.Decomp[OF Decomp.IH[OF Decomp.prems]] ik_est_append assignment_rhs_est_append
    by auto
qed simp_all

private lemma well_analyzed_singleton:
  "well_analyzed [Step (send⟨t⟩_st)]" "well_analyzed [Step (receive⟨t⟩_st)]"
  "well_analyzed [Step (⟨a: t ≐ t'⟩_st)]" "well_analyzed [Step (∀X⟨∨≠: F⟩_st)]"
  "¬well_analyzed [Decomp t]"
proof -
  show "well_analyzed [Step (send⟨t⟩_st)]" "well_analyzed [Step (receive⟨t⟩_st)]"
```

```
      "well_analyzed [Step (⟨a: t ≐ t'⟩st)]" "well_analyzed [Step (∀X⟨∨≠: F⟩st)]"
    using well_analyzed.Step[OF well_analyzed.Nil]
    by simp_all

  show "¬well_analyzed [Decomp t]" using well_analyzed.cases[of "[Decomp t]"] by auto
qed

private lemma well_analyzed_decomp_rm_est_fv: "well_analyzed A ⟹ fv_est (decomp_rm_est A) = fv_est A"
proof
  assume "well_analyzed A" thus "fv_est A ⊆ fv_est (decomp_rm_est A)"
  proof (induction A rule: well_analyzed.induct)
    case Decomp thus ?case using ik_assignment_rhs_decomp_fv decomp_rm_est_append by auto
  next
    case (Step A x)
    have "fv_est (A@[Step x]) = fv_est A ∪ fv_stp x"
         "fv_est (decomp_rm_est (A@[Step x])) = fv_est (decomp_rm_est A) ∪ fv_stp x"
      using fv_est_append decomp_rm_est_append by auto
    thus ?case using Step by auto
  qed simp
qed (rule fv_decomp_rm)

private lemma sem_est_d_split_left: assumes "sem_est_d M0 I (A@A')" shows "sem_est_d M0 I A"
using assms sem_est_d.cases by (induction A' rule: List.rev_induct) fastforce+

private lemma sem_est_d_eq_sem_st: "sem_est_d M0 I A = ⟦M0; to_st A⟧d' I"
proof
  show "⟦M0; to_st A⟧d' I ⟹ sem_est_d M0 I A"
  proof (induction A arbitrary: M0 rule: List.rev_induct)
    case Nil show ?case using to_st_nil_inv by simp
  next
    case (snoc a A)
    hence IH: "sem_est_d M0 I A" and *: "⟦ik_est A ∪ M0; to_st [a]⟧d' I"
      using to_st_append by (auto simp add: sup.commute)
    thus ?case using snoc
    proof (cases a)
      case (Step b) thus ?thesis
      proof (cases b)
        case (Send t) thus ?thesis using sem_est_d.Send[OF IH] * Step by auto
      next
        case (Receive t) thus ?thesis using sem_est_d.Receive[OF IH] Step by auto
      next
        case (Equality a t t') thus ?thesis using sem_est_d.Equality[OF IH] * Step by auto
      next
        case (Inequality X F) thus ?thesis using sem_est_d.Inequality[OF IH] * Step by auto
      qed
    next
      case (Decomp t)
      obtain K M where Ana: "Ana t = (K,M)" by moura
      have "to_st [a] = decomp t" using Decomp by auto
      hence "to_st [a] = send⟨t⟩st#map Send K@map Receive M"
        using Ana unfolding decomp_def by auto
      hence **: "ik_est A ∪ M0 ·set I ⊢ t · I" and "⟦ik_est A ∪ M0; map Send K⟧d' I"
        using * by auto
      hence "⋀k. k ∈ set K ⟹ ik_est A ∪ M0 ·set I ⊢ k · I"
        using *
        by (metis (full_types) strand_sem_d.simps(2) strand_sem_eq_defs(2) strand_sem_Send_split(2))
      thus ?thesis using Decomp sem_est_d.Decompose[OF IH ** Ana] by metis
    qed
  qed

  show "sem_est_d M0 I A ⟹ ⟦M0; to_st A⟧d' I"
  proof (induction rule: sem_est_d.induct)
    case Nil thus ?case by simp
```

```
  next
    case (Send M₀ ℐ 𝒜 t) thus ?case
      using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[send⟨t⟩st]"]
            to_st_append[of 𝒜 "[Step (send⟨t⟩st)]"]
      by (simp add: sup.commute)
  next
    case (Receive M₀ ℐ 𝒜 t) thus ?case
      using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[receive⟨t⟩st]"]
            to_st_append[of 𝒜 "[Step (receive⟨t⟩st)]"]
      by (simp add: sup.commute)
  next
    case (Equality M₀ ℐ 𝒜 t t' a) thus ?case
      using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[⟨a: t ≐ t'⟩st]"]
            to_st_append[of 𝒜 "[Step (⟨a: t ≐ t'⟩st)]"]
      by (simp add: sup.commute)
  next
    case (Inequality M₀ ℐ 𝒜 X F) thus ?case
      using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[∀X⟨∨≠: F⟩st]"]
            to_st_append[of 𝒜 "[Step (∀X⟨∨≠: F⟩st)]"]
      by (simp add: sup.commute)
  next
    case (Decompose M₀ ℐ 𝒜 t K M)
    have "⟦M₀ ∪ ikst (to_st 𝒜); decomp t⟧d' ℐ"
    proof -
      have "⟦M₀ ∪ ikst (to_st 𝒜); [send⟨t⟩st]⟧d' ℐ"
        using Decompose.hyps(2) by (auto simp add: sup.commute)
      moreover have "⋀k. k ∈ set K ⟹ M₀ ∪ ikst (to_st 𝒜) ·set ℐ ⊢ k · ℐ"
        using Decompose by (metis sup.commute)
      hence "⋀k. k ∈ set K ⟹ ⟦M₀ ∪ ikst (to_st 𝒜); [Send k]⟧d' ℐ" by auto
      hence "⟦M₀ ∪ ikst (to_st 𝒜); map Send K⟧d' ℐ"
        using strand_sem_Send_map(2)[of K, of "M₀ ∪ ikst (to_st 𝒜) ·set ℐ" ℐ] strand_sem_eq_defs(2)
        by auto
      moreover have "⟦M₀ ∪ ikst (to_st 𝒜); map Receive M⟧d' ℐ"
        by (metis strand_sem_Receive_map(2) strand_sem_eq_defs(2))
      ultimately have
          "⟦M₀ ∪ ikst (to_st 𝒜); send⟨t⟩st#map Send K@map Receive M⟧d' ℐ"
        by auto
      thus ?thesis using Decompose.hyps(3) unfolding decomp_def by auto
    qed
    hence "⟦M₀; to_st 𝒜@decomp t⟧d' ℐ"
      using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "decomp t"] Decompose.IH
      by simp
    thus ?case using to_st_append[of 𝒜 "[Decomp t]"] by simp
  qed
qed

private lemma semest_c_eq_sem_st: "semest_c M₀ ℐ 𝒜 = ⟦M₀; to_st 𝒜⟧c' ℐ"
proof
  show "⟦M₀; to_st 𝒜⟧c' ℐ ⟹ semest_c M₀ ℐ 𝒜"
  proof (induction 𝒜 arbitrary: M₀ rule: List.rev_induct)
    case Nil show ?case using to_st_nil_inv by simp
  next
    case (snoc a 𝒜)
    hence IH: "semest_c M₀ ℐ 𝒜" and *: "⟦ikest 𝒜 ∪ M₀; to_st [a]⟧c' ℐ"
      using to_st_append
      by (auto simp add: sup.commute)
    thus ?case using snoc
    proof (cases a)
      case (Step b) thus ?thesis
      proof (cases b)
        case (Send t) thus ?thesis using semest_c.Send[OF IH] * Step by auto
      next
        case (Receive t) thus ?thesis using semest_c.Receive[OF IH] Step by auto
```

```
    next
      case (Equality t) thus ?thesis using sem_est_c.Equality[OF IH] * Step by auto
    next
      case (Inequality t) thus ?thesis using sem_est_c.Inequality[OF IH] * Step by auto
    qed
  next
    case (Decomp t)
    obtain K M where Ana: "Ana t = (K,M)" by moura
    have "to_st [a] = decomp t" using Decomp by auto
    hence "to_st [a] = send⟨t⟩_st#map Send K@map Receive M"
      using Ana unfolding decomp_def by auto
    hence **: "ik_est 𝒜 ∪ M₀ ·_set ℐ ⊢_c t · ℐ" and "⟦ik_est 𝒜 ∪ M₀; map Send K⟧_c' ℐ"
      using * by auto
    hence "⋀k. k ∈ set K ⟹ ik_est 𝒜 ∪ M₀ ·_set ℐ ⊢_c k · ℐ"
      using * strand_sem_Send_split(1) strand_sem_eq_defs(1)
      by auto
    thus ?thesis using Decomp sem_est_c.Decompose[OF IH ** Ana] by metis
  qed
qed

show "sem_est_c M₀ ℐ 𝒜 ⟹ ⟦M₀; to_st 𝒜⟧_c' ℐ"
proof (induction rule: sem_est_c.induct)
  case Nil thus ?case by simp
next
  case (Send M₀ ℐ 𝒜 t) thus ?case
    using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[send⟨t⟩_st]"]
          to_st_append[of 𝒜 "[Step (send⟨t⟩_st)]"]
    by (simp add: sup.commute)
next
  case (Receive M₀ ℐ 𝒜 t) thus ?case
    using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[receive⟨t⟩_st]"]
          to_st_append[of 𝒜 "[Step (receive⟨t⟩_st)]"]
    by (simp add: sup.commute)
next
  case (Equality M₀ ℐ 𝒜 t t' a) thus ?case
    using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[⟨a: t ≐ t'⟩_st]"]
          to_st_append[of 𝒜 "[Step (⟨a: t ≐ t'⟩_st)]"]
    by (simp add: sup.commute)
next
  case (Inequality M₀ ℐ 𝒜 X F) thus ?case
    using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "[∀X⟨∨≠: F⟩_st]"]
          to_st_append[of 𝒜 "[Step (∀X⟨∨≠: F⟩_st)]"]
    by (auto simp add: sup.commute)
next
  case (Decompose M₀ ℐ 𝒜 t K M)
  have "⟦M₀ ∪ ik_st (to_st 𝒜); decomp t⟧_c' ℐ"
  proof -
    have "⟦M₀ ∪ ik_st (to_st 𝒜); [send⟨t⟩_st]⟧_c' ℐ"
      using Decompose.hyps(2) by (auto simp add: sup.commute)
    moreover have "⋀k. k ∈ set K ⟹ M₀ ∪ ik_st (to_st 𝒜) ·_set ℐ ⊢_c k · ℐ"
      using Decompose by (metis sup.commute)
    hence "⋀k. k ∈ set K ⟹ ⟦M₀ ∪ ik_st (to_st 𝒜); [Send k]⟧_c' ℐ" by auto
    hence "⟦M₀ ∪ ik_st (to_st 𝒜); map Send K⟧_c' ℐ"
      using strand_sem_Send_map(1)[of K, of "M₀ ∪ ik_st (to_st 𝒜) ·_set ℐ" ℐ]
            strand_sem_eq_defs(1)
      by auto
    moreover have "⟦M₀ ∪ ik_st (to_st 𝒜); map Receive M⟧_c' ℐ"
      by (metis strand_sem_Receive_map(1) strand_sem_eq_defs(1))
    ultimately have
        "⟦M₀ ∪ ik_st (to_st 𝒜); send⟨t⟩_st#map Send K@map Receive M⟧_c' ℐ"
      by auto
    thus ?thesis using Decompose.hyps(3) unfolding decomp_def by auto
  qed
```

```
    hence "⟦M₀; to_st 𝒜@decomp t⟧c' ℐ"
      using strand_sem_append'[of M₀ "to_st 𝒜" ℐ "decomp t"] Decompose.IH
      by simp
    thus ?case using to_st_append[of 𝒜 "[Decomp t]"] by simp
  qed
qed

private lemma sem_est_c_decomp_rm_est_deduct_aux:
  assumes "sem_est_c M₀ ℐ A" "t ∈ ik_est A ·set ℐ" "t ∉ ik_est (decomp_rm_est A) ·set ℐ"
  shows "ik_est (decomp_rm_est A) ∪ M₀ ·set ℐ ⊢ t"
using assms
proof (induction M₀ ℐ A arbitrary: t rule: sem_est_c.induct)
  case (Send M₀ ℐ A t') thus ?case using decomp_rm_est_append ik_est_append by auto
next
  case (Receive M₀ ℐ A t')
  hence "t ∈ ik_est A ·set ℐ" "t ∉ ik_est (decomp_rm_est A) ·set ℐ"
    using decomp_rm_est_append ik_est_append by auto
  hence IH: "ik_est (decomp_rm_est A) ∪ M₀ ·set ℐ ⊢ t" using Receive.IH by auto
  show ?case using ideduct_mono[OF IH] decomp_rm_est_append ik_est_append by auto
next
  case (Equality M₀ ℐ A t') thus ?case using decomp_rm_est_append ik_est_append by auto
next
  case (Inequality M₀ ℐ A t') thus ?case using decomp_rm_est_append ik_est_append by auto
next
  case (Decompose M₀ ℐ A t' K M t)
  have *: "ik_est (decomp_rm_est A) ∪ M₀ ·set ℐ ⊢ t' · ℐ" using Decompose.hyps(2)
  proof (induction rule: intruder_synth_induct)
    case (AxiomC t'')
    moreover {
      assume "t'' ∈ ik_est A ·set ℐ" "t'' ∉ ik_est (decomp_rm_est A) ·set ℐ"
      hence ?case using Decompose.IH by auto
    }
    ultimately show ?case by force
  qed simp

  { fix k assume "k ∈ set K"
    hence "ik_est A ∪ M₀ ·set ℐ ⊢c k · ℐ" using Decompose.hyps by auto
    hence "ik_est (decomp_rm_est A) ∪ M₀ ·set ℐ ⊢ k · ℐ"
    proof (induction rule: intruder_synth_induct)
      case (AxiomC t'')
      moreover {
        assume "t'' ∈ ik_est A ·set ℐ" "t'' ∉ ik_est (decomp_rm_est A) ·set ℐ"
        hence ?case using Decompose.IH by auto
      }
      ultimately show ?case by force
    qed simp
  }
  hence **: "⋀k. k ∈ set (K ·list ℐ) ⟹ ik_est (decomp_rm_est A) ∪ M₀ ·set ℐ ⊢ k" by auto

  show ?case
  proof (cases "t ∈ ik_est A ·set ℐ")
    case True thus ?thesis using Decompose.IH Decompose.prems(2) decomp_rm_est_append by auto
  next
    case False
    hence "t ∈ ik_st (decomp t') ·set ℐ" using Decompose.prems(1) ik_est_append by auto
    hence ***: "t ∈ set (M ·list ℐ)" using Decompose.hyps(3) decomp_ik by auto
    hence "M ≠ []" by auto
    hence ****: "Ana (t' · ℐ) = (K ·list ℐ, M ·list ℐ)" using Ana_subst[OF Decompose.hyps(3)] by auto

    have "ik_est (decomp_rm_est A) ∪ M₀ ·set ℐ ⊢ t" by (rule intruder_deduct.Decompose[OF * **** **
***])
    thus ?thesis using ideduct_mono decomp_rm_est_append by auto
  qed
```

```
qed simp

private lemma sem_est_c_decomp_rm_est_deduct:
  assumes "sem_est_c M0 I A" "ik_est A ∪ M0 ·set I ⊢c t"
  shows "ik_est (decomp_rm_est A) ∪ M0 ·set I ⊢ t"
using assms(2)
proof (induction t rule: intruder_synth_induct)
  case (AxiomC t)
  hence "t ∈ ik_est A ·set I ∨ t ∈ M0 ·set I" by auto
  moreover {
    assume "t ∈ ik_est A ·set I" "t ∈ ik_est (decomp_rm_est A) ·set I"
    hence ?case using ideduct_mono[OF intruder_deduct.Axiom] by auto
  }
  moreover {
    assume "t ∈ ik_est A ·set I" "t ∉ ik_est (decomp_rm_est A) ·set I"
    hence ?case using sem_est_c_decomp_rm_est_deduct_aux[OF assms(1)] by auto
  }
  ultimately show ?case by auto
qed simp

private lemma sem_est_d_decomp_rm_est_if_sem_est_c: "sem_est_c M0 I A ⟹ sem_est_d M0 I (decomp_rm_est A)"
proof (induction M0 I A rule: sem_est_c.induct)
  case (Send M0 I A t)
  thus ?case using decomp_rm_est_append sem_est_d.Send[OF Send.IH] sem_est_c_decomp_rm_est_deduct by auto
next
  case (Receive t) thus ?case using decomp_rm_est_append sem_est_d.Receive by auto
next
  case (Equality M0 I A t)
  thus ?case
    using decomp_rm_est_append sem_est_d.Equality[OF Equality.IH] sem_est_c_decomp_rm_est_deduct
    by auto
next
  case (Inequality M0 I A t)
  thus ?case
    using decomp_rm_est_append sem_est_d.Inequality[OF Inequality.IH] sem_est_c_decomp_rm_est_deduct
    by auto
next
  case Decompose thus ?case using decomp_rm_est_append by auto
qed auto

private lemma sem_est_c_decomps_est_append:
  assumes "sem_est_c {} I A" "D ∈ decomps_est (ik_est A) (assignment_rhs_est A) I"
  shows "sem_est_c {} I (A@D)"
using assms(2,1)
proof (induction D rule: decomps_est.induct)
  case (Decomp D f T K M)
  hence *: "sem_est_c {} I (A @ D)" "ik_est (A@D) ∪ {} ·set I ⊢c Fun f T · I"
          "⋀k. k ∈ set K ⟹ ik_est (A @ D) ∪ {} ·set I ⊢c k · I"
    using ik_est_append by auto
  show ?case using sem_est_c.Decompose[OF *(1,2) Decomp.hyps(3) *(3)] by simp
qed auto

private lemma decomps_est_preserves_wf:
  assumes "D ∈ decomps_est (ik_est A) (assignment_rhs_est A) I" "wf_est V A"
  shows "wf_est V (A@D)"
using assms
proof (induction D rule: decomps_est.induct)
  case (Decomp D f T K M)
  have "wfrestrictedvars_st (decomp (Fun f T)) ⊆ fv_set (ik_est A ∪ assignment_rhs_est A)"
    using decomp_vars fv_subset_subterms[OF Decomp.hyps(2)] by fast
  hence "wfrestrictedvars_st (decomp (Fun f T)) ⊆ wfrestrictedvars_est A"
    using ik_st_assignment_rhs_st_wfrestrictedvars_subset[of "to_st A"] by blast
  hence "wfrestrictedvars_st (decomp (Fun f T)) ⊆ wfrestrictedvars_st (to_st (A@D)) ∪ V"
```

```
    using to_st_append[of A D] strand_vars_split(2)[of "to_st A" "to_st D"]
    by (metis le_supI1)
  thus ?case
    using wf_append_suffix[OF Decomp.IH[OF Decomp.prems], of "decomp (Fun f T)"]
          to_st_append[of "A@D" "[Decomp (Fun f T)]"]
    by auto
qed auto

private lemma decomps_est_preserves_model_c:
  assumes "D ∈ decomps_est (ik_est A) (assignment_rhs_est A) ℐ" "sem_est_c M0 ℐ A"
  shows "sem_est_c M0 ℐ (A@D)"
using assms
proof (induction D rule: decomps_est.induct)
  case (Decomp D f T K M) show ?case
    using sem_est_c.Decompose[OF Decomp.IH[OF Decomp.prems] _ Decomp.hyps(3)]
          Decomp.hyps(5,6) ideduct_synth_mono ik_est_append
    by (metis (mono_tags, lifting) List.append_assoc image_Un sup_ge1)
qed auto

private lemma decomps_est_exist_aux:
  assumes "D ∈ decomps_est M N ℐ" "M ∪ ik_est D ⊢ t" "¬(M ∪ (ik_est D) ⊢c t)"
  obtains D' where
    "D@D' ∈ decomps_est M N ℐ" "M ∪ ik_est (D@D') ⊢c t" "M ∪ ik_est D ⊂ M ∪ ik_est (D@D')"
proof -
  have "∃D' ∈ decomps_est M N ℐ. M ∪ ik_est D' ⊢c t" using assms(2)
  proof (induction t rule: intruder_deduct_induct)
    case (Compose X f)
    from Compose.IH have "∃D ∈ decomps_est M N ℐ. ∀x ∈ set X. M ∪ ik_est D ⊢c x"
    proof (induction X)
      case (Cons t X)
      then obtain D' D'' where
          D': "D' ∈ decomps_est M N ℐ" "M ∪ ik_est D' ⊢c t" and
          D'': "D'' ∈ decomps_est M N ℐ" "∀x ∈ set X. M ∪ ik_est D'' ⊢c x"
        by moura
      hence "M ∪ ik_est (D'@D'') ⊢c t" "∀x ∈ set X. M ∪ ik_est (D'@D'') ⊢c x"
        by (auto intro: ideduct_synth_mono simp add: ik_est_append)
      thus ?case using decomps_est_append[OF D'(1) D''(1)] by (metis set_ConsD)
    qed (auto intro: decomps_est.Nil)
    thus ?case using intruder_synth.ComposeC[OF Compose.hyps(1,2)] by metis
  next
    case (Decompose t K T t_i)
    have "∃D ∈ decomps_est M N ℐ. ∀k ∈ set K. M ∪ ik_est D ⊢c k" using Decompose.IH
    proof (induction K)
      case (Cons t X)
      then obtain D' D'' where
          D': "D' ∈ decomps_est M N ℐ" "M ∪ ik_est D' ⊢c t" and
          D'': "D'' ∈ decomps_est M N ℐ" "∀x ∈ set X. M ∪ ik_est D'' ⊢c x"
        using assms(1) by moura
      hence "M ∪ ik_est (D'@D'') ⊢c t" "∀x ∈ set X. M ∪ ik_est (D'@D'') ⊢c x"
        by (auto intro: ideduct_synth_mono simp add: ik_est_append)
      thus ?case using decomps_est_append[OF D'(1) D''(1)] by auto
    qed auto
    then obtain D' where D': "D' ∈ decomps_est M N ℐ" "⋀k. k ∈ set K ⟹ M ∪ ik_est D' ⊢c k" by
metis
    obtain D'' where D'': "D'' ∈ decomps_est M N ℐ" "M ∪ ik_est D'' ⊢c t" by (metis Decompose.IH(1))
    obtain f X where fX: "t = Fun f X" "t_i ∈ set X"
      using Decompose.hyps(2,4) by (cases t) (auto dest: Ana_fun_subterm)

    from decomps_est_append[OF D'(1) D''(1)] D'(2) D''(2) have *:
        "D'@D'' ∈ decomps_est M N ℐ" "⋀k. k ∈ set K ⟹ M ∪ ik_est (D'@D'') ⊢c k"
        "M ∪ ik_est (D'@D'') ⊢c t"
      by (auto intro: ideduct_synth_mono simp add: ik_est_append)
    hence **: "⋀k. k ∈ set K ⟹ M ∪ ik_est (D'@D'') ·set ℐ ⊢c k · ℐ"
```

```
      using ideduct_synth_subst by auto

    have "tᵢ ∈ ik_st (decomp t)" using Decompose.hyps(2,4) ik_rcv_map unfolding decomp_def by auto
    with *(3) fX(1) Decompose.hyps(2) show ?case
    proof (induction t rule: intruder_synth_induct)
      case (AxiomC t)
      hence t_in_subterms: "t ∈ subterms_set (M ∪ N)"
        using decomps_est_ik_subset[OF *(1)] subset_subterms_Union
        by auto
      have "M ∪ ik_est (D'@D'') ·set 𝓘 ⊢c t · 𝓘"
        using ideduct_synth_subst[OF intruder_synth.AxiomC[OF AxiomC.hyps(1)]] by metis
      moreover have "T ≠ []" using decomp_ik[OF ‹Ana t = (K,T)›] ‹tᵢ ∈ ik_st (decomp t)› by auto
      ultimately have "D'@D''@[Decomp (Fun f X)] ∈ decomps_est M N 𝓘"
        using AxiomC decomps_est.Decomp[OF *(1) _ _ _ _ **] subset_subterms_Union t_in_subterms
        by (simp add: subset_eq)
      moreover have "decomp t = to_st [Decomp (Fun f X)]" using AxiomC.prems(1,2) by auto
      ultimately show ?case
        by (metis AxiomC.prems(3) UnCI intruder_synth.AxiomC ik_est_append to_st_append)
    qed (auto intro!: fX(2) *(1))
  qed (fastforce intro: intruder_synth.AxiomC assms(1))
  hence "∃D' ∈ decomps_est M N 𝓘. M ∪ ik_est (D@D') ⊢c t"
    by (auto intro: ideduct_synth_mono simp add: ik_est_append)
  thus thesis using that[OF decomps_est_append[OF assms(1)]] assms ik_est_append by moura
qed

private lemma decomps_est_ik_max_exist:
  assumes "finite A" "finite N"
  shows "∃D ∈ decomps_est A N 𝓘. ∀D' ∈ decomps_est A N 𝓘. ik_est D' ⊆ ik_est D"
proof -
  let ?IK = "λM. ⋃D ∈ M. ik_est D"
  have "?IK (decomps_est A N 𝓘) ⊆ (⋃t ∈ A ∪ N. subterms t)" by (auto dest!: decomps_est_ik_subset)
  hence "finite (?IK (decomps_est A N 𝓘))"
    using subterms_union_finite[OF assms(1)] subterms_union_finite[OF assms(2)] infinite_super
    by auto
  then obtain M where M: "finite M" "M ⊆ decomps_est A N 𝓘" "?IK M = ?IK (decomps_est A N 𝓘)"
    using finite_subset_Union by moura
  show ?thesis using decomps_est_finite_ik_append[OF M(1,2)] M(3) by auto
qed

private lemma decomps_est_exist:
  assumes "finite A" "finite N"
  shows "∃D ∈ decomps_est A N 𝓘. ∀t. A ⊢ t ⟶ A ∪ ik_est D ⊢c t"
proof (rule ccontr)
  assume neg: "¬(∃D ∈ decomps_est A N 𝓘. ∀t. A ⊢ t ⟶ A ∪ ik_est D ⊢c t)"

  obtain D where D: "D ∈ decomps_est A N 𝓘" "∀D' ∈ decomps_est A N 𝓘. ik_est D' ⊆ ik_est D"
    using decomps_est_ik_max_exist[OF assms] by moura
  then obtain t where t: "A ∪ ik_est D ⊢ t" "¬(A ∪ ik_est D ⊢c t)"
    using neg by (fastforce intro: ideduct_mono)

  obtain D' where D':
      "D@D' ∈ decomps_est A N 𝓘" "A ∪ ik_est (D@D') ⊢c t"
      "A ∪ ik_est D ⊂ A ∪ ik_est (D@D')"
    by (metis decomps_est_exist_aux t D(1))
  hence "ik_est D ⊂ ik_est (D@D')" using ik_est_append by auto
  moreover have "ik_est (D@D') ⊆ ik_est D" using D(2) D'(1) by auto
  ultimately show False by simp
qed

private lemma decomps_est_exist_subst:
  assumes "ik_est A ·set 𝓘 ⊢ t · 𝓘"
  and "sem_est_c {} 𝓘 A" "wf_est {} A" "interpretation_subst 𝓘"
  and "Ana_invar_subst (ik_est A ∪ assignment_rhs_est A)"
```

```
  and "well_analyzed A"
  shows "∃D ∈ decomps_est (ik_est A) (assignment_rhs_est A) ℐ. ik_est (A@D) ·_set ℐ ⊢_c t · ℐ"
proof -
  have ik_eq: "ik_est (A ·_est ℐ) = ik_est A ·_set ℐ" using assms(5,6)
  proof (induction A rule: List.rev_induct)
    case (snoc a A)
    hence "Ana_invar_subst (ik_est A ∪ assignment_rhs_est A)"
      using Ana_invar_subst_subset[OF snoc.prems(1)] ik_est_append assignment_rhs_est_append
      unfolding Ana_invar_subst_def by simp
    with snoc have IH:
        "ik_est (A@[a] ·_est ℐ) = (ik_est A ·_set ℐ) ∪ ik_est ([a] ·_est ℐ)"
        "ik_est (A@[a]) ·_set ℐ = (ik_est A ·_set ℐ) ∪ (ik_est [a] ·_set ℐ)"
      using well_analyzed_split_left[OF snoc.prems(2)]
      by (auto simp add: to_st_append ik_est_append_subst)

    have "ik_est [a ·_estp ℐ] = ik_est [a] ·_set ℐ"
    proof (cases a)
      case (Step b) thus ?thesis by (cases b) auto
    next
      case (Decomp t)
      then obtain f T where t: "t = Fun f T" using well_analyzedD[OF snoc.prems(2)] by force
      obtain K M where Ana_t: "Ana (Fun f T) = (K,M)" by (metis surj_pair)
      moreover have "Fun f T ∈ subterms_set ((ik_est (A@[a]) ∪ assignment_rhs_est (A@[a])))"
        using t Decomp snoc.prems(2)
        by (auto dest: well_analyzed_inv simp add: ik_est_append assignment_rhs_est_append)
      hence "Ana (Fun f T · ℐ) = (K ·_list ℐ, M ·_list ℐ)"
        using Ana_t snoc.prems(1)
        unfolding Ana_invar_subst_def by force
      ultimately show ?thesis using Decomp t by (auto simp add: decomp_ik)
    qed
    thus ?case using IH unfolding subst_apply_extstrand_def by simp
  qed simp
  moreover have assignment_rhs_eq: "assignment_rhs_est (A ·_est ℐ) = assignment_rhs_est A ·_set ℐ"
    using assms(5,6)
  proof (induction A rule: List.rev_induct)
    case (snoc a A)
    hence "Ana_invar_subst (ik_est A ∪ assignment_rhs_est A)"
      using Ana_invar_subst_subset[OF snoc.prems(1)] ik_est_append assignment_rhs_est_append
      unfolding Ana_invar_subst_def by simp
    hence "assignment_rhs_est (A ·_est ℐ) = assignment_rhs_est A ·_set ℐ"
      using snoc.IH well_analyzed_split_left[OF snoc.prems(2)]
      by simp
    hence IH:
        "assignment_rhs_est (A@[a] ·_est ℐ) = (assignment_rhs_est A ·_set ℐ) ∪ assignment_rhs_est ([a] ·_est
ℐ)"
        "assignment_rhs_est (A@[a]) ·_set ℐ = (assignment_rhs_est A ·_set ℐ) ∪ (assignment_rhs_est [a] ·_set
ℐ)"
      by (metis assignment_rhs_est_append_subst(1), metis assignment_rhs_est_append_subst(2))

    have "assignment_rhs_est [a ·_estp ℐ] = assignment_rhs_est [a] ·_set ℐ"
    proof (cases a)
      case (Step b) thus ?thesis by (cases b) auto
    next
      case (Decomp t)
      then obtain f T where t: "t = Fun f T" using well_analyzedD[OF snoc.prems(2)] by force
      obtain K M where Ana_t: "Ana (Fun f T) = (K,M)" by (metis surj_pair)
      moreover have "Fun f T ∈ subterms_set ((ik_est (A@[a]) ∪ assignment_rhs_est (A@[a])))"
        using t Decomp snoc.prems(2)
        by (auto dest: well_analyzed_inv simp add: ik_est_append assignment_rhs_est_append)
      hence "Ana (Fun f T · ℐ) = (K ·_list ℐ, M ·_list ℐ)"
        using Ana_t snoc.prems(1) unfolding Ana_invar_subst_def by force
      ultimately show ?thesis using Decomp t by (auto simp add: decomp_assignment_rhs_empty)
    qed
```

```
    thus ?case using IH unfolding subst_apply_extstrand_def by simp
  qed simp
  ultimately obtain D where D:
      "D ∈ decomps_est (ik_est A ·set ℐ) (assignment_rhs_est A ·set ℐ) Var"
      "(ik_est A ·set ℐ) ∪ (ik_est D) ⊢c t · ℐ"
    using decomps_est_exist[OF ik_est_finite assignment_rhs_est_finite, of "A ·est ℐ" "A ·est ℐ"]
          ik_est_append assignment_rhs_est_append assms(1)
    by force

  let ?P = "λD D'. ∀t. (ik_est A ·set ℐ) ∪ (ik_est D) ⊢c t ⟶ (ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c t"

  have "∃D' ∈ decomps_est (ik_est A) (assignment_rhs_est A) ℐ. ?P D D'" using D(1)
  proof (induction D rule: decomps_est.induct)
    case Nil
    have "ik_est [] = ik_est [] ·set ℐ" by auto
    thus ?case by (metis decomps_est.Nil)
  next
    case (Decomp D f T K M)
    obtain D' where D': "D' ∈ decomps_est (ik_est A) (assignment_rhs_est A) ℐ" "?P D D'"
      using Decomp.IH by auto
    hence IH: "⋀k. k ∈ set K ⟹ (ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c k"
              "(ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c Fun f T"
      using Decomp.hyps(5,6) by auto

    have D'_ik: "ik_est D' ·set ℐ ⊆ subterms_set ((ik_est A ∪ assignment_rhs_est A)) ·set ℐ"
                "ik_est D' ⊆ subterms_set (ik_est A ∪ assignment_rhs_est A)"
      using decomps_est_ik_subset[OF D'(1)] by (metis subst_all_mono, metis)

    show ?case using IH(2,1) Decomp.hyps(2,3,4)
    proof (induction "Fun f T" arbitrary: f T K M rule: intruder_synth_induct)
      case (AxiomC f T)
      then obtain s where s: "s ∈ ik_est A ∪ ik_est D'" "Fun f T = s · ℐ" using AxiomC.prems by blast
      hence fT_s_in: "Fun f T ∈ (subterms_set (ik_est A ∪ assignment_rhs_est A)) ·set ℐ"
                     "s ∈ subterms_set (ik_est A ∪ assignment_rhs_est A)"
        using AxiomC D'_ik subset_subterms_Union[of "ik_est A ∪ assignment_rhs_est A"]
              subst_all_mono[OF subset_subterms_Union, of ℐ]
        by (metis (no_types) Un_iff image_eqI subset_Un_eq, metis (no_types) Un_iff subset_Un_eq)
      obtain Ks Ms where Ana_s: "Ana s = (Ks,Ms)" by moura

      have AD'_props: "wf_est {} (A@D')" "⟦{}; to_st (A@D')⟧c ℐ"
        using decomps_est_preserves_model_c[OF D'(1) assms(2)]
              decomps_est_preserves_wf[OF D'(1) assms(3)]
              sem_est_c_eq_sem_st strand_sem_eq_defs(1)
        by auto

      show ?case
      proof (cases s)
        case (Var x)
```

— In this case ℐ x (is a subterm of something that) was derived from an "earlier intruder knowledge" because A is well-formed and has ℐ as a model. So either the intruder composed Fun f T himself (making Decomp (Fun f T) unnecessary) or Fun f T is an instance of something else in the intruder knowledge (in which case the "something" can be used in place of Fun f T)

```
        hence "Var x ∈ ik_est (A@D')" "ℐ x = Fun f T" using s ik_est_append by auto

        show ?thesis
        proof (cases "∀m ∈ set M. ik_est A ∪ ik_est D' ·set ℐ ⊢c m")
          case True
```

— All terms acquired by decomposing Fun f T are already derivable. Hence there is no need to consider decomposition of Fun f T at all.

```
          have *: "(ik_est A ·set ℐ) ∪ ik_est (D@[Decomp (Fun f T)]) = (ik_est A ·set ℐ) ∪ ik_est D ∪ set M"
            using decomp_ik[OF ‹Ana (Fun f T) = (K,M)›] ik_est_append[of D "[Decomp (Fun f T)]"]
            by auto
```

```
{ fix t' assume "(ik_est A ·_set ℐ) ∪ ik_est D ∪ set M ⊢_c t'"
  hence "(ik_est A ·_set ℐ) ∪ (ik_est D' ·_set ℐ) ⊢_c t'"
  proof (induction t' rule: intruder_synth_induct)
    case (AxiomC t') thus ?case
    proof
      assume "t' ∈ set M"
      moreover have "(ik_est A ·_set ℐ) ∪ (ik_est D' ·_set ℐ) = ik_est A ∪ ik_est D' ·_set ℐ" by
auto
      ultimately show ?case using True by auto
    qed (metis D'(2) intruder_synth.AxiomC)
  qed auto
}
thus ?thesis using D'(1) * by metis
next
  case False
  — Some term acquired by decomposition of Fun f T cannot be derived in ⊢_c. Fun f T must therefore be an
instance of something else in the intruder knowledge, because of well-formedness.
  then obtain t_i where t_i: "t_i ∈ set T" "¬ik_est (A@D') ·_set ℐ ⊢_c t_i"
    using Ana_fun_subterm[OF ‹Ana (Fun f T) = (K,M)›] by (auto simp add: ik_est_append)
  obtain S where fS:
      "Fun f S ∈ subterms_set (ik_est (A@D')) ∨
       Fun f S ∈ subterms_set (assignment_rhs_est (A@D'))"
      "ℐ x = Fun f S · ℐ"
    using strand_sem_wf_ik_or_assignment_rhs_fun_subterm[
            OF AD'_props ‹Var x ∈ ik_est (A@D')› _ t_i ‹interpretation_subst ℐ›]
          ‹ℐ x = Fun f T›
    by moura
  hence fS_in: "Fun f S · ℐ ∈ ik_est A ∪ ik_est D' ·_set ℐ"
               "Fun f S ∈ subterms_set (ik_est A ∪ assignment_rhs_est A)"
    using imageI[OF s(1), of "λx. x · ℐ"] Var
          ik_est_append[of A D'] assignment_rhs_est_append[of A D']
          decomps_est_subterms[OF D'(1)] decomps_est_assignment_rhs_empty[OF D'(1)]
    by auto
  obtain KS MS where Ana_fS: "Ana (Fun f S) = (KS, MS)" by moura
  hence "K = KS ·_list ℐ" "M = MS ·_list ℐ"
    using Ana_invar_substD[OF assms(5) fS_in(2)]
          s(2) fS(2) ‹s = Var x› ‹Ana (Fun f T) = (K,M)›
    by simp_all
  hence "MS ≠ []" using ‹M ≠ []› by simp
  have "⋀k. k ∈ set KS ⟹ ik_est A ∪ ik_est D' ·_set ℐ ⊢_c k · ℐ"
    using AxiomC.prems(1) ‹K = KS ·_list ℐ› by (simp add: image_Un)
  hence D'': "D'@[Decomp (Fun f S)] ∈ decomps_est (ik_est A) (assignment_rhs_est A) ℐ"
    using decomps_est.Decomp[OF D'(1) fS_in(2) Ana_fS ‹MS ≠ []›] AxiomC.prems(1)
          intruder_synth.AxiomC[OF fS_in(1)]
    by simp
  moreover {
    fix t' assume "(ik_est A ·_set ℐ) ∪ ik_est (D@[Decomp (Fun f T)]) ⊢_c t'"
    hence "(ik_est A ·_set ℐ) ∪ (ik_est (D'@[Decomp (Fun f S)]) ·_set ℐ) ⊢_c t'"
    proof (induction t' rule: intruder_synth_induct)
      case (AxiomC t')
      hence "t' ∈ (ik_est A ·_set ℐ) ∪ ik_est D ∨ t' ∈ ik_est [Decomp (Fun f T)]"
        by (simp add: ik_est_append)
      thus ?case
      proof
        assume "t' ∈ ik_est [Decomp (Fun f T)]"
        hence "t' ∈ ik_est [Decomp (Fun f S)] ·_set ℐ"
          using decomp_ik ‹Ana (Fun f T) = (K,M)› ‹Ana (Fun f S) = (KS,MS)› ‹M = MS ·_list ℐ›
          by simp
        thus ?case
          using ideduct_synth_mono[
                  OF intruder_synth.AxiomC[of t' "ik_est [Decomp (Fun f S)] ·_set ℐ"],
                  of "(ik_est A ·_set ℐ) ∪ (ik_est (D'@[Decomp (Fun f S)]) ·_set ℐ)"]
```

```
              by (auto simp add: ik_est_append)
          next
            assume "t' ∈ (ik_est A ·set ℐ) ∪ ik_est D"
            hence "(ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c t'"
              by (metis D'(2) intruder_synth.AxiomC)
            hence "(ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ∪ (ik_est [Decomp (Fun f S)] ·set ℐ) ⊢c t'"
              by (simp add: ideduct_synth_mono)
            thus ?case
              using ik_est_append[of D' "[Decomp (Fun f S)]"]
                    image_Un[of "λx. x · ℐ" "ik_est D'" "ik_est [Decomp (Fun f S)]"]
              by (simp add: sup_aci(2))
          qed
        qed auto
      }
      ultimately show ?thesis using D'' by auto
    qed
  next
    case (Fun g S) — Hence Decomp (Fun f T) can be substituted for Decomp (Fun g S)
    hence KM: "K = Ks ·list ℐ" "M = Ms ·list ℐ" "set K = set Ks ·set ℐ" "set M = set Ms ·set ℐ"
      using fT_s_in(2) ‹Ana (Fun f T) = (K,M)› Ana_s s(2)
            Ana_invar_substD[OF assms(5), of g S]
      by auto
    hence Ms_nonempty: "Ms ≠ []" using ‹M ≠ []› by auto
    { fix t' assume "(ik_est A ·set ℐ) ∪ ik_est (D@[Decomp (Fun f T)]) ⊢c t'"
      hence "(ik_est A ·set ℐ) ∪ (ik_est (D'@[Decomp (Fun g S)]) ·set ℐ) ⊢c t'" using AxiomC
      proof (induction t' rule: intruder_synth_induct)
        case (AxiomC t')
        hence "t' ∈ ik_est A ·set ℐ ∨ t' ∈ ik_est D ∨ t' ∈ set M"
          by (simp add: decomp_ik ik_est_append)
        thus ?case
        proof (elim disjE)
          assume "t' ∈ ik_est D"
          hence *: "(ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c t'" using D'(2) by simp
          show ?case by (auto intro: ideduct_synth_mono[OF *] simp add: ik_est_append_subst(2))
        next
          assume "t' ∈ set M"
          hence "t' ∈ ik_est [Decomp (Fun g S)] ·set ℐ"
            using KM(2) Fun decomp_ik[OF Ana_s] by auto
          thus ?case by (simp add: image_Un ik_est_append)
        qed (simp add: ideduct_synth_mono[OF intruder_synth.AxiomC])
      qed auto
    }
    thus ?thesis
      using s Fun Ana_s AxiomC.prems(1) KM(3) fT_s_in
            decomps_est.Decomp[OF D'(1) _ _ Ms_nonempty, of g S Ks]
      by (metis AxiomC.hyps image_Un image_eqI intruder_synth.AxiomC)
  qed
next
  case (ComposeC T f)
  have *: "⋀m. m ∈ set M ⟹ (ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c m"
    using Ana_fun_subterm[OF ‹Ana (Fun f T) = (K, M)›] ComposeC.hyps(3)
    by auto

  have **: "ik_est (D@[Decomp (Fun f T)]) = ik_est D ∪ set M"
    using decomp_ik[OF ‹Ana (Fun f T) = (K, M)›] ik_est_append by auto

  { fix t' assume "(ik_est A ·set ℐ) ∪ ik_est (D@[Decomp (Fun f T)]) ⊢c t'"
    hence "(ik_est A ·set ℐ) ∪ (ik_est D' ·set ℐ) ⊢c t'"
      by (induct rule: intruder_synth_induct) (auto simp add: D'(2) * **)
  }
  thus ?case using D'(1) by auto
 qed
qed
```

```
  thus ?thesis using D(2) assms(1) by (auto simp add: ik_est_append_subst(2))
qed

private lemma wf_sts'_update_st_nil: assumes "wf_sts' 𝒮 𝒜" shows "wf_sts' (update_st 𝒮 []) 𝒜"
using assms unfolding wf_sts'_def by auto

private lemma wf_sts'_update_st_snd:
  assumes "wf_sts' 𝒮 𝒜" "send⟨t⟩_st#S ∈ 𝒮"
  shows "wf_sts' (update_st 𝒮 (send⟨t⟩_st#S)) (𝒜@[Step (receive⟨t⟩_st)])"
unfolding wf_sts'_def
proof (intro conjI)
  let ?S = "send⟨t⟩_st#S"
  let ?A = "𝒜@[Step (receive⟨t⟩_st)]"

  have 𝒮: "⋀S'. S' ∈ update_st 𝒮 ?S ⟹ S' = S ∨ S' ∈ 𝒮" by auto

  have 1: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est 𝒜) (dual_st S)" using assms unfolding wf_sts'_def by auto
  moreover have 2: "wfrestrictedvars_est ?A = wfrestrictedvars_est 𝒜 ∪ fv t"
    using wfrestrictedvars_est_split(2) by (auto simp add: Un_assoc)
  ultimately have 3: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by (metis wf_vars_mono)

  have 4: "∀S ∈ 𝒮. ∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}" using assms unfolding wf_sts'_def by simp

  have "wf_st (wfrestrictedvars_est ?A) (dual_st S)" using 1 2 3 assms(2) by auto
  thus "∀S ∈ update_st 𝒮 ?S. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by (metis 3 𝒮)

  have "fv_st S ∩ bvars_st S = {}"
       "∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}"
       "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st S = {}"
    using 4 assms(2) unfolding wf_sts'_def by force+
  thus "∀S ∈ update_st 𝒮 ?S. ∀S' ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_st S' = {}" by (metis 4 𝒮)

  have "∀S' ∈ 𝒮. fv_st ?S ∩ bvars_st S' = {}" "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st ?S = {}"
    using assms unfolding wf_sts'_def by metis+
  hence 5: "fv_est ?A = fv_est 𝒜 ∪ fv t" "bvars_est ?A = bvars_est 𝒜" "∀S' ∈ 𝒮. fv t ∩ bvars_st S' = {}"
    using to_st_append by fastforce+

  have *: "∀S ∈ 𝒮. fv_st S ∩ bvars_est ?A = {}"
    using 5 assms(1) unfolding wf_sts'_def by fast
  hence "fv_st ?S ∩ bvars_est ?A = {}" using assms(2) by metis
  hence "fv_st S ∩ bvars_est ?A = {}" by auto
  thus "∀S ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_est ?A = {}" by (metis * 𝒮)

  have **: "∀S ∈ 𝒮. fv_est ?A ∩ bvars_st S = {}"
    using 5 assms(1) unfolding wf_sts'_def by fast
  hence "fv_est ?A ∩ bvars_st ?S = {}" using assms(2) by metis
  hence "fv_est ?A ∩ bvars_st S = {}" by fastforce
  thus "∀S ∈ update_st 𝒮 ?S. fv_est ?A ∩ bvars_st S = {}" by (metis ** 𝒮)
qed

private lemma wf_sts'_update_st_rcv:
  assumes "wf_sts' 𝒮 𝒜" "receive⟨t⟩_st#S ∈ 𝒮"
  shows "wf_sts' (update_st 𝒮 (receive⟨t⟩_st#S)) (𝒜@[Step (send⟨t⟩_st)])"
unfolding wf_sts'_def
proof (intro conjI)
  let ?S = "receive⟨t⟩_st#S"
  let ?A = "𝒜@[Step (send⟨t⟩_st)]"

  have 𝒮: "⋀S'. S' ∈ update_st 𝒮 ?S ⟹ S' = S ∨ S' ∈ 𝒮" by auto

  have 1: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est 𝒜) (dual_st S)" using assms unfolding wf_sts'_def by auto
  moreover have 2: "wfrestrictedvars_est ?A = wfrestrictedvars_est 𝒜 ∪ fv t"
    using wfrestrictedvars_est_split(2) by (auto simp add: Un_assoc)
```

```
  ultimately have 3: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by (metis wf_vars_mono)

  have 4: "∀S ∈ 𝒮. ∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}" using assms unfolding wf_sts'_def by simp

  have "wf_st (wfrestrictedvars_est ?A) (dual_st S)" using 1 2 3 assms(2) by auto
  thus "∀S ∈ update_st 𝒮 ?S. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by (metis 3 𝒮)

  have "fv_st S ∩ bvars_st S = {}"
       "∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}"
       "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st S = {}"
    using 4 assms(2) unfolding wf_sts'_def by force+
  thus "∀S ∈ update_st 𝒮 ?S. ∀S' ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_st S' = {}" by (metis 4 𝒮)

  have "∀S' ∈ 𝒮. fv_st ?S ∩ bvars_st S' = {}" "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st ?S = {}"
    using assms unfolding wf_sts'_def by metis+
  hence 5: "fv_est ?A = fv_est 𝒜 ∪ fv t" "bvars_est ?A = bvars_est 𝒜" "∀S' ∈ 𝒮. fv t ∩ bvars_st S' = {}"
    using to_st_append by fastforce+

  have *: "∀S ∈ 𝒮. fv_st S ∩ bvars_est ?A = {}"
    using 5 assms(1) unfolding wf_sts'_def by fast
  hence "fv_st ?S ∩ bvars_est ?A = {}" using assms(2) by metis
  hence "fv_st S ∩ bvars_est ?A = {}" by auto
  thus "∀S ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_est ?A = {}" by (metis * 𝒮)

  have **: "∀S ∈ 𝒮. fv_est ?A ∩ bvars_st S = {}"
    using 5 assms(1) unfolding wf_sts'_def by fast
  hence "fv_est ?A ∩ bvars_st ?S = {}" using assms(2) by metis
  hence "fv_est ?A ∩ bvars_st S = {}" by fastforce
  thus "∀S ∈ update_st 𝒮 ?S. fv_est ?A ∩ bvars_st S = {}" by (metis ** 𝒮)
qed

private lemma wf_sts'_update_st_eq:
  assumes "wf_sts' 𝒮 𝒜" "⟨a: t ≐ t'⟩_st#S ∈ 𝒮"
  shows "wf_sts' (update_st 𝒮 (⟨a: t ≐ t'⟩_st#S)) (𝒜@[Step (⟨a: t ≐ t'⟩_st)])"
unfolding wf_sts'_def
proof (intro conjI)
  let ?S = "⟨a: t ≐ t'⟩_st#S"
  let ?A = "𝒜@[Step (⟨a: t ≐ t'⟩_st)]"

  have 𝒮: "⋀S'. S' ∈ update_st 𝒮 ?S ⟹ S' = S ∨ S' ∈ 𝒮" by auto

  have 1: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est 𝒜) (dual_st S)" using assms unfolding wf_sts'_def by auto
  moreover have 2:
      "a = Assign ⟹ wfrestrictedvars_est ?A = wfrestrictedvars_est 𝒜 ∪ fv t ∪ fv t'"
      "a = Check ⟹ wfrestrictedvars_est ?A = wfrestrictedvars_est 𝒜"
    using wfrestrictedvars_est_split(2) by (auto simp add: Un_assoc)
  ultimately have 3: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est ?A) (dual_st S)"
    by (cases a) (metis wf_vars_mono, metis)

  have 4: "∀S ∈ 𝒮. ∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}" using assms unfolding wf_sts'_def by simp

  have "wf_st (wfrestrictedvars_est ?A) (dual_st S)" using 1 2 3 assms(2) by (cases a) auto
  thus "∀S ∈ update_st 𝒮 ?S. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by (metis 3 𝒮)

  have "fv_st S ∩ bvars_st S = {}"
       "∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}"
       "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st S = {}"
    using 4 assms(2) unfolding wf_sts'_def by force+
  thus "∀S ∈ update_st 𝒮 ?S. ∀S' ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_st S' = {}" by (metis 4 𝒮)

  have "∀S' ∈ 𝒮. fv_st ?S ∩ bvars_st S' = {}" "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st ?S = {}"
    using assms unfolding wf_sts'_def by metis+
  hence 5: "fv_est ?A = fv_est 𝒜 ∪ fv t ∪ fv t'" "bvars_est ?A = bvars_est 𝒜"
```

```
        "∀S' ∈ 𝒮. fv t ∩ bvars_st S' = {}" "∀S' ∈ 𝒮. fv t' ∩ bvars_st S' = {}"
    using to_st_append by fastforce+

  have *: "∀S ∈ 𝒮. fv_st S ∩ bvars_est ?A = {}"
    using 5 assms(1) unfolding wf_sts'_def by fast
  hence "fv_st ?S ∩ bvars_est ?A = {}" using assms(2) by metis
  hence "fv_st S ∩ bvars_est ?A = {}" by auto
  thus "∀S ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_est ?A = {}" by (metis * 𝒮)

  have **: "∀S ∈ 𝒮. fv_est ?A ∩ bvars_st S = {}"
    using 5 assms(1) unfolding wf_sts'_def by fast
  hence "fv_est ?A ∩ bvars_st ?S = {}" using assms(2) by metis
  hence "fv_est ?A ∩ bvars_st S = {}" by fastforce
  thus "∀S ∈ update_st 𝒮 ?S. fv_est ?A ∩ bvars_st S = {}" by (metis ** 𝒮)
qed

private lemma wf_sts'_update_st_ineq:
  assumes "wf_sts' 𝒮 𝒜" "∀X⟨∨≠: F⟩_st#S ∈ 𝒮"
  shows "wf_sts' (update_st 𝒮 (∀X⟨∨≠: F⟩_st#S)) (𝒜@[Step (∀X⟨∨≠: F⟩_st)])"
unfolding wf_sts'_def
proof (intro conjI)
  let ?S = "∀X⟨∨≠: F⟩_st#S"
  let ?A = "𝒜@[Step (∀X⟨∨≠: F⟩_st)]"

  have 𝒮: "⋀S'. S' ∈ update_st 𝒮 ?S ⟹ S' = S ∨ S' ∈ 𝒮" by auto

  have 1: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est 𝒜) (dual_st S)" using assms unfolding wf_sts'_def by auto
  moreover have 2: "wfrestrictedvars_est ?A = wfrestrictedvars_est 𝒜"
    using wfrestrictedvars_est_split(2) by (auto simp add: Un_assoc)
  ultimately have 3: "∀S ∈ 𝒮. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by metis

  have 4: "∀S ∈ 𝒮. ∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}" using assms unfolding wf_sts'_def by simp

  have "wf_st (wfrestrictedvars_est ?A) (dual_st S)" using 1 2 3 assms(2) by auto
  thus "∀S ∈ update_st 𝒮 ?S. wf_st (wfrestrictedvars_est ?A) (dual_st S)" by (metis 3 𝒮)

  have "fv_st S ∩ bvars_st S = {}"
       "∀S' ∈ 𝒮. fv_st S ∩ bvars_st S' = {}"
       "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st S = {}"
    using 4 assms(2) unfolding wf_sts'_def by force+
  thus "∀S ∈ update_st 𝒮 ?S. ∀S' ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_st S' = {}" by (metis 4 𝒮)

  have "∀S' ∈ 𝒮. fv_st ?S ∩ bvars_st S' = {}" "∀S' ∈ 𝒮. fv_st S' ∩ bvars_st ?S = {}"
    using assms unfolding wf_sts'_def by metis+
  moreover have "fv_pairs F - set X ⊆ fv_st (∀X⟨∨≠: F⟩_st # S)" by auto
  ultimately have 5:
      "∀S' ∈ 𝒮. (fv_pairs F - set X) ∩ bvars_st S' = {}"
      "fv_est ?A = fv_est 𝒜 ∪ (fv_pairs F - set X)" "bvars_est ?A = set X ∪ bvars_est 𝒜"
      "∀S ∈ 𝒮. fv_st S ∩ set X = {}"
    using to_st_append
    by (blast, force, force, force)

  have *: "∀S ∈ 𝒮. fv_st S ∩ bvars_est ?A = {}" using 5(3,4) assms(1) unfolding wf_sts'_def by blast
  hence "fv_st ?S ∩ bvars_est ?A = {}" using assms(2) by metis
  hence "fv_st S ∩ bvars_est ?A = {}" by auto
  thus "∀S ∈ update_st 𝒮 ?S. fv_st S ∩ bvars_est ?A = {}" by (metis * 𝒮)

  have **: "∀S ∈ 𝒮. fv_est ?A ∩ bvars_st S = {}"
    using 5(1,2) assms(1) unfolding wf_sts'_def by fast
  hence "fv_est ?A ∩ bvars_st ?S = {}" using assms(2) by metis
  hence "fv_est ?A ∩ bvars_st S = {}" by auto
  thus "∀S ∈ update_st 𝒮 ?S. fv_est ?A ∩ bvars_st S = {}" by (metis ** 𝒮)
qed
```

```
private lemma trms_st_update_st_eq:
  assumes "x#S ∈ 𝒮"
  shows "⋃(trms_st ` update_st 𝒮 (x#S)) ∪ trms_stp x = ⋃(trms_st ` 𝒮)" (is "?A = ?B")
proof
  show "?B ⊆ ?A"
  proof
    have "trms_stp x ⊆ trms_st (x#S)" by auto
    hence "⋀t'. t' ∈ ?B ⟹ t' ∈ trms_stp x ⟹ t' ∈ ?A" by simp
    moreover {
      fix t' assume t': "t' ∈ ?B" "t' ∉ trms_stp x"
      then obtain S' where S': "t' ∈ trms_st S'" "S' ∈ 𝒮" by auto
      hence "S' = x#S ∨ S' ∈ update_st 𝒮 (x#S)" by auto
      moreover {
        assume "S' = x#S"
        hence "t' ∈ trms_st S" using S' t' by simp
        hence "t' ∈ ?A" by auto
      }
      ultimately have "t' ∈ ?A" using t' S' by auto
    }
    ultimately show "⋀t'. t' ∈ ?B ⟹ t' ∈ ?A" by metis
  qed

  show "?A ⊆ ?B"
  proof
    have "⋀t'. t' ∈ ?A ⟹ t' ∈ trms_stp x ⟹ trms_stp x ⊆ ?B"
      using assms by force+
    moreover {
      fix t' assume t': "t' ∈ ?A" "t' ∉ trms_stp x"
      then obtain S' where "t' ∈ trms_st S'" "S' ∈ update_st 𝒮 (x#S)" by auto
      hence "S' = S ∨ S' ∈ 𝒮" by auto
      moreover have "trms_st S ⊆ ?B" using assms trms_st_cons[of x S] by blast
      ultimately have "t' ∈ ?B" using t' by fastforce
    }
    ultimately show "⋀t'. t' ∈ ?A ⟹ t' ∈ ?B" by blast
  qed
qed

private lemma trms_st_update_st_eq_snd:
  assumes "send⟨t⟩_st#S ∈ 𝒮" "𝒮' = update_st 𝒮 (send⟨t⟩_st#S)" "𝒜' = 𝒜@[Step (receive⟨t⟩_st)]"
  shows "(⋃(trms_st ` 𝒮)) ∪ (trms_est 𝒜) = (⋃(trms_st ` 𝒮')) ∪ (trms_est 𝒜')"
proof -
  have "(trms_est 𝒜') = (trms_est 𝒜) ∪ {t}" "⋃(trms_st ` 𝒮') ∪ {t} = ⋃(trms_st ` 𝒮)"
    using to_st_append trms_st_update_st_eq[OF assms(1)] assms(2,3) by auto
  thus ?thesis
    by (metis (no_types, lifting) Un_insert_left Un_insert_right sup_bot.right_neutral)
qed

private lemma trms_st_update_st_eq_rcv:
  assumes "receive⟨t⟩_st#S ∈ 𝒮" "𝒮' = update_st 𝒮 (receive⟨t⟩_st#S)" "𝒜' = 𝒜@[Step (send⟨t⟩_st)]"
  shows "(⋃(trms_st ` 𝒮)) ∪ (trms_est 𝒜) = (⋃(trms_st ` 𝒮')) ∪ (trms_est 𝒜')"
proof -
  have "(trms_est 𝒜') = (trms_est 𝒜) ∪ {t}" "⋃(trms_st ` 𝒮') ∪ {t} = ⋃(trms_st ` 𝒮)"
    using to_st_append trms_st_update_st_eq[OF assms(1)] assms(2,3) by auto
  thus ?thesis
    by (metis (no_types, lifting) Un_insert_left Un_insert_right sup_bot.right_neutral)
qed

private lemma trms_st_update_st_eq_eq:
  assumes "⟨a: t ≐ t'⟩_st#S ∈ 𝒮" "𝒮' = update_st 𝒮 (⟨a: t ≐ t'⟩_st#S)" "𝒜' = 𝒜@[Step (⟨a: t ≐
t'⟩_st)]"
  shows "(⋃(trms_st ` 𝒮)) ∪ (trms_est 𝒜) = (⋃(trms_st ` 𝒮')) ∪ (trms_est 𝒜')"
proof -
```

```
  have "(trms_est 𝒜') = (trms_est 𝒜) ∪ {t,t'}" "⋃(trms_st ` 𝒮') ∪ {t,t'} = ⋃(trms_st ` 𝒮)"
    using to_st_append trms_st_update_st_eq[OF assms(1)] assms(2,3) by auto
  thus ?thesis
    by (metis (no_types, lifting) Un_insert_left Un_insert_right sup_bot.right_neutral)
qed

private lemma trms_st_update_st_eq_ineq:
  assumes "∀X⟨∨≠: F⟩_st#S ∈ 𝒮" "𝒮' = update_st 𝒮 (∀X⟨∨≠: F⟩_st#S)" "𝒜' = 𝒜@[Step (∀X⟨∨≠: F⟩_st)]"
  shows "(⋃(trms_st ` 𝒮)) ∪ (trms_est 𝒜) = (⋃(trms_st ` 𝒮')) ∪ (trms_est 𝒜')"
proof -
  have "(trms_est 𝒜') = (trms_est 𝒜) ∪ trms_pairs F" "⋃(trms_st ` 𝒮') ∪ trms_pairs F = ⋃(trms_st ` 𝒮)"
    using to_st_append trms_st_update_st_eq[OF assms(1)] assms(2,3) by auto
  thus ?thesis by (simp add: Un_commute sup_left_commute)
qed

private lemma ik_st_update_st_subset:
  assumes "x#S ∈ 𝒮"
  shows "⋃(ik_st`dual_st ` (update_st 𝒮 (x#S))) ⊆ ⋃(ik_st`dual_st ` 𝒮)" (is ?A)
        "⋃(assignment_rhs_st ` (update_st 𝒮 (x#S))) ⊆ ⋃(assignment_rhs_st ` 𝒮)" (is ?B)
proof -
  { fix t assume "t ∈ ⋃(ik_st`dual_st ` (update_st 𝒮 (x#S)))"
    then obtain S' where S': "S' ∈ update_st 𝒮 (x#S)" "t ∈ ik_st (dual_st S')" by auto

    have *: "ik_st (dual_st S) ⊆ ik_st (dual_st (x#S))"
      using ik_append[of "dual_st [x]" "dual_st S"] dual_st_append[of "[x]" S]
      by auto

    hence "t ∈ ⋃(ik_st`dual_st ` 𝒮)"
    proof (cases "S' = S")
      case True thus ?thesis using * assms S' by auto
    next
      case False thus ?thesis using S' by auto
    qed
  }
  moreover
  { fix t assume "t ∈ ⋃(assignment_rhs_st ` (update_st 𝒮 (x#S)))"
    then obtain S' where S': "S' ∈ update_st 𝒮 (x#S)" "t ∈ assignment_rhs_st S'" by auto

    have "assignment_rhs_st S ⊆ assignment_rhs_st (x#S)"
      using assignment_rhs_append[of "[x]" S] by simp
    hence "t ∈ ⋃(assignment_rhs_st ` 𝒮)"
      using assms S' by (cases "S' = S") auto
  }
  ultimately show ?A ?B by (metis subsetI)+
qed

private lemma ik_st_update_st_subset_snd:
  assumes "send⟨t⟩_st#S ∈ 𝒮"
          "𝒮' = update_st 𝒮 (send⟨t⟩_st#S)"
          "𝒜' = 𝒜@[Step (receive⟨t⟩_st)]"
  shows "(⋃(ik_st ` dual_st ` 𝒮')) ∪ (ik_est 𝒜') ⊆
         (⋃(ik_st ` dual_st ` 𝒮)) ∪ (ik_est 𝒜)" (is ?A)
        "(⋃(assignment_rhs_st ` 𝒮')) ∪ (assignment_rhs_est 𝒜') ⊆
         (⋃(assignment_rhs_st ` 𝒮)) ∪ (assignment_rhs_est 𝒜)" (is ?B)
proof -
  { fix t' assume t'_in: "t' ∈ (⋃(ik_st`dual_st ` 𝒮')) ∪ (ik_est 𝒜')"
    hence "t' ∈ (⋃(ik_st`dual_st ` 𝒮')) ∪ (ik_est 𝒜) ∪ {t}" using assms ik_est_append by auto
    moreover have "t ∈ ⋃(ik_st`dual_st ` 𝒮)" using assms(1) by force
    ultimately have "t' ∈ (⋃(ik_st`dual_st ` 𝒮)) ∪ (ik_est 𝒜)"
      using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  }
  moreover
  { fix t' assume t'_in: "t' ∈ (⋃(assignment_rhs_st ` 𝒮')) ∪ (assignment_rhs_est 𝒜')"
```

```
    hence "t' ∈ (⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A)"
      using assms assignment_rhs_est_append by auto
    hence "t' ∈ (⋃(assignment_rhs_st ` S)) ∪ (assignment_rhs_est A)"
      using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  }
  ultimately show ?A ?B by (metis subsetI)+
qed

private lemma ik_st_update_st_subset_rcv:
  assumes "receive⟨t⟩_st#S ∈ S"
          "S' = update_st S (receive⟨t⟩_st#S)"
          "A' = A@[Step (send⟨t⟩_st)]"
  shows "(⋃(ik_st ` dual_st ` S')) ∪ (ik_est A') ⊆
         (⋃(ik_st ` dual_st ` S)) ∪ (ik_est A)" (is ?A)
        "(⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A') ⊆
         (⋃(assignment_rhs_st ` S)) ∪ (assignment_rhs_est A)" (is ?B)
proof -
  { fix t' assume t'_in: "t' ∈ (⋃(ik_st`dual_st ` S')) ∪ (ik_est A')"
    hence "t' ∈ (⋃(ik_st`dual_st ` S')) ∪ (ik_est A)" using assms ik_est_append by auto
    hence "t' ∈ (⋃(ik_st`dual_st ` S)) ∪ (ik_est A)"
      using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  }
  moreover
  { fix t' assume t'_in: "t' ∈ (⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A')"
    hence "t' ∈ (⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A)"
      using assms assignment_rhs_est_append by auto
    hence "t' ∈ (⋃(assignment_rhs_st ` S)) ∪ (assignment_rhs_est A)"
      using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  }
  ultimately show ?A ?B by (metis subsetI)+
qed

private lemma ik_st_update_st_subset_eq:
  assumes "⟨a: t ≐ t'⟩_st#S ∈ S"
          "S' = update_st S (⟨a: t ≐ t'⟩_st#S)"
          "A' = A@[Step (⟨a: t ≐ t'⟩_st)]"
  shows "(⋃(ik_st ` dual_st ` S')) ∪ (ik_est A') ⊆
         (⋃(ik_st ` dual_st ` S)) ∪ (ik_est A)" (is ?A)
        "(⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A') ⊆
         (⋃(assignment_rhs_st ` S)) ∪ (assignment_rhs_est A)" (is ?B)
proof -
  have 1: "t' ∈ (⋃(ik_st`dual_st ` S)) ∪ (ik_est A)"
    when "t' ∈ (⋃(ik_st`dual_st ` S')) ∪ (ik_est A')"
    for t'
  proof -
    have "t' ∈ (⋃(ik_st`dual_st ` S')) ∪ (ik_est A)" using that assms ik_est_append by auto
    thus ?thesis using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  qed

  have 2: "t'' ∈ (⋃(assignment_rhs_st ` S)) ∪ (assignment_rhs_est A)"
    when "t'' ∈ (⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A')" "a = Assign"
    for t''
  proof -
    have "t'' ∈ (⋃(assignment_rhs_st ` S')) ∪ (assignment_rhs_est A) ∪ {t'}"
      using that assms assignment_rhs_est_append by auto
    moreover have "t' ∈ ⋃(assignment_rhs_st ` S)" using assms(1) that by force
    ultimately show ?thesis using ik_st_update_st_subset[OF assms(1)] assms(2) that by auto
  qed

  have 3: "assignment_rhs_est A' = assignment_rhs_est A" (is ?C)
          "(⋃(assignment_rhs_st ` S')) ⊆ (⋃(assignment_rhs_st ` S))" (is ?D)
    when "a = Check"
  proof -
```

```
    show ?C using that assms(2,3) by (simp add: assignment_rhs_est_append)
    show ?D using assms(1,2,3) ik_st_update_st_subset(2) by auto
  qed

  show ?A using 1 2 by (metis subsetI)
  show ?B using 1 2 3 by (cases a) blast+
qed

private lemma ik_st_update_st_subset_ineq:
  assumes "∀X⟨∨≠: F⟩_st#S ∈ 𝒮"
          "𝒮' = update_st 𝒮 (∀X⟨∨≠: F⟩_st#S)"
          "𝒜' = 𝒜@[Step (∀X⟨∨≠: F⟩_st)]"
  shows "(⋃(ik_st‘dual_st ‘ 𝒮')) ∪ (ik_est 𝒜') ⊆
          (⋃(ik_st‘dual_st ‘ 𝒮)) ∪ (ik_est 𝒜)" (is ?A)
        "(⋃(assignment_rhs_st ‘ 𝒮')) ∪ (assignment_rhs_est 𝒜') ⊆
         (⋃(assignment_rhs_st ‘ 𝒮)) ∪ (assignment_rhs_est 𝒜)" (is ?B)
proof -
  { fix t' assume t'_in: "t' ∈ (⋃(ik_st‘dual_st ‘ 𝒮')) ∪ (ik_est 𝒜')"
    hence "t' ∈ (⋃(ik_st‘dual_st ‘ 𝒮')) ∪ (ik_est 𝒜)" using assms ik_est_append by auto
    hence "t' ∈ (⋃(ik_st‘dual_st ‘ 𝒮)) ∪ (ik_est 𝒜)"
      using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  }
  moreover
  { fix t' assume t'_in: "t' ∈ (⋃(assignment_rhs_st ‘ 𝒮')) ∪ (assignment_rhs_est 𝒜')"
    hence "t' ∈ (⋃(assignment_rhs_st ‘ 𝒮')) ∪ (assignment_rhs_est 𝒜)"
      using assms assignment_rhs_est_append by auto
    hence "t' ∈ (⋃(assignment_rhs_st ‘ 𝒮)) ∪ (assignment_rhs_est 𝒜)"
      using ik_st_update_st_subset[OF assms(1)] assms(2) by auto
  }
  ultimately show ?A ?B by (metis subsetI)+
qed
```

Transition Systems Definitions

```
inductive pts_symbolic::
  "(('fun,'var) strands × ('fun,'var) strand) ⇒
   (('fun,'var) strands × ('fun,'var) strand) ⇒ bool"
(infix "⇒•" 50) where
  Nil[simp]:        "[] ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒• (update_st 𝒮 [],𝒜)"
| Send[simp]:       "send⟨t⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒• (update_st 𝒮 (send⟨t⟩_st#S),𝒜@[receive⟨t⟩_st])"
| Receive[simp]:    "receive⟨t⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒• (update_st 𝒮 (receive⟨t⟩_st#S),𝒜@[send⟨t⟩_st])"
| Equality[simp]:   "⟨a: t ≐ t'⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒• (update_st 𝒮 (⟨a: t ≐ t'⟩_st#S),𝒜@[⟨a: t ≐
t'⟩_st])"
| Inequality[simp]: "∀X⟨∨≠: F⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒• (update_st 𝒮 (∀X⟨∨≠: F⟩_st#S),𝒜@[∀X⟨∨≠:
F⟩_st])"

private inductive pts_symbolic_c::
  "(('fun,'var) strands × ('fun,'var) extstrand) ⇒
   (('fun,'var) strands × ('fun,'var) extstrand) ⇒ bool"
(infix "⇒•_c" 50) where
  Nil[simp]:        "[] ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒•_c (update_st 𝒮 [],𝒜)"
| Send[simp]:       "send⟨t⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒•_c (update_st 𝒮 (send⟨t⟩_st#S),𝒜@[Step
(receive⟨t⟩_st)])"
| Receive[simp]:    "receive⟨t⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒•_c (update_st 𝒮 (receive⟨t⟩_st#S),𝒜@[Step
(send⟨t⟩_st)])"
| Equality[simp]:   "⟨a: t ≐ t'⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒•_c (update_st 𝒮 (⟨a: t ≐ t'⟩_st#S),𝒜@[Step (⟨a:
t ≐ t'⟩_st)])"
| Inequality[simp]: "∀X⟨∨≠: F⟩_st#S ∈ 𝒮 ⟹ (𝒮,𝒜) ⇒•_c (update_st 𝒮 (∀X⟨∨≠: F⟩_st#S),𝒜@[Step
(∀X⟨∨≠: F⟩_st)])"
| Decompose[simp]:  "Fun f T ∈ subterms_set (ik_est 𝒜 ∪ assignment_rhs_est 𝒜)
                     ⟹ (𝒮,𝒜) ⇒•_c (𝒮,𝒜@[Decomp (Fun f T)])"

abbreviation pts_symbolic_rtrancl (infix "⇒•*" 50) where "a ⇒•* b ≡ pts_symbolic** a b"
```

```
private abbreviation pts_symbolic_c_rtrancl (infix "⇒•c*" 50) where "a ⇒•c* b ≡ pts_symbolic_c** a
b"

lemma pts_symbolic_induct[consumes 1, case_names Nil Send Receive Equality Inequality]:
  assumes "(𝒮,𝒜) ⇒• (𝒮',𝒜')"
  and "⟦[] ∈ 𝒮; 𝒮' = update_st 𝒮 []; 𝒜' = 𝒜⟧ ⟹ P"
  and "⋀t S. ⟦send⟨t⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (send⟨t⟩_st#S); 𝒜' = 𝒜@[receive⟨t⟩_st]⟧ ⟹ P"
  and "⋀t S. ⟦receive⟨t⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (receive⟨t⟩_st#S); 𝒜' = 𝒜@[send⟨t⟩_st]⟧ ⟹ P"
  and "⋀a t t' S. ⟦⟨a: t ≐ t'⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (⟨a: t ≐ t'⟩_st#S); 𝒜' = 𝒜@[⟨a: t ≐
t'⟩_st]⟧ ⟹ P"
  and "⋀X F S. ⟦∀X⟨∨≠: F⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (∀X⟨∨≠: F⟩_st#S); 𝒜' = 𝒜@[∀X⟨∨≠: F⟩_st]⟧ ⟹
P"
  shows "P"
apply (rule pts_symbolic.cases[OF assms(1)])
using assms(2,3,4,5,6) by simp_all

private lemma pts_symbolic_c_induct[consumes 1, case_names Nil Send Receive Equality Inequality
Decompose]:
  assumes "(𝒮,𝒜) ⇒•c (𝒮',𝒜')"
  and "⟦[] ∈ 𝒮; 𝒮' = update_st 𝒮 []; 𝒜' = 𝒜⟧ ⟹ P"
  and "⋀t S. ⟦send⟨t⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (send⟨t⟩_st#S); 𝒜' = 𝒜@[Step (receive⟨t⟩_st)]⟧ ⟹ P"
  and "⋀t S. ⟦receive⟨t⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (receive⟨t⟩_st#S); 𝒜' = 𝒜@[Step (send⟨t⟩_st)]⟧ ⟹
P"
  and "⋀a t t' S. ⟦⟨a: t ≐ t'⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (⟨a: t ≐ t'⟩_st#S); 𝒜' = 𝒜@[Step (⟨a: t ≐
t'⟩_st)]⟧ ⟹ P"
  and "⋀X F S. ⟦∀X⟨∨≠: F⟩_st#S ∈ 𝒮; 𝒮' = update_st 𝒮 (∀X⟨∨≠: F⟩_st#S); 𝒜' = 𝒜@[Step (∀X⟨∨≠:
F⟩_st)]⟧ ⟹ P"
  and "⋀f T. ⟦Fun f T ∈ subterms_set (ik_est 𝒜 ∪ assignment_rhs_est 𝒜); 𝒮' = 𝒮; 𝒜' = 𝒜@[Decomp (Fun
f T)]⟧ ⟹ P"
  shows "P"
apply (rule pts_symbolic_c.cases[OF assms(1)])
using assms(2,3,4,5,6,7) by simp_all

private lemma pts_symbolic_c_preserves_wf_prot:
  assumes "(𝒮,𝒜) ⇒•c* (𝒮',𝒜')" "wf_sts' 𝒮 𝒜"
  shows "wf_sts' 𝒮' 𝒜'"
using assms
proof (induction rule: rtranclp_induct2)
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  from step.hyps(2) step.IH[OF step.prems] show ?case
  proof (induction rule: pts_symbolic_c_induct)
    case Decompose
    hence "fv_est 𝒜2 = fv_est 𝒜1" "bvars_est 𝒜2 = bvars_est 𝒜1"
      using bvars_decomp ik_assignment_rhs_decomp_fv by metis+
    thus ?case using Decompose unfolding wf_sts'_def
      by (metis wf_vars_mono wfrestrictedvars_est_split(2))
  qed (metis wf_sts'_update_st_nil, metis wf_sts'_update_st_snd,
       metis wf_sts'_update_st_rcv, metis wf_sts'_update_st_eq,
       metis wf_sts'_update_st_ineq)
qed metis

private lemma pts_symbolic_c_preserves_wf_is:
  assumes "(𝒮,𝒜) ⇒•c* (𝒮',𝒜')" "wf_sts' 𝒮 𝒜" "wf_st V (to_st 𝒜)"
  shows "wf_st V (to_st 𝒜')"
using assms
proof (induction rule: rtranclp_induct2)
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  hence "(𝒮, 𝒜) ⇒•c* (𝒮2, 𝒜2)" by auto
  hence *: "wf_sts' 𝒮1 𝒜1" "wf_sts' 𝒮2 𝒜2"
    using pts_symbolic_c_preserves_wf_prot[OF _ step.prems(1)] step.hyps(1)
    by auto

  from step.hyps(2) step.IH[OF step.prems] show ?case
```

```
  proof (induction rule: pts_symbolic_c_induct)
    case Nil thus ?case by auto
  next
    case (Send t S)
    hence "wf_st (wfrestrictedvars_est 𝒜1) (receive⟨t⟩_st#(dual_st S))"
      using *(1) unfolding wf_sts'_def by fastforce
    hence "fv t ⊆ wfrestrictedvars_st (to_st 𝒜1) ∪ V"
      using wfrestrictedvars_est_eq_wfrestrictedvars_st by auto
    thus ?case using Send wf_rcv_append''' to_st_append by simp
  next
    case (Receive t) thus ?case using wf_snd_append to_st_append by simp
  next
    case (Equality a t t' S)
    hence "wf_st (wfrestrictedvars_est 𝒜1) (⟨a: t ≐ t'⟩_st#(dual_st S))"
      using *(1) unfolding wf_sts'_def by fastforce
    hence "fv t' ⊆ wfrestrictedvars_st (to_st 𝒜1) ∪ V" when "a = Assign"
      using wfrestrictedvars_est_eq_wfrestrictedvars_st that by auto
    thus ?case using Equality wf_eq_append''' to_st_append by (cases a) auto
  next
    case (Inequality t t' S) thus ?case using wf_ineq_append'' to_st_append by simp
  next
    case (Decompose f T)
    hence "fv (Fun f T) ⊆ wfrestrictedvars_est 𝒜1"
      by (metis fv_subterms_set fv_subset subset_trans
                ik_st_assignment_rhs_st_wfrestrictedvars_subset)
    hence "vars_st (decomp (Fun f T)) ⊆ wfrestrictedvars_st (to_st 𝒜1) ∪ V"
      using decomp_vars[of "Fun f T"] wfrestrictedvars_est_eq_wfrestrictedvars_st[of 𝒜1] by auto
    thus ?case
      using to_st_append[of 𝒜1 "[Decomp (Fun f T)]"]
            wf_append_suffix[OF Decompose.prems] Decompose.hyps(3)
      by (metis append_Nil2 decomp_vars(1,2) to_st.simps(1,3))
  qed
qed metis

private lemma pts_symbolic_c_preserves_tfr_set:
  assumes "(𝒮,𝒜) ⇒•_c* (𝒮',𝒜')"
    and "tfr_set ((⋃(trms_st ` 𝒮)) ∪ (trms_est 𝒜))"
    and "wf_trms ((⋃(trms_st ` 𝒮)) ∪ (trms_est 𝒜))"
  shows "tfr_set ((⋃(trms_st ` 𝒮')) ∪ (trms_est 𝒜')) ∧ wf_trms ((⋃(trms_st ` 𝒮')) ∪ (trms_est 𝒜'))"
using assms
proof (induction rule: rtranclp_induct2)
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  from step.hyps(2) step.IH[OF step.prems] show ?case
  proof (induction rule: pts_symbolic_c_induct)
    case Nil
    hence "⋃(trms_st ` 𝒮1) = ⋃(trms_st ` 𝒮2)" by force
    thus ?case using Nil by metis
  next
    case (Decompose f T)
    obtain t where t: "t ∈ ik_est 𝒜1 ∪ assignment_rhs_est 𝒜1" "Fun f T ⊑ t"
      using Decompose.hyps(1) by auto
    have t_wf: "wf_trm t"
      using Decompose.prems wf_trm_subterm[of _ t]
            trms_est_ik_assignment_rhsI[OF t(1)]
      unfolding tfr_set_def
      by (metis UN_E Un_iff)
    have "t ∈ subterms_set (trms_est 𝒜1)" using trms_est_ik_assignment_rhsI t by auto
    hence "Fun f T ∈ SMP (trms_est 𝒜1)"
      by (metis (no_types) SMP.MP SMP.Subterm UN_E t(2))
    hence "{Fun f T} ⊆ SMP (trms_est 𝒜1)" using SMP.Subterm[of "Fun f T"] by auto
    moreover have "trms_est 𝒜2 = insert (Fun f T) (trms_est 𝒜1)"
      using Decompose.hyps(3) by auto
    ultimately have *: "SMP (trms_est 𝒜1) = SMP (trms_est 𝒜2)"
```

```
      using SMP_subset_union_eq[of "{Fun f T}"]
      by (simp add: Un_commute)
    hence "SMP ((⋃(trms_st ` S1)) ∪ (trms_est A1)) = SMP ((⋃(trms_st ` S2)) ∪ (trms_est A2))"
      using Decompose.hyps(2) SMP_union by auto
    moreover have "∀t ∈ trms_est A1. wf_trm t" "wf_trm (Fun f T)"
      using Decompose.prems wf_trm_subterm t(2) t_wf unfolding tfr_set_def by auto
    hence "∀t ∈ trms_est A2. wf_trm t" by (metis * SMP.MP SMP_wf_trm)
    hence "∀t ∈ (⋃(trms_st ` S2)) ∪ (trms_est A2). wf_trm t"
      using Decompose.prems Decompose.hyps(2) unfolding tfr_set_def by force
    ultimately show ?thesis using Decompose.prems unfolding tfr_set_def by presburger
  qed (metis trms_st_update_st_eq_snd, metis trms_st_update_st_eq_rcv,
       metis trms_st_update_st_eq_eq, metis trms_st_update_st_eq_ineq)
qed metis

private lemma pts_symbolic_c_preserves_tfr_stp:
  assumes "(S,A) ⇒•_c* (S',A')" "∀S ∈ S ∪ {to_st A}. list_all tfr_stp S"
  shows "∀S ∈ S' ∪ {to_st A'}. list_all tfr_stp S"
using assms
proof (induction rule: rtranclp_induct2)
  case (step S1 A1 S2 A2)
  from step.hyps(2) step.IH[OF step.prems] show ?case
  proof (induction rule: pts_symbolic_c_induct)
    case Nil
    have 1: "∀S ∈ {to_st A2}. list_all tfr_stp S" using Nil by simp
    have 2: "S2 = S1 - {[]}" "∀S ∈ S1. list_all tfr_stp S"  using Nil by simp_all
    have "∀S ∈ S2. list_all tfr_stp S"
    proof
      fix S assume "S ∈ S2"
      hence "S ∈ S1" using 2(1) by simp
      thus "list_all tfr_stp S" using 2(2) by simp
    qed
    thus ?case using 1 by auto
  next
    case (Send t S)
    have 1: "∀S ∈ {to_st A2}. list_all tfr_stp S" using Send by (simp add: to_st_append)
    have 2: "S2 = insert S (S1 - {send⟨t⟩_st#S})" "∀S ∈ S1. list_all tfr_stp S"  using Send by
simp_all
    have 3: "∀S ∈ S2. list_all tfr_stp S"
    proof
      fix S' assume "S' ∈ S2"
      hence "S' ∈ S1 ∨ S' = S" using 2(1) by auto
      moreover have "list_all tfr_stp S" using Send.hyps 2(2) by auto
      ultimately show "list_all tfr_stp S'" using 2(2) by blast
    qed
    thus ?case using 1 by auto
  next
    case (Receive t S)
    have 1: "∀S ∈ {to_st A2}. list_all tfr_stp S" using Receive by (simp add: to_st_append)
    have 2: "S2 = insert S (S1 - {receive⟨t⟩_st#S})" "∀S ∈ S1. list_all tfr_stp S"
      using Receive by simp_all
    have 3: "∀S ∈ S2. list_all tfr_stp S"
    proof
      fix S' assume "S' ∈ S2"
      hence "S' ∈ S1 ∨ S' = S" using 2(1) by auto
      moreover have "list_all tfr_stp S" using Receive.hyps 2(2) by auto
      ultimately show "list_all tfr_stp S'" using 2(2) by blast
    qed
    show ?case using 1 3 by auto
  next
    case (Equality a t t' S)
    have 1: "to_st A2 = to_st A1@[⟨a: t ≐ t'⟩_st]" "list_all tfr_stp (to_st A1)"
      using Equality by (simp_all add: to_st_append)
    have 2: "list_all tfr_stp [⟨a: t ≐ t'⟩_st]" using Equality by fastforce
```

```
    have 3: "list_all tfr_stp (to_st 𝒜2)"
      using tfr_stp_all_append[of "to_st 𝒜1" "[⟨a: t ≐ t'⟩_st]"] 1 2 by metis
    hence 4: "∀S ∈ {to_st 𝒜2}. list_all tfr_stp S" using Equality by simp
    have 5: "𝒮2 = insert S (𝒮1 - {⟨a: t ≐ t'⟩_st#S})" "∀S ∈ 𝒮1. list_all tfr_stp S"
      using Equality by simp_all
    have 6: "∀S ∈ 𝒮2. list_all tfr_stp S"
    proof
      fix S' assume "S' ∈ 𝒮2"
      hence "S' ∈ 𝒮1 ∨ S' = S" using 5(1) by auto
      moreover have "list_all tfr_stp S" using Equality.hyps 5(2) by auto
      ultimately show "list_all tfr_stp S'" using 5(2) by blast
    qed
    thus ?case using 4 by auto
  next
    case (Inequality X F S)
    have 1: "to_st 𝒜2 = to_st 𝒜1@[∀X⟨∨≠: F⟩_st]" "list_all tfr_stp (to_st 𝒜1)"
      using Inequality by (simp_all add: to_st_append)
    have "list_all tfr_stp (∀X⟨∨≠: F⟩_st#S)" using Inequality(1,4) by blast
    hence 2: "list_all tfr_stp [∀X⟨∨≠: F⟩_st]" by simp
    have 3: "list_all tfr_stp (to_st 𝒜2)"
      using tfr_stp_all_append[of "to_st 𝒜1" "[∀X⟨∨≠: F⟩_st]"] 1 2 by metis
    hence 4: "∀S ∈ {to_st 𝒜2}. list_all tfr_stp S" using Inequality by simp
    have 5: "𝒮2 = insert S (𝒮1 - {∀X⟨∨≠: F⟩_st#S})" "∀S ∈ 𝒮1. list_all tfr_stp S"
      using Inequality by simp_all
    have 6: "∀S ∈ 𝒮2. list_all tfr_stp S"
    proof
      fix S' assume "S' ∈ 𝒮2"
      hence "S' ∈ 𝒮1 ∨ S' = S" using 5(1) by auto
      moreover have "list_all tfr_stp S" using Inequality.hyps 5(2) by auto
      ultimately show "list_all tfr_stp S'" using 5(2) by blast
    qed
    thus ?case using 4 by auto
  next
    case (Decompose f T)
    hence 1: "∀S ∈ 𝒮2. list_all tfr_stp S" by blast
    have 2: "list_all tfr_stp (to_st 𝒜1)" "list_all tfr_stp (to_st [Decomp (Fun f T)])"
      using Decompose.prems decomp_tfr_stp by auto
    hence "list_all tfr_stp (to_st 𝒜1@to_st [Decomp (Fun f T)])" by auto
    hence "list_all tfr_stp (to_st 𝒜2)"
      using Decompose.hyps(3) to_st_append[of 𝒜1 "[Decomp (Fun f T)]"]
      by auto
    thus ?case using 1 by blast
  qed
qed

private lemma pts_symbolic_c_preserves_well_analyzed:
  assumes "(𝒮,𝒜) ⇒•c* (𝒮',𝒜')" "well_analyzed 𝒜"
  shows "well_analyzed 𝒜'"
using assms
proof (induction rule: rtranclp_induct2)
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  from step.hyps(2) step.IH[OF step.prems] show ?case
  proof (induction rule: pts_symbolic_c_induct)
    case Receive thus ?case by (metis well_analyzed_singleton(1) well_analyzed_append)
  next
    case Send thus ?case by (metis well_analyzed_singleton(2) well_analyzed_append)
  next
    case Equality thus ?case by (metis well_analyzed_singleton(3) well_analyzed_append)
  next
    case Inequality thus ?case by (metis well_analyzed_singleton(4) well_analyzed_append)
  next
    case (Decompose f T)
    hence "Fun f T ∈ subterms_set (ik_est 𝒜1 ∪ assignment_rhs_est 𝒜1) - (Var`𝒱)" by auto
```

```
    thus ?case by (metis well_analyzed.Decomp Decompose.prems Decompose.hyps(3))
  qed simp
qed metis

private lemma pts_symbolic_c_preserves_Ana_invar_subst:
  assumes "(𝒮,𝒜) ⇒•c* (𝒮',𝒜')"
    and "Ana_invar_subst (
          (⋃(ikst ` dualst ` 𝒮) ∪ (ikest 𝒜)) ∪
          (⋃(assignment_rhsst ` 𝒮) ∪ (assignment_rhsest 𝒜)))"
  shows "Ana_invar_subst (
          (⋃(ikst ` dualst ` 𝒮') ∪ (ikest 𝒜')) ∪
          (⋃(assignment_rhsst ` 𝒮') ∪ (assignment_rhsest 𝒜')))"
using assms
proof (induction rule: rtranclp_induct2)
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  from step.hyps(2) step.IH[OF step.prems] show ?case
  proof (induction rule: pts_symbolic_c_induct)
    case Nil
    hence "⋃(ikst ` dualst ` 𝒮1) = ⋃(ikst ` dualst ` 𝒮2)"
          "⋃(assignment_rhsst ` 𝒮1) = ⋃(assignment_rhsst ` 𝒮2)"
      by force+
    thus ?case using Nil by metis
  next
    case Send show ?case
      using ikst_updatest_subset_snd[OF Send.hyps]
            Ana_invar_subst_subset[OF Send.prems]
      by (metis Un_mono)
  next
    case Receive show ?case
      using ikst_updatest_subset_rcv[OF Receive.hyps]
            Ana_invar_subst_subset[OF Receive.prems]
      by (metis Un_mono)
  next
    case Equality show ?case
      using ikst_updatest_subset_eq[OF Equality.hyps]
            Ana_invar_subst_subset[OF Equality.prems]
      by (metis Un_mono)
  next
    case Inequality show ?case
      using ikst_updatest_subset_ineq[OF Inequality.hyps]
            Ana_invar_subst_subset[OF Inequality.prems]
      by (metis Un_mono)
  next
    case (Decompose f T)
    let ?X = "⋃(assignment_rhsst`𝒮2) ∪ assignment_rhsest 𝒜2"
    let ?Y = "⋃(assignment_rhsst`𝒮1) ∪ assignment_rhsest 𝒜1"
    obtain K M where Ana: "Ana (Fun f T) = (K,M)" by moura
    hence *: "ikest 𝒜2 = ikest 𝒜1 ∪ set M" "assignment_rhsest 𝒜2 = assignment_rhsest 𝒜1"
      using ikest_append assignment_rhsest_append decomp_ik
            decomp_assignment_rhs_empty Decompose.hyps(3)
      by auto
    { fix g S assume "Fun g S ∈ subtermsset (⋃(ikst`dualst`𝒮2) ∪ ikest 𝒜2 ∪ ?X)"
      hence "Fun g S ∈ subtermsset (⋃(ikst`dualst ` 𝒮1) ∪ ikest 𝒜1 ∪ set M ∪ ?X)"
        using * Decompose.hyps(2) by auto
      hence "Fun g S ∈ subtermsset (⋃(ikst`dualst ` 𝒮1))
            ∨ Fun g S ∈ subtermsset (ikest 𝒜1)
            ∨ Fun g S ∈ subtermsset (set M)
            ∨ Fun g S ∈ subtermsset (⋃(assignment_rhsst`𝒮1))
            ∨ Fun g S ∈ subtermsset (assignment_rhsest 𝒜1)"
        using Decompose * Ana_fun_subterm[OF Ana] by auto
      moreover have "Fun f T ∈ subtermsset (ikest 𝒜1 ∪ assignment_rhsest 𝒜1)"
        using trmsest_ik_subtermsI Decompose.hyps(1) by auto
      hence "subterms (Fun f T) ⊆ subtermsset (ikest 𝒜1 ∪ assignment_rhsest 𝒜1)"
```

```
        by (metis in_subterms_subset_Union)
      hence "subterms_set (set M) ⊆ subterms_set (ik_est 𝒜1 ∪ assignment_rhs_est 𝒜1)"
        by (meson Un_upper2 Ana_subterm[OF Ana] subterms_subset_set psubsetE subset_trans)
      ultimately have "Fun g S ∈ subterms_set (⋃(ik_st`dual_st ` 𝒮1) ∪ ik_est 𝒜1 ∪ ?Y)"
        by auto
    }
    thus ?case using Decompose unfolding Ana_invar_subst_def by metis
  qed
qed

private lemma pts_symbolic_c_preserves_constr_disj_vars:
  assumes "(𝒮,𝒜) ⇒•c* (𝒮',𝒜')" "wf_sts' 𝒮 𝒜" "fv_est 𝒜 ∩ bvars_est 𝒜 = {}"
  shows "fv_est 𝒜' ∩ bvars_est 𝒜' = {}"
using assms
proof (induction rule: rtranclp_induct2)
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  have *: "⋀S. S ∈ 𝒮1 ⟹ fv_st S ∩ bvars_est 𝒜1 = {}" "⋀S. S ∈ 𝒮1 ⟹ fv_est 𝒜1 ∩ bvars_st S = {}"
    using pts_symbolic_c_preserves_wf_prot[OF step.hyps(1) step.prems(1)]
    unfolding wf_sts'_def by auto
  from step.hyps(2) step.IH[OF step.prems]
  show ?case
  proof (induction rule: pts_symbolic_c_induct)
    case Nil thus ?case by auto
  next
    case (Send t S)
    hence "fv_est 𝒜2 = fv_est 𝒜1 ∪ fv t" "bvars_est 𝒜2 = bvars_est 𝒜1"
          "fv_st (send⟨t⟩_st#S) = fv t ∪ fv_st S"
      using fv_est_append bvars_est_append by simp+
    thus ?case using *(1)[OF Send(1)] Send(4) by auto
  next
    case (Receive t S)
    hence "fv_est 𝒜2 = fv_est 𝒜1 ∪ fv t" "bvars_est 𝒜2 = bvars_est 𝒜1"
          "fv_st (receive⟨t⟩_st#S) = fv t ∪ fv_st S"
      using fv_est_append bvars_est_append by simp+
    thus ?case using *(1)[OF Receive(1)] Receive(4) by auto
  next
    case (Equality a t t' S)
    hence "fv_est 𝒜2 = fv_est 𝒜1 ∪ fv t ∪ fv t'" "bvars_est 𝒜2 = bvars_est 𝒜1"
          "fv_st (⟨a: t ≐ t'⟩_st#S) = fv t ∪ fv t' ∪ fv_st S"
      using fv_est_append bvars_est_append by fastforce+
    thus ?case using *(1)[OF Equality(1)] Equality(4) by auto
  next
    case (Inequality X F S)
    hence "fv_est 𝒜2 = fv_est 𝒜1 ∪ (fv_pairs F - set X)" "bvars_est 𝒜2 = bvars_est 𝒜1 ∪ set X"
          "fv_st (∀X⟨∨≠: F⟩_st#S) = (fv_pairs F - set X) ∪ fv_st S"
      using fv_est_append bvars_est_append strand_vars_split(3)[of "[∀X⟨∨≠: F⟩_st]" S]
      by auto+
    moreover have "fv_est 𝒜1 ∩ set X = {}" using *(2)[OF Inequality(1)] by auto
    ultimately show ?case using *(1)[OF Inequality(1)] Inequality(4) by auto
  next
    case (Decompose f T)
    thus ?case
      using Decompose(3,4) bvars_decomp ik_assignment_rhs_decomp_fv[OF Decompose(1)] by auto
  qed
qed
```

Theorem: The Typing Result Lifted to the Transition System Level

```
private lemma wf_sts'_decomp_rm:
  assumes "well_analyzed A" "wf_sts' S (decomp_rm_est A)" shows "wf_sts' S A"
unfolding wf_sts'_def
proof (intro conjI)
  show "∀S∈S. wf_st (wfrestrictedvars_est A) (dual_st S)"
```

```
    by (metis (no_types) assms(2) wf_sts'_def wfrestrictedvars_est_decomp_rm_est_subset
             wf_vars_mono le_iff_sup)

  show "∀Sa∈S. ∀S'∈S. fv_st Sa ∩ bvars_st S' = {}" by (metis assms(2) wf_sts'_def)

  show "∀S∈S. fv_st S ∩ bvars_est A = {}" by (metis assms(2) wf_sts'_def bvars_decomp_rm)

  show "∀S∈S. fv_est A ∩ bvars_st S = {}" by (metis assms wf_sts'_def well_analyzed_decomp_rm_est_fv)
qed

private lemma decomps_est_pts_symbolic_c:
  assumes "D ∈ decomps_est (ik_est A) (assignment_rhs_est A) ℐ"
  shows "(S,A) ⇒•c* (S,A@D)"
using assms(1)
proof (induction D rule: decomps_est.induct)
  case (Decomp B f X K T)
  have "subterms_set (ik_est A ∪ assignment_rhs_est A) ⊆
        subterms_set (ik_est (A@B) ∪ assignment_rhs_est (A@B))"
    using ik_est_append[of A B] assignment_rhs_est_append[of A B]
    by auto
  hence "Fun f X ∈ subterms_set (ik_est (A@B) ∪ assignment_rhs_est (A@B))" using Decomp.hyps by auto
  hence "(S,A@B) ⇒•c (S,A@B@[Decomp (Fun f X)])"
    using pts_symbolic_c.Decompose[of f X "A@B"]
    by simp
  thus ?case
    using Decomp.IH rtrancl_into_rtrancl
          rtranclp_rtrancl_eq[of pts_symbolic_c "(S,A)" "(S,A@B)"]
    by auto
qed simp

private lemma pts_symbolic_to_pts_symbolic_c:
  assumes "(𝒮,to_st (decomp_rm_est 𝒜_d)) ⇒•* (𝒮',𝒜')" "sem_est_d {} ℐ (to_est 𝒜')" "sem_est_c {} ℐ
𝒜_d"
  and wf: "wf_sts' 𝒮 (decomp_rm_est 𝒜_d)" "wf_est {} 𝒜_d"
  and tar: "Ana_invar_subst ((⋃(ik_st ` dual_st ` 𝒮) ∪ (ik_est 𝒜_d))
                          ∪ (⋃(assignment_rhs_st ` 𝒮) ∪ (assignment_rhs_est 𝒜_d)))"
  and wa: "well_analyzed 𝒜_d"
  and ℐ: "interpretation_subst ℐ"
  shows "∃𝒜_d'. 𝒜' = to_st (decomp_rm_est 𝒜_d') ∧ (𝒮,𝒜_d) ⇒•c* (𝒮',𝒜_d') ∧ sem_est_c {} ℐ 𝒜_d'"
using assms(1,2)
proof (induction rule: rtranclp_induct2)
  case refl thus ?case using assms by auto
next
  case (step 𝒮1 𝒜1 𝒮2 𝒜2)
  have "sem_est_d {} ℐ (to_est 𝒜1)" using step.hyps(2) step.prems
    by (induct rule: pts_symbolic_induct, metis, (metis sem_est_d_split_left to_est_append)+)
  then obtain 𝒜1d where
      𝒜1d: "𝒜1 = to_st (decomp_rm_est 𝒜1d)" "(𝒮, 𝒜_d) ⇒•c* (𝒮1, 𝒜1d)" "sem_est_c {} ℐ 𝒜1d"
    using step.IH by moura

  show ?case using step.hyps(2)
  proof (induction rule: pts_symbolic_induct)
    case Nil
    hence "(𝒮, 𝒜_d) ⇒•c* (𝒮2, 𝒜1d)" using 𝒜1d pts_symbolic_c.Nil[OF Nil.hyps(1), of 𝒜1d] by simp
    thus ?case using 𝒜1d Nil by auto
  next
    case (Send t S)
    hence "sem_est_c {} ℐ (𝒜1d@[Step (receive⟨t⟩_st)])" using sem_est_c.Receive[OF 𝒜1d(3)] by simp
    moreover have "(𝒮1, 𝒜1d) ⇒•c (𝒮2, 𝒜1d@[Step (receive⟨t⟩_st)])"
      using Send.hyps(2) pts_symbolic_c.Send[OF Send.hyps(1), of 𝒜1d] by simp
    moreover have "to_st (decomp_rm_est (𝒜1d@[Step (receive⟨t⟩_st)])) = 𝒜2"
      using Send.hyps(3) decomp_rm_est_append 𝒜1d(1) by (simp add: to_st_append)
    ultimately show ?case using 𝒜1d(2) by auto
```

```
  next
    case (Equality a t t' S)
    hence "t · ℐ = t' · ℐ"
      using step.prems sem_est_d_eq_sem_st[of "{}" ℐ "to_est 𝒜2"]
            to_st_append to_est_append to_st_to_est_inv
      by auto
    hence "sem_est_c {} ℐ (𝒜1d@[Step (⟨a: t ≐ t'⟩_st)])" using sem_est_c.Equality[OF 𝒜1d(3)] by simp
    moreover have "(𝒮1, 𝒜1d) ⇒•_c (𝒮2, 𝒜1d@[Step (⟨a: t ≐ t'⟩_st)])"
      using Equality.hyps(2) pts_symbolic_c.Equality[OF Equality.hyps(1), of 𝒜1d] by simp
    moreover have "to_st (decomp_rm_est (𝒜1d@[Step (⟨a: t ≐ t'⟩_st)])) = 𝒜2"
      using Equality.hyps(3) decomp_rm_est_append 𝒜1d(1) by (simp add: to_st_append)
    ultimately show ?case using 𝒜1d(2) by auto
  next
    case (Inequality X F S)
    hence "ineq_model ℐ X F"
      using step.prems sem_est_d_eq_sem_st[of "{}" ℐ "to_est 𝒜2"]
            to_st_append to_est_append to_st_to_est_inv
      by auto
    hence "sem_est_c {} ℐ (𝒜1d@[Step (∀X⟨∨≠: F⟩_st)])" using sem_est_c.Inequality[OF 𝒜1d(3)] by simp
    moreover have "(𝒮1, 𝒜1d) ⇒•_c (𝒮2, 𝒜1d@[Step (∀X⟨∨≠: F⟩_st)])"
      using Inequality.hyps(2) pts_symbolic_c.Inequality[OF Inequality.hyps(1), of 𝒜1d] by simp
    moreover have "to_st (decomp_rm_est (𝒜1d@[Step (∀X⟨∨≠: F⟩_st)])) = 𝒜2"
      using Inequality.hyps(3) decomp_rm_est_append 𝒜1d(1) by (simp add: to_st_append)
    ultimately show ?case using 𝒜1d(2) by auto
  next
    case (Receive t S)
    hence "ik_st 𝒜1 ·_set ℐ ⊢ t · ℐ"
      using step.prems sem_est_d_eq_sem_st[of "{}" ℐ "to_est 𝒜2"]
            strand_sem_split(4)[of "{}" 𝒜1 "[send⟨t⟩_st]" ℐ]
            to_st_append to_est_append to_st_to_est_inv
      by auto
    moreover have "ik_st 𝒜1 ·_set ℐ ⊆ ik_est 𝒜1d ·_set ℐ" using 𝒜1d(1) decomp_rm_est_ik_subset by auto
    ultimately have *: "ik_est 𝒜1d ·_set ℐ ⊢ t · ℐ" using ideduct_mono by auto

    have "wf_sts' 𝒮 𝒜_d" by (rule wf_sts'_decomp_rm[OF wa assms(4)])
    hence **: "wf_est {} 𝒜1d" by (rule pts_symbolic_c_preserves_wf_is[OF 𝒜1d(2) _ assms(5)])

    have "Ana_invar_subst (⋃(ik_st`dual_st`𝒮1) ∪ (ik_est 𝒜1d) ∪
                           (⋃(assignment_rhs_st`𝒮1) ∪ (assignment_rhs_est 𝒜1d)))"
      using tar 𝒜1d(2) pts_symbolic_c_preserves_Ana_invar_subst by metis
    hence "Ana_invar_subst (ik_est 𝒜1d)" "Ana_invar_subst (assignment_rhs_est 𝒜1d)"
      using Ana_invar_subst_subset by blast+
    moreover have "well_analyzed 𝒜1d"
      using pts_symbolic_c_preserves_well_analyzed[OF 𝒜1d(2) wa] by metis
    ultimately obtain D where D:
        "D ∈ decomps_est (ik_est 𝒜1d) (assignment_rhs_est 𝒜1d) ℐ"
        "ik_est (𝒜1d@D) ·_set ℐ ⊢_c t · ℐ"
      using decomps_est_exist_subst[OF * 𝒜1d(3) ** assms(8)] unfolding Ana_invar_subst_def by auto

    have "(𝒮, 𝒜_d) ⇒•_c* (𝒮1, 𝒜1d@D)" using 𝒜1d(2) decomps_est_pts_symbolic_c[OF D(1), of 𝒮1] by
auto
    hence "(𝒮, 𝒜_d) ⇒•_c* (𝒮2, 𝒜1d@D@[Step (send⟨t⟩_st)])"
      using Receive(2) pts_symbolic_c.Receive[OF Receive.hyps(1), of "𝒜1d@D"] by auto
    moreover have "𝒜2 = to_st (decomp_rm_est (𝒜1d@D@[Step (send⟨t⟩_st)]))"
      using Receive.hyps(3) 𝒜1d(1) decomps_est_decomp_rm_est_empty[OF D(1)]
            decomp_rm_est_append to_st_append
      by auto
    moreover have "sem_est_c {} ℐ (𝒜1d@D@[Step (send⟨t⟩_st)])"
      using D(2) sem_est_c.Send[OF sem_est_c_decomps_est_append[OF 𝒜1d(3) D(1)]] by simp
    ultimately show ?case by auto
  qed
qed
```

```
private lemma pts_symbolic_c_to_pts_symbolic:
  assumes "(𝒮,𝒜) ⇒•c* (𝒮',𝒜')" "sem_est_c {} ℐ 𝒜'"
  shows "(𝒮,to_st (decomp_rm_est 𝒜)) ⇒•* (𝒮',to_st (decomp_rm_est 𝒜'))"
        "sem_est_d {} ℐ (decomp_rm_est 𝒜')"
proof -
  show "(𝒮,to_st (decomp_rm_est 𝒜)) ⇒•* (𝒮',to_st (decomp_rm_est 𝒜'))" using assms(1)
  proof (induction rule: rtranclp_induct2)
    case (step 𝒮1 𝒜1 𝒮2 𝒜2) show ?case using step.hyps(2,1) step.IH
    proof (induction rule: pts_symbolic_c_induct)
      case Nil thus ?case
        using pts_symbolic.Nil[OF Nil.hyps(1), of "to_st (decomp_rm_est 𝒜1)"] by simp
    next
      case (Send t S) thus ?case
        using pts_symbolic.Send[OF Send.hyps(1), of "to_st (decomp_rm_est 𝒜1)"]
        by (simp add: decomp_rm_est_append to_st_append)
    next
      case (Receive t S) thus ?case
        using pts_symbolic.Receive[OF Receive.hyps(1), of "to_st (decomp_rm_est 𝒜1)"]
        by (simp add: decomp_rm_est_append to_st_append)
    next
      case (Equality a t t' S) thus ?case
        using pts_symbolic.Equality[OF Equality.hyps(1), of "to_st (decomp_rm_est 𝒜1)"]
        by (simp add: decomp_rm_est_append to_st_append)
    next
      case (Inequality t t' S) thus ?case
        using pts_symbolic.Inequality[OF Inequality.hyps(1), of "to_st (decomp_rm_est 𝒜1)"]
        by (simp add: decomp_rm_est_append to_st_append)
    next
      case (Decompose t) thus ?case using decomp_rm_est_append by simp
    qed
  qed simp
qed (rule sem_est_d_decomp_rm_est_if_sem_est_c[OF assms(2)])

private lemma pts_symbolic_to_pts_symbolic_c_from_initial:
  assumes "(𝒮0,[]) ⇒•* (𝒮,𝒜)" "ℐ ⊨ ⟨𝒜⟩" "wf_sts' 𝒮0 []"
  and "Ana_invar_subst (⋃(ik_st ` dual_st ` 𝒮0) ∪ ⋃(assignment_rhs_st ` 𝒮0))" "interpretation_subst ℐ"
  shows "∃𝒜d. 𝒜 = to_st (decomp_rm_est 𝒜d) ∧ (𝒮0,[]) ⇒•c* (𝒮,𝒜d) ∧ (ℐ ⊨c ⟨to_st 𝒜d⟩)"
using assms pts_symbolic_to_pts_symbolic_c[of 𝒮0 "[]" 𝒮 𝒜 ℐ]
      sem_est_c_eq_sem_st[of "{}" ℐ] sem_est_d_eq_sem_st[of "{}" ℐ]
      to_st_to_est_inv[of 𝒜] strand_sem_eq_defs
by (auto simp add: constr_sem_c_def constr_sem_d_def simp del: subst_range.simps)

private lemma pts_symbolic_c_to_pts_symbolic_from_initial:
  assumes "(𝒮0,[]) ⇒•c* (𝒮,𝒜)" "ℐ ⊨c ⟨to_st 𝒜⟩"
  shows "(𝒮0,[]) ⇒•* (𝒮,to_st (decomp_rm_est 𝒜))" "ℐ ⊨ ⟨to_st (decomp_rm_est 𝒜)⟩"
using assms pts_symbolic_c_to_pts_symbolic[of 𝒮0 "[]" 𝒮 𝒜 ℐ]
      sem_est_c_eq_sem_st[of "{}" ℐ] sem_est_d_eq_sem_st[of "{}" ℐ] strand_sem_eq_defs
by (auto simp add: constr_sem_c_def constr_sem_d_def)

private lemma to_st_trms_wf:
  assumes "wf_trms (trms_est A)"
  shows "wf_trms (trms_st (to_st A))"
using assms
proof (induction A)
  case (Cons x A)
  hence IH: "∀t ∈ trms_st (to_st A). wf_trm t" by auto
  with Cons show ?case
  proof (cases x)
    case (Decomp t)
    hence "wf_trm t" using Cons.prems by auto
    obtain K T where Ana_t: "Ana t = (K,T)" by moura
    hence "trms_st (decomp t) ⊆ {t} ∪ set K ∪ set T" using decomp_set_unfold[OF Ana_t] by force
    moreover have "∀t ∈ set T. wf_trm t" using Ana_subterm[OF Ana_t] ‹wf_trm t› wf_trm_subterm by
```

```
auto
    ultimately have "∀t ∈ trms_st (decomp t). wf_trm t" using Ana_keys_wf'[OF Ana_t] ‹wf_trm t› by auto
    thus ?thesis using IH Decomp by auto
  qed auto
qed simp

private lemma to_st_trms_SMP_subset: "trms_st (to_st A) ⊆ SMP (trms_est A)"
proof
  fix t assume "t ∈ trms_st (to_st A)" thus "t ∈ SMP (trms_est A)"
  proof (induction A)
    case (Cons x A)
    hence *: "t ∈ trms_st (to_st [x]) ∪ trms_st (to_st A)" using to_st_append[of "[x]" A] by auto
    have **: "trms_st (to_st A) ⊆ trms_st (to_st (x#A))" "trms_est A ⊆ trms_est (x#A)"
      using to_st_append[of "[x]" A] by auto
    show ?case
    proof (cases "t ∈ trms_st (to_st A)")
      case True thus ?thesis using Cons.IH SMP_mono[OF **(2)] by auto
    next
      case False
      hence ***: "t ∈ trms_st (to_st [x])" using * by auto
      thus ?thesis
      proof (cases x)
        case (Decomp t')
        hence ****: "t ∈ trms_st (decomp t')" "t' ∈ trms_est (x#A)" using *** by auto
        obtain K T where Ana_t': "Ana t' = (K,T)" by moura
        hence "t ∈ {t'} ∪ set K ∪ set T" using decomp_set_unfold[OF Ana_t'] ****(1) by force
        moreover
        { assume "t = t'" hence ?thesis using SMP.MP[OF ****(2)] by simp }
        moreover
        { assume "t ∈ set K" hence ?thesis using SMP.Ana[OF SMP.MP[OF ****(2)] Ana_t'] by auto }
        moreover
        { assume "t ∈ set T" "t ≠ t'"
          hence "t ⊏ t'" using Ana_subterm[OF Ana_t'] by blast
          hence ?thesis using SMP.Subterm[OF SMP.MP[OF ****(2)]] by auto
        }
        ultimately show ?thesis using Decomp by auto
      qed auto
    qed
  qed simp
qed

private lemma to_st_trms_tfr_set:
  assumes "tfr_set (trms_est A)"
  shows "tfr_set (trms_st (to_st A))"
proof -
  have *: "trms_st (to_st A) ⊆ SMP (trms_est A)"
    using to_st_trms_wf to_st_trms_SMP_subset assms unfolding tfr_set_def by auto
  have "trms_st (to_st A) = trms_st (to_st A) ∪ trms_est A" by (blast dest!: trms_estD)
  hence "SMP (trms_est A) = SMP (trms_st (to_st A))" using SMP_subset_union_eq[OF *] by auto
  thus ?thesis using * assms unfolding tfr_set_def by presburger
qed

theorem wt_attack_if_tfr_attack_pts:
  assumes "wf_sts 𝒮₀" "tfr_set (⋃(trms_st ` 𝒮₀))" "wf_trms (⋃(trms_st ` 𝒮₀))" "∀S ∈ 𝒮₀. list_all tfr_stp
S"
  and "Ana_invar_subst (⋃(ik_st ` dual_st ` 𝒮₀) ∪ ⋃(assignment_rhs_st ` 𝒮₀))"
  and "(𝒮₀,[]) ⇒•* (𝒮,𝒜)" "interpretation_subst ℐ" "ℐ ⊨ ⟨𝒜, Var⟩"
  shows "∃ℐ_τ. interpretation_subst ℐ_τ ∧ (ℐ_τ ⊨ ⟨𝒜, Var⟩) ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ)"
proof -
  have "(⋃(trms_st ` 𝒮₀)) ∪ (trms_est []) = ⋃(trms_st ` 𝒮₀)" "to_st [] = []" "list_all tfr_stp []"
    using assms by simp_all
  hence *: "tfr_set ((⋃(trms_st ` 𝒮₀)) ∪ (trms_est []))"
           "wf_trms ((⋃(trms_st ` 𝒮₀)) ∪ (trms_est []))"
```

```
      "wf_sts' S₀ []" "∀S ∈ S₀ ∪ {to_st []}. list_all tfr_stp S"
    using assms wf_sts_wf_sts' by (metis, metis, metis, simp)

  obtain A_d where A_d: "A = to_st (decomp_rm_est A_d)" "(S₀,[]) ⇒•_c* (S,A_d)" "I ⊨_c ⟨to_st A_d⟩"
    using pts_symbolic_to_pts_symbolic_c_from_initial assms *(3) by metis
  hence "tfr_set (⋃(trms_st ` S) ∪ (trms_est A_d))" "wf_trms (⋃(trms_st ` S) ∪ (trms_est A_d))"
    using pts_symbolic_c_preserves_tfr_set[OF _ *(1,2)] by blast+
  hence "tfr_set (trms_est A_d)" "wf_trms (trms_est A_d)"
    unfolding tfr_set_def by (metis DiffE DiffI SMP_union UnCI, metis UnCI)
  hence "tfr_set (trms_st (to_st A_d))" "wf_trms (trms_st (to_st A_d))"
    by (metis to_st_trms_tfr_set, metis to_st_trms_wf)
  moreover have "wf_constr (to_st A_d) Var"
  proof -
    have "wt_subst Var" "wf_trms (subst_range Var)" "subst_domain Var ∩ vars_est A_d = {}"
         "range_vars Var ∩ bvars_est A_d = {}"
      by (simp_all add: range_vars_alt_def)
    moreover have "wf_est {} A_d"
      using pts_symbolic_c_preserves_wf_is[OF A_d(2) *(3), of "{}"]
      by auto
    moreover have "fv_st (to_st A_d) ∩ bvars_est A_d = {}"
      using pts_symbolic_c_preserves_constr_disj_vars[OF A_d(2)] assms(1) wf_sts_wf_sts'
      by fastforce
    ultimately show ?thesis unfolding wf_constr_def wf_subst_def by simp
  qed
  moreover have "list_all tfr_stp (to_st A_d)"
    using pts_symbolic_c_preserves_tfr_stp[OF A_d(2) *(4)] by blast
  moreover have "wt_subst Var" "wf_trms (subst_range Var)" by simp_all
  ultimately obtain I_τ where I_τ:
      "interpretation_subst I_τ" "I_τ ⊨_c ⟨to_st A_d, Var⟩" "wt_subst I_τ" "wf_trms (subst_range I_τ)"
    using wt_attack_if_tfr_attack[OF assms(7) A_d(3)]
          ‹tfr_set (trms_st (to_st A_d))› ‹list_all tfr_stp (to_st A_d)›
    unfolding tfr_st_def by metis
  hence "I_τ ⊨ ⟨A, Var⟩" using pts_symbolic_c_to_pts_symbolic_from_initial A_d by metis
  thus ?thesis using I_τ(1,3,4) by metis
qed
```

Corollary: The Typing Result on the Level of Constraints

There exists well-typed models of satisfiable type-flaw resistant constraints

```
corollary wt_attack_if_tfr_attack_d:
  assumes "wf_st {} A" "fv_st A ∩ bvars_st A = {}" "tfr_st A" "wf_trms (trms_st A)"
  and "Ana_invar_subst (ik_st A ∪ assignment_rhs_st A)"
  and "interpretation_subst I" "I ⊨ ⟨A⟩"
  shows "∃I_τ. interpretation_subst I_τ ∧ (I_τ ⊨ ⟨A⟩) ∧ wt_subst I_τ ∧ wf_trms (subst_range I_τ)"
proof -
  { fix S A have "({S},A) ⇒•* ({},A@dual_st S)"
    proof (induction S arbitrary: A)
      case Nil thus ?case using pts_symbolic.Nil[of "{[]}"] by auto
    next
      case (Cons x S)
      hence "({S}, A@dual_st [x]) ⇒•* ({}, A@dual_st (x#S))"
        by (metis dual_st_append List.append_assoc List.append_Nil List.append_Cons)
      moreover have "({x#S}, A) ⇒• ({S}, A@dual_st [x])"
        using pts_symbolic.Send[of _ S "{x#S}"] pts_symbolic.Receive[of _ S "{x#S}"]
              pts_symbolic.Equality[of _ _ _ S "{x#S}"] pts_symbolic.Inequality[of _ _ S "{x#S}"]
        by (cases x) auto
      ultimately show ?case by simp
    qed
  }
  hence 0: "({dual_st A},[]) ⇒•* ({},A)" using dual_st_self_inverse by (metis List.append_Nil)

  have "fv_st (dual_st A) ∩ bvars_st (dual_st A) = {}" using assms(2) dual_st_fv dual_st_bvars by metis+
  hence 1: "wf_sts {dual_st A}" using assms(1,2) dual_st_self_inverse[of A] unfolding wf_sts_def by auto
```

```
  have "⋃(trms_st ` {𝒜}) = trms_st 𝒜" "⋃(trms_st ` {dual_st 𝒜}) = trms_st (dual_st 𝒜)" by auto
  hence "tfr_set (⋃(trms_st ` {𝒜}))" "wf_trms (⋃(trms_st ` {𝒜}))"
        "(⋃(trms_st ` {𝒜})) = ⋃(trms_st ` {dual_st 𝒜})"
    using assms(3,4) unfolding tfr_st_def
    by (metis, metis, metis dual_st_trms_eq)
  hence 2: "tfr_set (⋃(trms_st ` {dual_st 𝒜}))" and 3: "wf_trms (⋃(trms_st ` {dual_st 𝒜}))" by metis+

  have 4: "∀S ∈ {dual_st 𝒜}. list_all tfr_stp S"
    using dual_st_tfr_stp assms(3) unfolding tfr_st_def by blast

  have "assignment_rhs_st 𝒜 = assignment_rhs_st (dual_st 𝒜)"
    by (induct 𝒜 rule: assignment_rhs_st.induct) auto
  hence 5: "Ana_invar_subst (⋃(ik_st`dual_st`{dual_st 𝒜}) ∪ ⋃(assignment_rhs_st`{dual_st 𝒜}))"
    using assms(5) dual_st_self_inverse[of 𝒜] by auto

  show ?thesis by (rule wt_attack_if_tfr_attack_pts[OF 1 2 3 4 5 0 assms(6,7)])
qed

end

end

end
```

4 The Typing Result for Stateful Protocols

In this chapter, we lift the typing result to stateful protocols. For more details, we refer the reader to [3] and [1, chapter 4].

4.1 Stateful Strands (Stateful_Strands)

```
theory Stateful_Strands
imports Strands_and_Constraints
begin
```

4.1.1 Stateful Constraints

```
datatype (funs_sstp: 'a, vars_sstp: 'b) stateful_strand_step =
  Send (the_msg: "('a,'b) term") ("send⟨_⟩" 80)
| Receive (the_msg: "('a,'b) term") ("receive⟨_⟩" 80)
| Equality (the_check: poscheckvariant) (the_lhs: "('a,'b) term") (the_rhs: "('a,'b) term")
    ("⟨_: _ ≐ _⟩" [80,80])
| Insert (the_elem_term: "('a,'b) term") (the_set_term: "('a,'b) term") ("insert⟨_,_⟩" 80)
| Delete (the_elem_term: "('a,'b) term") (the_set_term: "('a,'b) term") ("delete⟨_,_⟩" 80)
| InSet (the_check: poscheckvariant) (the_elem_term: "('a,'b) term") (the_set_term: "('a,'b) term")
    ("⟨_: _ ∈ _⟩" [80,80])
| NegChecks (bvars_sstp: "'b list")
    (the_eqs: "(('a,'b) term × ('a,'b) term) list")
    (the_ins: "(('a,'b) term × ('a,'b) term) list")
    ("∀_⟨∨≠: _ ∨∉: _⟩" [80,80])
where
  "bvars_sstp (Send _) = []"
| "bvars_sstp (Receive _) = []"
| "bvars_sstp (Equality _ _ _) = []"
| "bvars_sstp (Insert _ _) = []"
| "bvars_sstp (Delete _ _) = []"
| "bvars_sstp (InSet _ _ _) = []"
```

```
type_synonym ('a,'b) stateful_strand = "('a,'b) stateful_strand_step list"
type_synonym ('a,'b) dbstatelist = "(('a,'b) term × ('a,'b) term) list"
type_synonym ('a,'b) dbstate = "(('a,'b) term × ('a,'b) term) set"
```

```
abbreviation
  "is_Assignment x ≡ (is_Equality x ∨ is_InSet x) ∧ the_check x = Assign"
```

```
abbreviation
  "is_Check x ≡ ((is_Equality x ∨ is_InSet x) ∧ the_check x = Check) ∨ is_NegChecks x"
```

```
abbreviation
  "is_Update x ≡ is_Insert x ∨ is_Delete x"
```

```
abbreviation InSet_select ("select⟨_,_⟩") where "select⟨t,s⟩ ≡ InSet Assign t s"
abbreviation InSet_check ("⟨_ in _⟩") where "⟨t in s⟩ ≡ InSet Check t s"
abbreviation Equality_assign ("⟨_ := _⟩") where "⟨t := s⟩ ≡ Equality Assign t s"
abbreviation Equality_check ("⟨_ == _⟩") where "⟨t == s⟩ ≡ Equality Check t s"
```

```
abbreviation NegChecks_Inequality1 ("⟨_ != _⟩") where
  "⟨t != s⟩ ≡ NegChecks [] [(t,s)] []"
```

```
abbreviation NegChecks_Inequality2 ("∀_⟨_ != _⟩") where
```

```
"∀x⟨t != s⟩ ≡ NegChecks [x] [(t,s)] []"
```

abbreviation *NegChecks_Inequality3* ("∀_,_⟨_ != _⟩") **where**
```
"∀x,y⟨t != s⟩ ≡ NegChecks [x,y] [(t,s)] []"
```

abbreviation *NegChecks_Inequality4* ("∀_,_,_⟨_ != _⟩") **where**
```
"∀x,y,z⟨t != s⟩ ≡ NegChecks [x,y,z] [(t,s)] []"
```

abbreviation *NegChecks_NotInSet1* ("⟨_ not in _⟩") **where**
```
"⟨t not in s⟩ ≡ NegChecks [] [] [(t,s)]"
```

abbreviation *NegChecks_NotInSet2* ("∀_⟨_ not in _⟩") **where**
```
"∀x⟨t not in s⟩ ≡ NegChecks [x] [] [(t,s)]"
```

abbreviation *NegChecks_NotInSet3* ("∀_,_⟨_ not in _⟩") **where**
```
"∀x,y⟨t not in s⟩ ≡ NegChecks [x,y] [] [(t,s)]"
```

abbreviation *NegChecks_NotInSet4* ("∀_,_,_⟨_ not in _⟩") **where**
```
"∀x,y,z⟨t not in s⟩ ≡ NegChecks [x,y,z] [] [(t,s)]"
```

fun *trms*$_{sstp}$ **where**
```
  "trms_sstp (Send t) = {t}"
| "trms_sstp (Receive t) = {t}"
| "trms_sstp (Equality _ t t') = {t,t'}"
| "trms_sstp (Insert t t') = {t,t'}"
| "trms_sstp (Delete t t') = {t,t'}"
| "trms_sstp (InSet _ t t') = {t,t'}"
| "trms_sstp (NegChecks _ F F') = trms_pairs F ∪ trms_pairs F'"
```

definition *trms*$_{sst}$ **where** "trms$_{sst}$ S ≡ ⋃(trms$_{sstp}$ ‘ set S)"
declare *trms*$_{sst}$*_def[simp]*

fun *trms_list*$_{sstp}$ **where**
```
  "trms_list_sstp (Send t) = [t]"
| "trms_list_sstp (Receive t) = [t]"
| "trms_list_sstp (Equality _ t t') = [t,t']"
| "trms_list_sstp (Insert t t') = [t,t']"
| "trms_list_sstp (Delete t t') = [t,t']"
| "trms_list_sstp (InSet _ t t') = [t,t']"
| "trms_list_sstp (NegChecks _ F F') = concat (map (λ(t,t'). [t,t']) (F@F'))"
```

definition *trms_list*$_{sst}$ **where** "trms_list$_{sst}$ S ≡ remdups (concat (map trms_list$_{sstp}$ S))"

definition *ik*$_{sst}$ **where** "ik$_{sst}$ A ≡ {t. Receive t ∈ set A}"

definition *bvars*$_{sst}$::"('a,'b) stateful_strand ⇒ 'b set" **where**
```
"bvars_sst S ≡ ⋃(set (map (set ∘ bvars_sstp) S))"
```

fun *fv*$_{sstp}$::"('a,'b) stateful_strand_step ⇒ 'b set" **where**
```
  "fv_sstp (Send t) = fv t"
| "fv_sstp (Receive t) = fv t"
| "fv_sstp (Equality _ t t') = fv t ∪ fv t'"
| "fv_sstp (Insert t t') = fv t ∪ fv t'"
| "fv_sstp (Delete t t') = fv t ∪ fv t'"
| "fv_sstp (InSet _ t t') = fv t ∪ fv t'"
| "fv_sstp (NegChecks X F F') = fv_pairs F ∪ fv_pairs F' - set X"
```

definition *fv*$_{sst}$::"('a,'b) stateful_strand ⇒ 'b set" **where**
```
"fv_sst S ≡ ⋃(set (map fv_sstp S))"
```

fun *fv_list*$_{sstp}$ **where**
```
  "fv_list_sstp (send⟨t⟩) = fv_list t"
| "fv_list_sstp (receive⟨t⟩) = fv_list t"
```

```
| "fv_list_sstp (⟨_: t ≐ s⟩) = fv_list t@fv_list s"
| "fv_list_sstp (insert⟨t,s⟩) = fv_list t@fv_list s"
| "fv_list_sstp (delete⟨t,s⟩) = fv_list t@fv_list s"
| "fv_list_sstp (⟨_: t ∈ s⟩) = fv_list t@fv_list s"
| "fv_list_sstp (∀X⟨∨≠: F ∨∉: F'⟩) = filter (λx. x ∉ set X) (fv_list_pairs (F@F'))"

definition fv_list_sst where
  "fv_list_sst S ≡ remdups (concat (map fv_list_sstp S))"

declare bvars_sst_def[simp]
declare fv_sst_def[simp]

definition vars_sst::"('a,'b) stateful_strand ⇒ 'b set" where
  "vars_sst S ≡ ⋃(set (map vars_sstp S))"

abbreviation wfrestrictedvars_sstp::"('a,'b) stateful_strand_step ⇒ 'b set" where
  "wfrestrictedvars_sstp x ≡
    case x of
      NegChecks _ _ _ ⇒ {}
    | Equality Check _ _ ⇒ {}
    | InSet Check _ _ ⇒ {}
    | Delete _ _ ⇒ {}
    | _ ⇒ vars_sstp x"

definition wfrestrictedvars_sst::"('a,'b) stateful_strand ⇒ 'b set" where
  "wfrestrictedvars_sst S ≡ ⋃(set (map wfrestrictedvars_sstp S))"

abbreviation wfvarsoccs_sstp where
  "wfvarsoccs_sstp x ≡
    case x of
      Send t ⇒ fv t
    | Equality Assign s t ⇒ fv s
    | InSet Assign s t ⇒ fv s ∪ fv t
    | _ ⇒ {}"

definition wfvarsoccs_sst where
  "wfvarsoccs_sst S ≡ ⋃(set (map wfvarsoccs_sstp S))"

fun wf'_sst::"'b set ⇒ ('a,'b) stateful_strand ⇒ bool" where
  "wf'_sst V [] = True"
| "wf'_sst V (Receive t#S) = (fv t ⊆ V ∧ wf'_sst V S)"
| "wf'_sst V (Send t#S) = wf'_sst (V ∪ fv t) S"
| "wf'_sst V (Equality Assign t t'#S) = (fv t' ⊆ V ∧ wf'_sst (V ∪ fv t) S)"
| "wf'_sst V (Equality Check _ _#S) = wf'_sst V S"
| "wf'_sst V (Insert t s#S) = (fv t ⊆ V ∧ fv s ⊆ V ∧ wf'_sst V S)"
| "wf'_sst V (Delete _ _#S) = wf'_sst V S"
| "wf'_sst V (InSet Assign t s#S) = wf'_sst (V ∪ fv t ∪ fv s) S"
| "wf'_sst V (InSet Check _ _#S) = wf'_sst V S"
| "wf'_sst V (NegChecks _ _ _#S) = wf'_sst V S"

abbreviation "wf_sst S ≡ wf'_sst {} S ∧ fv_sst S ∩ bvars_sst S = {}"

fun subst_apply_stateful_strand_step::
  "('a,'b) stateful_strand_step ⇒ ('a,'b) subst ⇒ ('a,'b) stateful_strand_step"
  (infix "·_sstp" 51) where
  "send⟨t⟩ ·_sstp ϑ = send⟨t · ϑ⟩"
| "receive⟨t⟩ ·_sstp ϑ = receive⟨t · ϑ⟩"
| "⟨a: t ≐ s⟩ ·_sstp ϑ = ⟨a: (t · ϑ) ≐ (s · ϑ)⟩"
| "⟨a: t ∈ s⟩ ·_sstp ϑ = ⟨a: (t · ϑ) ∈ (s · ϑ)⟩"
| "insert⟨t,s⟩ ·_sstp ϑ = insert⟨t · ϑ, s · ϑ⟩"
| "delete⟨t,s⟩ ·_sstp ϑ = delete⟨t · ϑ, s · ϑ⟩"
| "∀X⟨∨≠: F ∨∉: G⟩ ·_sstp ϑ = ∀X⟨∨≠: (F ·_pairs rm_vars (set X) ϑ) ∨∉: (G ·_pairs rm_vars (set X) ϑ)⟩"
```

definition *subst_apply_stateful_strand::*
  "('a,'b) stateful_strand ⇒ ('a,'b) subst ⇒ ('a,'b) stateful_strand"
  (infix *"$\cdot_{sst}$" 51)* **where**
  "S $\cdot_{sst}$ ϑ ≡ map (λx. x $\cdot_{sstp}$ ϑ) S"

fun *$dbupd_{sst}$::"('f,'v) stateful_strand ⇒ ('f,'v) subst ⇒ ('f,'v) dbstate ⇒ ('f,'v) dbstate"*
where
  "$dbupd_{sst}$ [] I D = D"
| *"$dbupd_{sst}$ (Insert t s#A) I D = $dbupd_{sst}$ A I (insert ((t,s) $\cdot_p$ I) D)"*
| *"$dbupd_{sst}$ (Delete t s#A) I D = $dbupd_{sst}$ A I (D - {((t,s) $\cdot_p$ I)})"*
| *"$dbupd_{sst}$ (_#A) I D = $dbupd_{sst}$ A I D"*

fun *db'_{sst}::"('f,'v) stateful_strand ⇒ ('f,'v) subst ⇒ ('f,'v) dbstatelist ⇒ ('f,'v) dbstatelist"*
where
  "db'_{sst} [] I D = D"
| *"db'_{sst} (Insert t s#A) I D = db'_{sst} A I (List.insert ((t,s) $\cdot_p$ I) D)"*
| *"db'_{sst} (Delete t s#A) I D = db'_{sst} A I (List.removeAll ((t,s) $\cdot_p$ I) D)"*
| *"db'_{sst} (_#A) I D = db'_{sst} A I D"*

definition *db_{sst}* **where**
  "db_{sst} S I ≡ db'_{sst} S I []"

fun *$setops_{sstp}$* **where**
  "$setops_{sstp}$ (Insert t s) = {(t,s)}"
| *"$setops_{sstp}$ (Delete t s) = {(t,s)}"*
| *"$setops_{sstp}$ (InSet _ t s) = {(t,s)}"*
| *"$setops_{sstp}$ (NegChecks _ _ F') = set F'"*
| *"$setops_{sstp}$ _ = {}"*

The set-operations of a stateful strand

definition *$setops_{sst}$* **where**
  "$setops_{sst}$ S ≡ ⋃($setops_{sstp}$ ` set S)"

fun *$setops\_list_{sstp}$* **where**
  "$setops\_list_{sstp}$ (Insert t s) = [(t,s)]"
| *"$setops\_list_{sstp}$ (Delete t s) = [(t,s)]"*
| *"$setops\_list_{sstp}$ (InSet _ t s) = [(t,s)]"*
| *"$setops\_list_{sstp}$ (NegChecks _ _ F') = F'"*
| *"$setops\_list_{sstp}$ _ = []"*

The set-operations of a stateful strand (list variant)

definition *$setops\_list_{sst}$* **where**
  "$setops\_list_{sst}$ S ≡ remdups (concat (map $setops\_list_{sstp}$ S))"

4.1.2 Small Lemmata

lemma *$trms\_list_{sst}\_is\_trms_{sst}$: "$trms_{sst}$ S = set ($trms\_list_{sst}$ S)"*
unfolding *$trms_{st}$_def $trms_list_{sst}$_def*
proof *(induction S)*
  case *(Cons x S)* **thus** *?case* **by** *(cases x) auto*
qed *simp*

lemma *$setops\_list_{sst}\_is\_setops_{sst}$: "$setops_{sst}$ S = set ($setops\_list_{sst}$ S)"*
unfolding *$setops_{sst}$_def $setops_list_{sst}$_def*
proof *(induction S)*
  case *(Cons x S)* **thus** *?case* **by** *(cases x) auto*
qed *simp*

lemma *$fv\_list_{sstp}\_is\_fv_{sstp}$: "$fv_{sstp}$ a = set (fv_list_{sstp} a)"*
proof *(cases a)*
  case *(NegChecks X F G)* **thus** *?thesis*
    using *fv_{pairs}_append[of F G] fv_list_{pairs}_append[of F G]*
          $fv\_list_{pairs}\_is\_fv_{pairs}$[of "F@G"]

```
    by auto
qed (simp_all add: fv_list_pairs_is_fv_pairs fv_list_is_fv)

lemma fv_list_sst_is_fv_sst: "fv_sst S = set (fv_list_sst S)"
unfolding fv_sst_def fv_list_sst_def by (induct S) (simp_all add: fv_list_sstp_is_fv_sstp)

lemma trms_sstp_finite[simp]: "finite (trms_sstp x)"
by (cases x) auto

lemma trms_sst_finite[simp]: "finite (trms_sst S)"
using trms_sstp_finite unfolding trms_sst_def by (induct S) auto

lemma vars_sstp_finite[simp]: "finite (vars_sstp x)"
by (cases x) auto

lemma vars_sst_finite[simp]: "finite (vars_sst S)"
using vars_sstp_finite unfolding vars_sst_def by (induct S) auto

lemma fv_sstp_finite[simp]: "finite (fv_sstp x)"
by (cases x) auto

lemma fv_sst_finite[simp]: "finite (fv_sst S)"
using fv_sstp_finite unfolding fv_sst_def by (induct S) auto

lemma bvars_sstp_finite[simp]: "finite (set (bvars_sstp x))"
by (rule finite_set)

lemma bvars_sst_finite[simp]: "finite (bvars_sst S)"
using bvars_sstp_finite unfolding bvars_sst_def by (induct S) auto

lemma subst_sst_nil[simp]: "[] ·sst δ = []"
by (simp add: subst_apply_stateful_strand_def)

lemma db_sst_nil[simp]: "db_sst [] I = []"
by (simp add: db_sst_def)

lemma ik_sst_nil[simp]: "ik_sst [] = {}"
by (simp add: ik_sst_def)

lemma ik_sst_append[simp]: "ik_sst (A@B) = ik_sst A ∪ ik_sst B"
  by (auto simp add: ik_sst_def)

lemma ik_sst_subst: "ik_sst (A ·sst δ) = ik_sst A ·set δ"
proof (induction A)
  case (Cons a A) thus ?case
    by (cases a) (auto simp add: ik_sst_def subst_apply_stateful_strand_def)
qed simp

lemma db_sst_set_is_dbupd_sst: "set (db'_sst A I D) = dbupd_sst A I (set D)" (is "?A = ?B")
proof
  show "?A ⊆ ?B"
  proof
    fix t s show "(t,s) ∈ ?A ⟹ (t,s) ∈ ?B" by (induct rule: db'_sst.induct) auto
  qed

  show "?B ⊆ ?A"
  proof
    fix t s show "(t,s) ∈ ?B ⟹ (t,s) ∈ ?A" by (induct arbitrary: D rule: dbupd_sst.induct) auto
  qed
qed

lemma dbupd_sst_no_upd:
  assumes "∀a ∈ set A. ¬is_Insert a ∧ ¬is_Delete a"
```

```
  shows "dbupd_sst A I D = D"
using assms
proof (induction A)
  case (Cons a A) thus ?case by (cases a) auto
qed simp

lemma db_sst_no_upd:
  assumes "∀a ∈ set A. ¬is_Insert a ∧ ¬is_Delete a"
  shows "db'_sst A I D = D"
using assms
proof (induction A)
  case (Cons a A) thus ?case by (cases a) auto
qed simp

lemma db_sst_no_upd_append:
  assumes "∀b ∈ set B. ¬is_Insert b ∧ ¬is_Delete b"
  shows "db'_sst A = db'_sst (A@B)"
  using assms
proof (induction A)
  case Nil thus ?case by (simp add: db_sst_no_upd)
next
  case (Cons a A) thus ?case by (cases a) simp_all
qed

lemma db_sst_append:
  "db'_sst (A@B) I D = db'_sst B I (db'_sst A I D)"
proof (induction A arbitrary: D)
  case (Cons a A) thus ?case by (cases a) auto
qed simp

lemma db_sst_in_cases:
  assumes "(t,s) ∈ set (db'_sst A I D)"
  shows "(t,s) ∈ set D ∨ (∃t' s'. insert⟨t',s'⟩ ∈ set A ∧ t = t' · I ∧ s = s' · I)"
  using assms
proof (induction A arbitrary: D)
  case (Cons a A) thus ?case by (cases a) fastforce+
qed simp

lemma db_sst_in_cases':
  assumes "(t,s) ∈ set (db'_sst A I D)"
    and "(t,s) ∉ set D"
  shows "∃B C t' s'. A = B@insert⟨t',s'⟩#C ∧ t = t' · I ∧ s = s' · I ∧
                    (∀t'' s''. delete⟨t'',s''⟩ ∈ set C ⟶ t ≠ t'' · I ∨ s ≠ s'' · I)"
  using assms(1)
proof (induction A rule: List.rev_induct)
  case (snoc a A)
  note * = snoc db_sst_append[of A "[a]" I D]
  thus ?case
  proof (cases a)
    case (Insert t' s')
    thus ?thesis using * by (cases "(t,s) ∈ set (db'_sst A I D)") force+
  next
    case (Delete t' s')
    hence **: "t ≠ t' · I ∨ s ≠ s' · I" using * by simp

    have "(t,s) ∈ set (db'_sst A I D)" using * Delete by force
    then obtain B C u v where B:
        "A = B@insert⟨u,v⟩#C" "t = u · I" "s = v · I"
        "∀t' s'. delete⟨t',s'⟩ ∈ set C ⟶ t ≠ t' · I ∨ s ≠ s' · I"
      using snoc.IH by moura

    have "A@[a] = B@insert⟨u,v⟩#(C@[a])"
         "∀t' s'. delete⟨t',s'⟩ ∈ set (C@[a]) ⟶ t ≠ t' · I ∨ s ≠ s' · I"
```

```
      using B(1,4) Delete ** by auto
    thus ?thesis using B(2,3) by blast
  qed force+
qed (simp add: assms(2))

lemma db_sst_filter:
  "db'_sst A I D = db'_sst (filter is_Update A) I D"
by (induct A I D rule: db'_sst.induct) simp_all

lemma subst_sst_cons: "a#A ·_sst δ = (a ·_sstp δ)#(A ·_sst δ)"
by (simp add: subst_apply_stateful_strand_def)

lemma subst_sst_snoc: "A@[a] ·_sst δ = (A ·_sst δ)@[a ·_sstp δ]"
by (simp add: subst_apply_stateful_strand_def)

lemma subst_sst_append[simp]: "A@B ·_sst δ = (A ·_sst δ)@(B ·_sst δ)"
by (simp add: subst_apply_stateful_strand_def)

lemma sst_vars_append_subset:
  "fv_sst A ⊆ fv_sst (A@B)" "bvars_sst A ⊆ bvars_sst (A@B)"
  "fv_sst B ⊆ fv_sst (A@B)" "bvars_sst B ⊆ bvars_sst (A@B)"
by auto

lemma sst_vars_disj_cons[simp]: "fv_sst (a#A) ∩ bvars_sst (a#A) = {} ⟹ fv_sst A ∩ bvars_sst A = {}"
unfolding fv_sst_def bvars_sst_def by auto

lemma fv_sst_cons_subset[simp]: "fv_sst A ⊆ fv_sst (a#A)"
by auto

lemma fv_sstp_subst_cases[simp]:
  "fv_sstp (send⟨t⟩ ·_sstp ϑ) = fv (t · ϑ)"
  "fv_sstp (receive⟨t⟩ ·_sstp ϑ) = fv (t · ϑ)"
  "fv_sstp (⟨c: t ≐ s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "fv_sstp (insert⟨t,s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "fv_sstp (delete⟨t,s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "fv_sstp (⟨c: t ∈ s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "fv_sstp (∀X⟨∨≠: F ∨∉: G⟩ ·_sstp ϑ) =
    fv_pairs (F ·_pairs rm_vars (set X) ϑ) ∪ fv_pairs (G ·_pairs rm_vars (set X) ϑ) - set X"
by simp_all

lemma vars_sstp_cases[simp]:
  "vars_sstp (send⟨t⟩) = fv t"
  "vars_sstp (receive⟨t⟩) = fv t"
  "vars_sstp (⟨c: t ≐ s⟩) = fv t ∪ fv s"
  "vars_sstp (insert⟨t,s⟩) = fv t ∪ fv s"
  "vars_sstp (delete⟨t,s⟩) = fv t ∪ fv s"
  "vars_sstp (⟨c: t ∈ s⟩) = fv t ∪ fv s"
  "vars_sstp (∀X⟨∨≠: F ∨∉: G⟩) = fv_pairs F ∪ fv_pairs G ∪ set X" (is ?A)
  "vars_sstp (∀X⟨∨≠: [(t,s)] ∨∉: []⟩) = fv t ∪ fv s ∪ set X" (is ?B)
  "vars_sstp (∀X⟨∨≠: [] ∨∉: [(t,s)]⟩) = fv t ∪ fv s ∪ set X" (is ?C)
proof
  show ?A ?B ?C by auto
qed simp_all

lemma vars_sstp_subst_cases[simp]:
  "vars_sstp (send⟨t⟩ ·_sstp ϑ) = fv (t · ϑ)"
  "vars_sstp (receive⟨t⟩ ·_sstp ϑ) = fv (t · ϑ)"
  "vars_sstp (⟨c: t ≐ s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "vars_sstp (insert⟨t,s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "vars_sstp (delete⟨t,s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "vars_sstp (⟨c: t ∈ s⟩ ·_sstp ϑ) = fv (t · ϑ) ∪ fv (s · ϑ)"
  "vars_sstp (∀X⟨∨≠: F ∨∉: G⟩ ·_sstp ϑ) =
    fv_pairs (F ·_pairs rm_vars (set X) ϑ) ∪ fv_pairs (G ·_pairs rm_vars (set X) ϑ) ∪ set X" (is ?A)
```

```
  "vars_sstp (∀X⟨∨≠: [(t,s)] ∨∉: []⟩ ·_sstp ϑ) =
    fv (t · rm_vars (set X) ϑ) ∪ fv (s · rm_vars (set X) ϑ) ∪ set X" (is ?B)
  "vars_sstp (∀X⟨∨≠: [] ∨∉: [(t,s)]⟩ ·_sstp ϑ) =
    fv (t · rm_vars (set X) ϑ) ∪ fv (s · rm_vars (set X) ϑ) ∪ set X" (is ?C)
proof
  show ?A ?B ?C by auto
qed simp_all

lemma bvars_sst_cons_subset: "bvars_sst A ⊆ bvars_sst (a#A)"
by auto

lemma bvars_sstp_subst: "bvars_sstp (a ·_sstp δ) = bvars_sstp a"
by (cases a) auto

lemma bvars_sst_subst: "bvars_sst (A ·_sst δ) = bvars_sst A"
using bvars_sstp_subst[of _ δ]
by (induct A) (simp_all add: subst_apply_stateful_strand_def)

lemma bvars_sstp_set_cases[simp]:
  "set (bvars_sstp (send⟨t⟩)) = {}"
  "set (bvars_sstp (receive⟨t⟩)) = {}"
  "set (bvars_sstp (⟨c: t ≐ s⟩)) = {}"
  "set (bvars_sstp (insert⟨t,s⟩)) = {}"
  "set (bvars_sstp (delete⟨t,s⟩)) = {}"
  "set (bvars_sstp (⟨c: t ∈ s⟩)) = {}"
  "set (bvars_sstp (∀X⟨∨≠: F ∨∉: G⟩)) = set X"
by simp_all

lemma bvars_sstp_NegChecks: "¬is_NegChecks a ⟹ bvars_sstp a = []"
by (cases a) simp_all

lemma bvars_sst_NegChecks: "bvars_sst A = bvars_sst (filter is_NegChecks A)"
proof (induction A)
  case (Cons a A) thus ?case by (cases a) fastforce+
qed simp

lemma vars_sst_append[simp]: "vars_sst (A@B) = vars_sst A ∪ vars_sst B"
by (simp add: vars_sst_def)

lemma vars_sst_Nil[simp]: "vars_sst [] = {}"
by (simp add: vars_sst_def)

lemma vars_sst_Cons: "vars_sst (a#A) = vars_sstp a ∪ vars_sst A"
by (simp add: vars_sst_def)

lemma fv_sst_Cons: "fv_sst (a#A) = fv_sstp a ∪ fv_sst A"
unfolding fv_sst_def by simp

lemma bvars_sst_Cons: "bvars_sst (a#A) = set (bvars_sstp a) ∪ bvars_sst A"
unfolding bvars_sst_def by auto

lemma vars_sst_Cons'[simp]:
  "vars_sst (send⟨t⟩#A) = vars_sstp (send⟨t⟩) ∪ vars_sst A"
  "vars_sst (receive⟨t⟩#A) = vars_sstp (receive⟨t⟩) ∪ vars_sst A"
  "vars_sst (⟨a: t ≐ s⟩#A) = vars_sstp (⟨a: t ≐ s⟩) ∪ vars_sst A"
  "vars_sst (insert⟨t,s⟩#A) = vars_sstp (insert⟨t,s⟩) ∪ vars_sst A"
  "vars_sst (delete⟨t,s⟩#A) = vars_sstp (delete⟨t,s⟩) ∪ vars_sst A"
  "vars_sst (⟨a: t ∈ s⟩#A) = vars_sstp (⟨a: t ∈ s⟩) ∪ vars_sst A"
  "vars_sst (∀X⟨∨≠: F ∨∉: G⟩#A) = vars_sstp (∀X⟨∨≠: F ∨∉: G⟩) ∪ vars_sst A"
by (simp_all add: vars_sst_def)

lemma vars_sstp_is_fv_sstp_bvars_sstp:
  fixes x::"('a,'b) stateful_strand_step"
```

```
  shows "vars_sstp x = fv_sstp x ∪ set (bvars_sstp x)"
proof (cases x)
  case (NegChecks X F G) thus ?thesis by (induct F) force+
qed simp_all

lemma vars_sst_is_fv_sst_bvars_sst:
  fixes S::"('a,'b) stateful_strand"
  shows "vars_sst S = fv_sst S ∪ bvars_sst S"
proof (induction S)
  case (Cons x S) thus ?case
    using vars_sstp_is_fv_sstp_bvars_sstp[of x]
    by (auto simp add: vars_sst_def)
qed simp

lemma vars_sstp_NegCheck[simp]:
  "vars_sstp (∀X⟨∨≠: F ∨∉: G⟩) = set X ∪ fv_pairs F ∪ fv_pairs G"
by (simp_all add: sup_commute sup_left_commute vars_sstp_is_fv_sstp_bvars_sstp)

lemma bvars_sstp_NegCheck[simp]:
  "bvars_sstp (∀X⟨∨≠: F ∨∉: G⟩) = X"
  "set (bvars_sstp (∀[]⟨∨≠: F ∨∉: G⟩)) = {}"
by simp_all

lemma fv_sstp_NegCheck[simp]:
  "fv_sstp (∀X⟨∨≠: F ∨∉: G⟩) = fv_pairs F ∪ fv_pairs G - set X"
  "fv_sstp (∀[]⟨∨≠: F ∨∉: G⟩) = fv_pairs F ∪ fv_pairs G"
  "fv_sstp (⟨t != s⟩) = fv t ∪ fv s"
  "fv_sstp (⟨t not in s⟩) = fv t ∪ fv s"
by simp_all

lemma fv_sst_append[simp]: "fv_sst (A@B) = fv_sst A ∪ fv_sst B"
by simp

lemma bvars_sst_append[simp]: "bvars_sst (A@B) = bvars_sst A ∪ bvars_sst B"
by auto

lemma fv_sstp_is_subterm_trms_sstp:
  assumes "x ∈ fv_sstp a"
  shows "Var x ∈ subterms_set (trms_sstp a)"
using assms var_is_subterm
proof (cases a)
  case (NegChecks X F F')
  hence "x ∈ fv_pairs F ∪ fv_pairs F' - set X" using assms by simp
  thus ?thesis using NegChecks var_is_subterm by fastforce
qed force+

lemma fv_sst_is_subterm_trms_sst: "x ∈ fv_sst A ⟹ Var x ∈ subterms_set (trms_sst A)"
proof (induction A)
  case (Cons a A) thus ?case using fv_sstp_is_subterm_trms_sstp by (cases "x ∈ fv_sst A") auto
qed simp

lemma var_subterm_trms_sstp_is_vars_sstp:
  assumes "Var x ∈ subterms_set (trms_sstp a)"
  shows "x ∈ vars_sstp a"
using assms vars_iff_subtermeq
proof (cases a)
  case (NegChecks X F F')
  hence "Var x ∈ subterms_set (trms_pairs F ∪ trms_pairs F')" using assms by simp
  thus ?thesis using NegChecks vars_iff_subtermeq by force
qed force+

lemma var_subterm_trms_sst_is_vars_sst: "Var x ∈ subterms_set (trms_sst A) ⟹ x ∈ vars_sst A"
proof (induction A)
```

```
  case (Cons a A)
  show ?case
  proof (cases "Var x ∈ subterms_set (trms_sst A)")
    case True thus ?thesis using Cons.IH by (simp add: vars_sst_def)
  next
    case False thus ?thesis
      using Cons.prems var_subterm_trms_sstp_is_vars_sstp
      by (fastforce simp add: vars_sst_def)
  qed
qed simp

lemma var_trms_sst_is_vars_sst: "Var x ∈ trms_sst A ⟹ x ∈ vars_sst A"
by (meson var_subterm_trms_sst_is_vars_sst UN_I term.order_refl)

lemma ik_sst_trms_sst_subset: "ik_sst A ⊆ trms_sst A"
by (force simp add: ik_sst_def)

lemma var_subterm_ik_sst_is_vars_sst: "Var x ∈ subterms_set (ik_sst A) ⟹ x ∈ vars_sst A"
using var_subterm_trms_sst_is_vars_sst ik_sst_trms_sst_subset by fast

lemma var_subterm_ik_sst_is_fv_sst:
  assumes "Var x ∈ subterms_set (ik_sst A)"
  shows "x ∈ fv_sst A"
proof -
  obtain t where t: "Receive t ∈ set A" "Var x ⊑ t" using assms unfolding ik_sst_def by moura
  hence "fv t ⊆ fv_sst A" unfolding fv_sst_def by force
  thus ?thesis using t(2) by (meson contra_subsetD subterm_is_var)
qed

lemma fv_ik_sst_is_fv_sst:
  assumes "x ∈ fv_set (ik_sst A)"
  shows "x ∈ fv_sst A"
using var_subterm_ik_sst_is_fv_sst assms var_is_subterm by fastforce

lemma fv_trms_sst_subset:
  "fv_set (trms_sst S) ⊆ vars_sst S"
  "fv_sst S ⊆ fv_set (trms_sst S)"
proof (induction S)
  case (Cons x S)
  have *: "fv_set (trms_sst (x#S)) = fv_set (trms_sstp x) ∪ fv_set (trms_sst S)"
          "fv_sst (x#S) = fv_sstp x ∪ fv_sst S" "vars_sst (x#S) = vars_sstp x ∪ vars_sst S"
    unfolding trms_sst_def fv_sst_def vars_sst_def
    by auto

  { case 1
    show ?case using Cons.IH(1)
    proof (cases x)
      case (NegChecks X F G)
      hence "trms_sstp x = trms_pairs F ∪ trms_pairs G"
            "vars_sstp x = fv_pairs F ∪ fv_pairs G ∪ set X"
        by (simp, meson vars_sstp_cases(7))
      hence "fv_set (trms_sstp x) ⊆ vars_sstp x"
        using fv_trms_pairs_is_fv_pairs[of F] fv_trms_pairs_is_fv_pairs[of G]
        by auto
      thus ?thesis
        using Cons.IH(1) *(1,3)
        by blast
    qed auto
  }

  { case 2
    show ?case using Cons.IH(2)
    proof (cases x)
```

```
      case (NegChecks X F G)
      hence "trms_sstp x = trms_pairs F ∪ trms_pairs G"
            "fv_sstp x = (fv_pairs F ∪ fv_pairs G) - set X"
        by auto
      hence "fv_sstp x ⊆ fv_set (trms_sstp x)"
        using fv_trms_pairs_is_fv_pairs[of F] fv_trms_pairs_is_fv_pairs[of G]
        by auto
      thus ?thesis
        using Cons.IH(2) *(1,2)
        by blast
    qed auto
  }
qed simp_all

lemma fv_ik_subset_fv_sst'[simp]: "fv_set (ik_sst S) ⊆ fv_sst S"
unfolding ik_sst_def by (induct S) auto

lemma fv_ik_subset_vars_sst'[simp]: "fv_set (ik_sst S) ⊆ vars_sst S"
using fv_ik_subset_fv_sst' fv_trms_sst_subset by fast

lemma ik_sst_var_is_fv: "Var x ∈ subterms_set (ik_sst A) ⟹ x ∈ fv_sst A"
by (meson fv_ik_subset_fv_sst'[of A] fv_subset_subterms subsetCE term.set_intros(3))

lemma vars_sstp_subst_cases':
  assumes x: "x ∈ vars_sstp (s ·sstp ϑ)"
  shows "x ∈ vars_sstp s ∨ x ∈ fv_set (ϑ ` vars_sstp s)"
using x vars_term_subst[of _ ϑ] vars_sstp_cases(1,2,3,4,5,6) vars_sstp_subst_cases(1,2)[of _ ϑ]
      vars_sstp_subst_cases(3,6)[of _ _ _ ϑ] vars_sstp_subst_cases(4,5)[of _ _ ϑ]
proof (cases s)
  case (NegChecks X F G)
  let ?ϑ' = "rm_vars (set X) ϑ"
  have "x ∈ fv_pairs (F ·pairs ?ϑ') ∨ x ∈ fv_pairs (G ·pairs ?ϑ') ∨ x ∈ set X"
    using vars_sstp_subst_cases(7)[of X F G ϑ] x NegChecks by simp
  hence "x ∈ fv_set (?ϑ' ` fv_pairs F) ∨ x ∈ fv_set (?ϑ' ` fv_pairs G) ∨ x ∈ set X"
    using fv_pairs_subst[of _ ?ϑ'] by blast
  hence "x ∈ fv_set (ϑ ` fv_pairs F) ∨ x ∈ fv_set (ϑ ` fv_pairs G) ∨ x ∈ set X"
    using rm_vars_fv_set_subst by fast
  thus ?thesis
    using NegChecks vars_sstp_cases(7)[of X F G]
    by auto
qed simp_all

lemma vars_sst_subst_cases:
  assumes "x ∈ vars_sst (S ·sst ϑ)"
  shows "x ∈ vars_sst S ∨ x ∈ fv_set (ϑ ` vars_sst S)"
  using assms
proof (induction S)
  case (Cons s S) thus ?case
  proof (cases "x ∈ vars_sst (S ·sst ϑ)")
    case False
    note * = subst_sst_cons[of s S ϑ] vars_sst_Cons[of "s ·sstp ϑ" "S ·sst ϑ"] vars_sst_Cons[of s S]
    have **: "x ∈ vars_sstp (s ·sstp ϑ)" using Cons.prems False * by simp
    show ?thesis using vars_sstp_subst_cases'[OF **] * by auto
  qed (auto simp add: vars_sst_def)
qed simp

lemma subset_subst_pairs_diff_exists:
  fixes I::"('a,'b) subst" and D D'::"('a,'b) dbstate"
  shows "∃Di. Di ⊆ D ∧ Di ·pset I = (D ·pset I) - D'"
by (metis (no_types, lifting) Diff_subset subset_image_iff)

lemma subset_subst_pairs_diff_exists':
  fixes I::"('a,'b) subst" and D::"('a,'b) dbstate"
```

```
  assumes "finite D"
  shows "∃Di. Di ⊆ D ∧ Di ·pset ℐ ⊆ {d ·p ℐ} ∧ d ·p ℐ ∉ (D - Di) ·pset ℐ"
using assms
proof (induction D rule: finite_induct)
  case (insert d' D)
  then obtain Di where IH: "Di ⊆ D" "Di ·pset ℐ ⊆ {d ·p ℐ}" "d ·p ℐ ∉ (D - Di) ·pset ℐ" by moura
  show ?case
  proof (cases "d' ·p ℐ = d ·p ℐ")
    case True
    hence "insert d' Di ⊆ insert d' D" "insert d' Di ·pset ℐ ⊆ {d ·p ℐ}"
          "d ·p ℐ ∉ (insert d' D - insert d' Di) ·pset ℐ"
      using IH by auto
    thus ?thesis by metis
  next
    case False
    hence "Di ⊆ insert d' D" "Di ·pset ℐ ⊆ {d ·p ℐ}"
          "d ·p ℐ ∉ (insert d' D - Di) ·pset ℐ"
      using IH by auto
    thus ?thesis by metis
  qed
qed simp

lemma stateful_strand_step_subst_inI[intro]:
  "send⟨t⟩ ∈ set A ⟹ send⟨t · ϑ⟩ ∈ set (A ·sst ϑ)"
  "receive⟨t⟩ ∈ set A ⟹ receive⟨t · ϑ⟩ ∈ set (A ·sst ϑ)"
  "⟨c: t ≐ s⟩ ∈ set A ⟹ ⟨c: (t · ϑ) ≐ (s · ϑ)⟩ ∈ set (A ·sst ϑ)"
  "insert⟨t, s⟩ ∈ set A ⟹ insert⟨t · ϑ, s · ϑ⟩ ∈ set (A ·sst ϑ)"
  "delete⟨t, s⟩ ∈ set A ⟹ delete⟨t · ϑ, s · ϑ⟩ ∈ set (A ·sst ϑ)"
  "⟨c: t ∈ s⟩ ∈ set A ⟹ ⟨c: (t · ϑ) ∈ (s · ϑ)⟩ ∈ set (A ·sst ϑ)"
  "∀X⟨∨≠: F ∨∉: G⟩ ∈ set A
    ⟹ ∀X⟨∨≠: (F ·pairs rm_vars (set X) ϑ) ∨∉: (G ·pairs rm_vars (set X) ϑ)⟩ ∈ set (A ·sst ϑ)"
  "⟨t != s⟩ ∈ set A ⟹ ⟨t · ϑ != s · ϑ⟩ ∈ set (A ·sst ϑ)"
  "⟨t not in s⟩ ∈ set A ⟹ ⟨t · ϑ not in s · ϑ⟩ ∈ set (A ·sst ϑ)"
proof (induction A)
  case (Cons a A)
  note * = subst_sst_cons[of a A ϑ]
  { case 1 thus ?case using Cons.IH(1) * by (cases a) auto }
  { case 2 thus ?case using Cons.IH(2) * by (cases a) auto }
  { case 3 thus ?case using Cons.IH(3) * by (cases a) auto }
  { case 4 thus ?case using Cons.IH(4) * by (cases a) auto }
  { case 5 thus ?case using Cons.IH(5) * by (cases a) auto }
  { case 6 thus ?case using Cons.IH(6) * by (cases a) auto }
  { case 7 thus ?case using Cons.IH(7) * by (cases a) auto }
  { case 8 thus ?case using Cons.IH(8) * by (cases a) auto }
  { case 9 thus ?case using Cons.IH(9) * by (cases a) auto }
qed simp_all

lemma stateful_strand_step_cases_subst:
  "is_Send a = is_Send (a ·sstp ϑ)"
  "is_Receive a = is_Receive (a ·sstp ϑ)"
  "is_Equality a = is_Equality (a ·sstp ϑ)"
  "is_Insert a = is_Insert (a ·sstp ϑ)"
  "is_Delete a = is_Delete (a ·sstp ϑ)"
  "is_InSet a = is_InSet (a ·sstp ϑ)"
  "is_NegChecks a = is_NegChecks (a ·sstp ϑ)"
  "is_Assignment a = is_Assignment (a ·sstp ϑ)"
  "is_Check a = is_Check (a ·sstp ϑ)"
  "is_Update a = is_Update (a ·sstp ϑ)"
by (cases a; simp_all)+

lemma stateful_strand_step_subst_inv_cases:
  "send⟨t⟩ ∈ set (S ·sst σ) ⟹ ∃t'. t = t' · σ ∧ send⟨t'⟩ ∈ set S"
  "receive⟨t⟩ ∈ set (S ·sst σ) ⟹ ∃t'. t = t' · σ ∧ receive⟨t'⟩ ∈ set S"
```

```
    "⟨c: t ≐ s⟩ ∈ set (S ·sst σ) ⟹ ∃t' s'. t = t' · σ ∧ s = s' · σ ∧ ⟨c: t' ≐ s'⟩ ∈ set S"
    "insert⟨t,s⟩ ∈ set (S ·sst σ) ⟹ ∃t' s'. t = t' · σ ∧ s = s' · σ ∧ insert⟨t',s'⟩ ∈ set S"
    "delete⟨t,s⟩ ∈ set (S ·sst σ) ⟹ ∃t' s'. t = t' · σ ∧ s = s' · σ ∧ delete⟨t',s'⟩ ∈ set S"
    "⟨c: t ∈ s⟩ ∈ set (S ·sst σ) ⟹ ∃t' s'. t = t' · σ ∧ s = s' · σ ∧ ⟨c: t' ∈ s'⟩ ∈ set S"
    "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (S ·sst σ) ⟹
      ∃F' G'. F = F' ·pairs rm_vars (set X) σ ∧ G = G' ·pairs rm_vars (set X) σ ∧
              ∀X⟨∨≠: F' ∨∉: G'⟩ ∈ set S"
proof (induction S)
  case (Cons a S)
  have *: "x ∈ set (S ·sst σ)"
    when "x ∈ set (a#S ·sst σ)" "x ≠ a ·sstp σ" for x
    using that by (simp add: subst_apply_stateful_strand_def)

  { case 1 thus ?case using Cons.IH(1)[OF *] by (cases a) auto }
  { case 2 thus ?case using Cons.IH(2)[OF *] by (cases a) auto }
  { case 3 thus ?case using Cons.IH(3)[OF *] by (cases a) auto }
  { case 4 thus ?case using Cons.IH(4)[OF *] by (cases a) auto }
  { case 5 thus ?case using Cons.IH(5)[OF *] by (cases a) auto }
  { case 6 thus ?case using Cons.IH(6)[OF *] by (cases a) auto }
  { case 7 thus ?case using Cons.IH(7)[OF *] by (cases a) auto }
qed simp_all

lemma stateful_strand_step_fv_subset_cases:
  "send⟨t⟩ ∈ set S ⟹ fv t ⊆ fvsst S"
  "receive⟨t⟩ ∈ set S ⟹ fv t ⊆ fvsst S"
  "⟨c: t ≐ s⟩ ∈ set S ⟹ fv t ∪ fv s ⊆ fvsst S"
  "insert⟨t,s⟩ ∈ set S ⟹ fv t ∪ fv s ⊆ fvsst S"
  "delete⟨t,s⟩ ∈ set S ⟹ fv t ∪ fv s ⊆ fvsst S"
  "⟨c: t ∈ s⟩ ∈ set S ⟹ fv t ∪ fv s ⊆ fvsst S"
  "∀X⟨∨≠: F ∨∉: G⟩ ∈ set S ⟹ fvpairs F ∪ fvpairs G - set X ⊆ fvsst S"
proof (induction S)
  case (Cons a S)
  { case 1 thus ?case using Cons.IH(1) by auto }
  { case 2 thus ?case using Cons.IH(2) by auto }
  { case 3 thus ?case using Cons.IH(3) by auto }
  { case 4 thus ?case using Cons.IH(4) by auto }
  { case 5 thus ?case using Cons.IH(5) by auto }
  { case 6 thus ?case using Cons.IH(6) by auto }
  { case 7 thus ?case using Cons.IH(7) by fastforce }
qed simp_all

lemma trmssst_nil[simp]:
  "trmssst [] = {}"
unfolding trmssst_def by simp

lemma trmssst_mono:
  "set M ⊆ set N ⟹ trmssst M ⊆ trmssst N"
by auto

lemma trmssst_in:
  assumes "t ∈ trmssst S"
  shows "∃a ∈ set S. t ∈ trmssstp a"
using assms unfolding trmssst_def by simp

lemma trmssst_cons: "trmssst (a#A) = trmssstp a ∪ trmssst A"
unfolding trmssst_def by force

lemma trmssst_append[simp]: "trmssst (A@B) = trmssst A ∪ trmssst B"
unfolding trmssst_def by force

lemma trmssstp_subst:
  assumes "set (bvarssstp a) ∩ subst_domain ϑ = {}"
  shows "trmssstp (a ·sstp ϑ) = trmssstp a ·set ϑ"
```

```
proof (cases a)
  case (NegChecks X F G)
  hence "rm_vars (set X) ϑ = ϑ" using assms rm_vars_apply'[of ϑ "set X"] by auto
  hence "trms_sstp (a ·_sstp ϑ) = trms_pairs (F ·_pairs ϑ) ∪ trms_pairs (G ·_pairs ϑ)"
        "trms_sstp a ·_set ϑ = (trms_pairs F ·_set ϑ) ∪ (trms_pairs G ·_set ϑ)"
    using NegChecks image_Un by simp_all
  thus ?thesis by (metis trms_pairs_subst)
qed simp_all

lemma trms_sstp_subst':
  assumes "¬is_NegChecks a"
  shows "trms_sstp (a ·_sstp ϑ) = trms_sstp a ·_set ϑ"
using assms by (cases a) simp_all

lemma trms_sstp_subst'':
  fixes t::"('a,'b) term" and δ::"('a,'b) subst"
  assumes "t ∈ trms_sstp (b ·_sstp δ)"
  shows "∃s ∈ trms_sstp b. t = s · rm_vars (set (bvars_sstp b)) δ"
proof (cases "is_NegChecks b")
  case True
  then obtain X F G where *: "b = NegChecks X F G" by (cases b) moura+
  thus ?thesis using assms trms_pairs_subst[of _ "rm_vars (set X) δ"] by auto
next
  case False
  hence "trms_sstp (b ·_sstp δ) = trms_sstp b ·_set rm_vars (set (bvars_sstp b)) δ"
    using trms_sstp_subst' bvars_sstp_NegChecks
    by fastforce
  thus ?thesis using assms by fast
qed

lemma trms_sstp_subst''':
  fixes t::"('a,'b) term" and δ ϑ::"('a,'b) subst"
  assumes "t ∈ trms_sstp (b ·_sstp δ) ·_set ϑ"
  shows "∃s ∈ trms_sstp b. t = s · rm_vars (set (bvars_sstp b)) δ ∘_s ϑ"
proof -
  obtain s where s: "s ∈ trms_sstp (b ·_sstp δ)" "t = s · ϑ" using assms by moura
  show ?thesis using trms_sstp_subst''[OF s(1)] s(2) by auto
qed

lemma trms_sst_subst:
  assumes "bvars_sst S ∩ subst_domain ϑ = {}"
  shows "trms_sst (S ·_sst ϑ) = trms_sst S ·_set ϑ"
using assms
proof (induction S)
  case (Cons a S)
  hence IH: "trms_sst (S ·_sst ϑ) = trms_sst S ·_set ϑ" and *: "set (bvars_sstp a) ∩ subst_domain ϑ = {}"
    by auto
  show ?case using trms_sstp_subst[OF *] IH by (auto simp add: subst_apply_stateful_strand_def)
qed simp

lemma trms_sst_subst_cons:
  "trms_sst (a#A ·_sst δ) = trms_sstp (a ·_sstp δ) ∪ trms_sst (A ·_sst δ)"
using subst_sst_cons[of a A δ] trms_sst_cons[of a A] trms_sst_append by simp

lemma (in intruder_model) wf_trms_trms_sstp_subst:
  assumes "wf_trms (trms_sstp a ·_set δ)"
  shows "wf_trms (trms_sstp (a ·_sstp δ))"
  using assms
proof (cases a)
  case (NegChecks X F G)
  hence *: "trms_sstp (a ·_sstp δ) =
            (trms_pairs (F ·_pairs rm_vars (set X) δ)) ∪ (trms_pairs (G ·_pairs rm_vars (set X) δ))"
    by simp
```

```
  have "trms_sstp a ·set δ = (trms_pairs F ·set δ) ∪ (trms_pairs G ·set δ)"
    using NegChecks image_Un by simp
  hence "wf_trms (trms_pairs F ·set δ)" "wf_trms (trms_pairs G ·set δ)" using * assms by auto
  hence "wf_trms (trms_pairs F ·set rm_vars (set X) δ)"
        "wf_trms (trms_pairs G ·set rm_vars (set X) δ)"
    using wf_trms_subst_rm_vars[of δ "trms_pairs F" "set X"]
          wf_trms_subst_rm_vars[of δ "trms_pairs G" "set X"]
    by fast+
  thus ?thesis
    using * trms_pairs_subst[of _ "rm_vars (set X) δ"]
    by auto
qed auto

lemma trms_sst_fv_vars_sst_subset: "t ∈ trms_sst A ⟹ fv t ⊆ vars_sst A"
proof (induction A)
  case (Cons a A) thus ?case by (cases a) auto
qed simp

lemma trms_sst_fv_subst_subset:
  assumes "t ∈ trms_sst S" "subst_domain ϑ ∩ bvars_sst S = {}"
  shows "fv (t · ϑ) ⊆ vars_sst (S ·sst ϑ)"
using assms
proof (induction S)
  case (Cons s S) show ?case
  proof (cases "t ∈ trms_sst S")
    case True
    hence "fv (t · ϑ) ⊆ vars_sst (S ·sst ϑ)" using Cons.IH Cons.prems by auto
    thus ?thesis using subst_sst_cons[of s S ϑ] unfolding vars_sst_def by auto
  next
    case False
    hence *: "t ∈ trms_sstp s" "subst_domain ϑ ∩ set (bvars_sstp s) = {}" using Cons.prems by auto
    hence "fv (t · ϑ) ⊆ vars_sstp (s ·sstp ϑ)"
    proof (cases s)
      case (NegChecks X F G)
      hence **: "t ∈ trms_pairs F ∨ t ∈ trms_pairs G" using *(1) by auto
      have ***: "rm_vars (set X) ϑ = ϑ" using *(2) NegChecks rm_vars_apply' by auto
      have "fv (t · ϑ) ⊆ fv_pairs (F ·pairs rm_vars (set X) ϑ) ∪ fv_pairs (G ·pairs rm_vars (set X) ϑ)"
        using ** *** trms_pairs_fv_subst_subset[of t _ ϑ] by auto
      thus ?thesis using *(2) using NegChecks vars_sstp_subst_cases(7)[of X F G ϑ] by blast
    qed auto
    thus ?thesis using subst_sst_cons[of s S ϑ] unfolding vars_sst_def by auto
  qed
qed simp

lemma trms_sst_fv_subst_subset':
  assumes "t ∈ subterms_set (trms_sst S)" "fv t ∩ bvars_sst S = {}" "fv (t · ϑ) ∩ bvars_sst S = {}"
  shows "fv (t · ϑ) ⊆ fv_sst (S ·sst ϑ)"
using assms
proof (induction S)
  case (Cons s S) show ?case
  proof (cases "t ∈ subterms_set (trms_sst S)")
    case True
    hence "fv (t · ϑ) ⊆ fv_sst (S ·sst ϑ)" using Cons.IH Cons.prems by auto
    thus ?thesis using subst_sst_cons[of s S ϑ] unfolding vars_sst_def by auto
  next
    case False
    hence 0: "t ∈ subterms_set (trms_sstp s)" "fv t ∩ set (bvars_sstp s) = {}"
             "fv (t · ϑ) ∩ set (bvars_sstp s) = {}"
      using Cons.prems by auto

    note 1 = UN_Un UN_insert fv_set.simps subst_apply_fv_subset subst_apply_fv_unfold
             subst_apply_term_empty sup_bot.comm_neutral fv_subterms_set fv_subset[OF 0(1)]
```

```
    note 2 = subst_apply_fv_union

    have "fv (t · ϑ) ⊆ fv_sstp (s ·_sstp ϑ)"
    proof (cases s)
      case (NegChecks X F G)
      hence 3: "t ∈ subterms_set (trms_pairs F) ∨ t ∈ subterms_set (trms_pairs G)" using 0(1) by auto
      have "t · rm_vars (set X) ϑ = t · ϑ" using 0(2) NegChecks rm_vars_ident[of t] by auto
      hence "fv (t · ϑ) ⊆ fv_pairs (F ·_pairs rm_vars (set X) ϑ) ∪ fv_pairs (G ·_pairs rm_vars (set X) ϑ)"
        using 3 trms_pairs_fv_subst_subset'[of t _ "rm_vars (set X) ϑ"] by fastforce
      thus ?thesis using 0(2,3) NegChecks fv_sstp_subst_cases(7)[of X F G ϑ] by auto
    qed (metis (no_types, lifting) 1 trms_sstp.simps(1) fv_sstp_subst_cases(1),
         metis (no_types, lifting) 1 trms_sstp.simps(2) fv_sstp_subst_cases(2),
         metis (no_types, lifting) 1 2 trms_sstp.simps(3) fv_sstp_subst_cases(3),
         metis (no_types, lifting) 1 2 trms_sstp.simps(4) fv_sstp_subst_cases(4),
         metis (no_types, lifting) 1 2 trms_sstp.simps(5) fv_sstp_subst_cases(5),
         metis (no_types, lifting) 1 2 trms_sstp.simps(6) fv_sstp_subst_cases(6))
    thus ?thesis using subst_sst_cons[of s S ϑ] unfolding fv_sst_def by auto
  qed
qed simp

lemma trms_sstp_funs_term_cases:
  assumes "t ∈ trms_sstp (s ·_sstp ϑ)" "f ∈ funs_term t"
  shows "(∃u ∈ trms_sstp s. f ∈ funs_term u) ∨ (∃x ∈ fv_sstp s. f ∈ funs_term (ϑ x))"
  using assms
proof (cases s)
  case (NegChecks X F G)
  hence "t ∈ trms_pairs (F ·_pairs rm_vars (set X) ϑ) ∨ t ∈ trms_pairs (G ·_pairs rm_vars (set X) ϑ)"
    using assms(1) by auto
  hence "(∃u∈trms_pairs F. f ∈ funs_term u) ∨ (∃x∈fv_pairs F. f ∈ funs_term (rm_vars (set X) ϑ x)) ∨
        (∃u∈trms_pairs G. f ∈ funs_term u) ∨ (∃x∈fv_pairs G. f ∈ funs_term (rm_vars (set X) ϑ x))"
    using trms_pairs_funs_term_cases[OF _ assms(2), of _ "rm_vars (set X) ϑ"] by meson
  hence "(∃u ∈ trms_pairs F ∪ trms_pairs G. f ∈ funs_term u) ∨
        (∃x ∈ fv_pairs F ∪ fv_pairs G. f ∈ funs_term (rm_vars (set X) ϑ x))"
    by blast
  thus ?thesis
  proof
    assume "∃x ∈ fv_pairs F ∪ fv_pairs G. f ∈ funs_term (rm_vars (set X) ϑ x)"
    then obtain x where x: "x ∈ fv_pairs F ∪ fv_pairs G" "f ∈ funs_term (rm_vars (set X) ϑ x)"
      by auto
    hence "x ∉ set X" "rm_vars (set X) ϑ x = ϑ x" by auto
    thus ?thesis using x by (auto simp add: assms NegChecks)
  qed (auto simp add: assms NegChecks)
qed (use assms funs_term_subst[of _ ϑ] in auto)

lemma trms_sst_funs_term_cases:
  assumes "t ∈ trms_sst (S ·_sst ϑ)" "f ∈ funs_term t"
  shows "(∃u ∈ trms_sst S. f ∈ funs_term u) ∨ (∃x ∈ fv_sst S. f ∈ funs_term (ϑ x))"
using assms(1)
proof (induction S)
  case (Cons s S) thus ?case
  proof (cases "t ∈ trms_sst (S ·_sst ϑ)")
    case False
    hence "t ∈ trms_sstp (s ·_sstp ϑ)" using Cons.prems(1) subst_sst_cons[of s S ϑ] trms_sst_cons by
auto
    thus ?thesis using trms_sstp_funs_term_cases[OF _ assms(2)] by fastforce
  qed auto
qed simp

lemma fv_sst_is_subterm_trms_sst_subst:
  assumes "x ∈ fv_sst T"
    and "bvars_sst T ∩ subst_domain ϑ = {}"
  shows "ϑ x ∈ subterms_set (trms_sst (T ·_sst ϑ))"
```

```
using trms_sst_subst[OF assms(2)] subterms_subst_subset'[of ϑ "trms_sst T"]
      fv_sst_is_subterm_trms_sst[OF assms(1)]
by (metis (no_types, lifting) image_iff subset_iff subst_apply_term.simps(1))

lemma fv_sst_subst_fv_subset:
  assumes "x ∈ fv_sst S" "x ∉ bvars_sst S" "fv (ϑ x) ∩ bvars_sst S = {}"
  shows "fv (ϑ x) ⊆ fv_sst (S ·_sst ϑ)"
using assms
proof (induction S)
  case (Cons a S)
  note 1 = fv_subst_subset[of _ _ ϑ]
  note 2 = subst_apply_fv_union subst_apply_fv_unfold[of _ ϑ] fv_subset image_eqI
  note 3 = fv_sstp_subst_cases
  note 4 = fv_sstp.simps
  from Cons show ?case
  proof (cases "x ∈ fv_sst S")
    case False
    hence 5: "x ∈ fv_sstp a" " fv (ϑ x) ∩ set (bvars_sstp a) = {}" "x ∉ set (bvars_sstp a)"
      using Cons.prems by auto
    hence "fv (ϑ x) ⊆ fv_sstp (a ·_sstp ϑ)"
    proof (cases a)
      case (NegChecks X F G)
      let ?δ = "rm_vars (set X) ϑ"
      have *: "x ∈ fv_pairs F ∪ fv_pairs G" using NegChecks 5(1) by auto
      have **: "fv (ϑ x) ∩ set X = {}" using NegChecks 5(2) by simp
      have ***: "ϑ x = ?δ x" using NegChecks 5(3) by auto
      have "fv (ϑ x) ⊆ fv_pairs (F ·_pairs ?δ) ∪ fv_pairs (G ·_pairs ?δ)"
        using fv_pairs_subst_fv_subset[of x _ ?δ] * *** by auto
      thus ?thesis using NegChecks ** by auto
    qed (metis (full_types) 1 5(1) 3(1) 4(1), metis (full_types) 1 5(1) 3(2) 4(2),
         metis (full_types) 2 5(1) 3(3) 4(3), metis (full_types) 2 5(1) 3(4) 4(4),
         metis (full_types) 2 5(1) 3(5) 4(5), metis (full_types) 2 5(1) 3(6) 4(6))
    thus ?thesis by (auto simp add: subst_sst_cons[of a S ϑ])
  qed (auto simp add: subst_sst_cons[of a S ϑ])
qed simp

lemma (in intruder_model) wf_trms_trms_sst_subst:
  assumes "wf_trms (trms_sst A ·_set δ)"
  shows "wf_trms (trms_sst (A ·_sst δ))"
  using assms
proof (induction A)
  case (Cons a A)
  hence IH: "wf_trms (trms_sst (A ·_sst δ))" and *: "wf_trms (trms_sstp a ·_set δ)" by auto
  have "wf_trms (trms_sstp (a ·_sstp δ))" by (rule wf_trms_trms_sstp_subst[OF *])
  thus ?case using IH trms_sst_subst_cons[of a A δ] by blast
qed simp

lemma fv_sst_subst_obtain_var:
  assumes "x ∈ fv_sst (S ·_sst δ)"
  shows "∃y ∈ fv_sst S. x ∈ fv (δ y)"
  using assms
proof (induction S)
  case (Cons s S)
  hence "x ∈ fv_sst (S ·_sst δ) ⟹ ∃y ∈ fv_sst S. x ∈ fv (δ y)"
    using bvars_sst_cons_subset[of S s]
    by blast
  thus ?case
  proof (cases "x ∈ fv_sst (S ·_sst δ)")
    case False
    hence *: "x ∈ fv_sstp (s ·_sstp δ)"
      using Cons.prems(1) subst_sst_cons[of s S δ]
      by fastforce
```

```
    have "∃y ∈ fv_sstp s. x ∈ fv (δ y)"
    proof (cases s)
      case (NegChecks X F G)
      hence "x ∈ fv_pairs (F ·_pairs rm_vars (set X) δ) ∨ x ∈ fv_pairs (G ·_pairs rm_vars (set X) δ)"
          and **: "x ∉ set X"
        using * by simp_all
      then obtain y where y: "y ∈ fv_pairs F ∨ y ∈ fv_pairs G" "x ∈ fv ((rm_vars (set X) δ) y)"
        using fv_pairs_subst_obtain_var[of _ _ "rm_vars (set X) δ"]
        by blast
      have "y ∉ set X"
      proof
        assume y_in: "y ∈ set X"
        hence "(rm_vars (set X) δ) y = Var y" by auto
        hence "x = y" using y(2) by simp
        thus False using ** y_in by metis
      qed
      thus ?thesis using NegChecks y by auto
    qed (use * fv_subst_obtain_var in force)+
    thus ?thesis by auto
  qed auto
qed simp

lemma fv_sst_subst_subset_range_vars_if_subset_domain:
  assumes "fv_sst S ⊆ subst_domain σ"
  shows "fv_sst (S ·_sst σ) ⊆ range_vars σ"
using assms fv_sst_subst_obtain_var[of _ S σ] subst_dom_vars_in_subst[of _ σ] subst_fv_imgI[of σ]
by (metis (no_types) in_mono subsetI)

lemma fv_sst_in_fv_trms_sst: "x ∈ fv_sst S ⟹ x ∈ fv_set (trms_sst S)"
proof (induction S)
  case (Cons s S) thus ?case
  proof (cases "x ∈ fv_sst S")
    case False
    hence *: "x ∈ fv_sstp s" using Cons.prems by simp
    hence "x ∈ fv_set (trms_sstp s)"
    proof (cases s)
      case (NegChecks X F G)
      hence "x ∈ fv_pairs F ∨ x ∈ fv_pairs G" using * by simp_all
      thus ?thesis using * fv_pairs_in_fv_trms_pairs[of x] NegChecks by auto
    qed auto
    thus ?thesis by simp
  qed simp
qed simp

lemma stateful_strand_step_subst_comp:
  assumes "range_vars δ ∩ set (bvars_sstp x) = {}"
  shows "x ·_sstp δ ∘_s ϑ = (x ·_sstp δ) ·_sstp ϑ"
proof (cases x)
  case (NegChecks X F G)
  hence *: "range_vars δ ∩ set X = {}" using assms by simp
  have "H ·_pairs rm_vars (set X) (δ ∘_s ϑ) = (H ·_pairs rm_vars (set X) δ) ·_pairs rm_vars (set X) ϑ" for H
    using pairs_subst_comp rm_vars_comp[OF *] by (induct H) (auto simp add: subst_apply_pairs_def)
  thus ?thesis using NegChecks by simp
qed simp_all

lemma stateful_strand_subst_comp:
  assumes "range_vars δ ∩ bvars_sst S = {}"
  shows "S ·_sst δ ∘_s ϑ = (S ·_sst δ) ·_sst ϑ"
using assms
proof (induction S)
  case (Cons s S)
  hence IH: "S ·_sst δ ∘_s ϑ = (S ·_sst δ) ·_sst ϑ" using Cons by auto
```

```
  have "s ·sstp δ ∘s ϑ = (s ·sstp δ) ·sstp ϑ"
    using Cons.prems stateful_strand_step_subst_comp[of δ s ϑ]
    unfolding range_vars_alt_def by auto
  thus ?case using IH by (simp add: subst_apply_stateful_strand_def)
qed simp

lemma subst_apply_bvars_disj_NegChecks:
  assumes "set X ∩ subst_domain ϑ = {}"
  shows "NegChecks X F G ·sstp ϑ = NegChecks X (F ·pairs ϑ) (G ·pairs ϑ)"
proof -
  have "rm_vars (set X) ϑ = ϑ" using assms rm_vars_apply'[of ϑ "set X"] by auto
  thus ?thesis by simp
qed

lemma subst_apply_NegChecks_no_bvars[simp]:
  "∀[]⟨∨≠: F ∨∉: F'⟩ ·sstp ϑ = ∀[]⟨∨≠: (F ·pairs ϑ) ∨∉: (F' ·pairs ϑ)⟩"
  "∀[]⟨∨≠: [] ∨∉: F'⟩ ·sstp ϑ = ∀[]⟨∨≠: [] ∨∉: (F' ·pairs ϑ)⟩"
  "∀[]⟨∨≠: F ∨∉: []⟩ ·sstp ϑ = ∀[]⟨∨≠: (F ·pairs ϑ) ∨∉: []⟩"
  "∀[]⟨∨≠: [] ∨∉: [(t,s)]⟩ ·sstp ϑ = ∀[]⟨∨≠: [] ∨∉: ([(t · ϑ,s · ϑ)])⟩" (is ?A)
  "∀[]⟨∨≠: [(t,s)] ∨∉: []⟩ ·sstp ϑ = ∀[]⟨∨≠: ([(t · ϑ,s · ϑ)]) ∨∉: []⟩" (is ?B)
by simp_all

lemma setopssst_mono:
  "set M ⊆ set N ⟹ setopssst M ⊆ setopssst N"
by (auto simp add: setopssst_def)

lemma setopssst_nil[simp]: "setopssst [] = {}"
by (simp add: setopssst_def)

lemma setopssst_cons[simp]: "setopssst (a#A) = setopssstp a ∪ setopssst A"
by (simp add: setopssst_def)

lemma setopssst_cons_subset[simp]: "setopssst A ⊆ setopssst (a#A)"
using setopssst_cons[of a A] by blast

lemma setopssst_append: "setopssst (A@B) = setopssst A ∪ setopssst B"
proof (induction A)
  case (Cons a A) thus ?case by (cases a) (auto simp add: setopssst_def)
qed (simp add: setopssst_def)

lemma setopssstp_member_iff:
  "(t,s) ∈ setopssstp x ⟷
    (x = Insert t s ∨ x = Delete t s ∨ (∃ac. x = InSet ac t s) ∨
     (∃X F F'. x = NegChecks X F F' ∧ (t,s) ∈ set F'))"
by (cases x) auto

lemma setopssst_member_iff:
  "(t,s) ∈ setopssst A ⟷
    (Insert t s ∈ set A ∨ Delete t s ∈ set A ∨ (∃ac. InSet ac t s ∈ set A) ∨
     (∃X F F'. NegChecks X F F' ∈ set A ∧ (t,s) ∈ set F'))"
  (is "?P ⟷ ?Q")
proof (induction A)
  case (Cons a A) thus ?case
  proof (cases "(t, s) ∈ setopssstp a")
    case True thus ?thesis using setopssstp_member_iff[of t s a] by auto
  qed auto
qed simp

lemma setopssstp_subst:
  assumes "set (bvarssstp a) ∩ subst_domain ϑ = {}"
  shows "setopssstp (a ·sstp ϑ) = setopssstp a ·pset ϑ"
proof (cases a)
  case (NegChecks X F G)
```

```
  hence "rm_vars (set X) ϑ = ϑ" using assms rm_vars_apply'[of ϑ "set X"] by auto
  hence "setops_sstp (a ·_sstp ϑ) = set (G ·_pairs ϑ)"
        "setops_sstp a ·_pset ϑ = set G ·_pset ϑ"
    using NegChecks image_Un by simp_all
  thus ?thesis by (simp add: subst_apply_pairs_def)
qed simp_all

lemma setops_sstp_subst':
  assumes "¬is_NegChecks a"
  shows "setops_sstp (a ·_sstp ϑ) = setops_sstp a ·_pset ϑ"
using assms by (cases a) auto

lemma setops_sstp_subst'':
  fixes t::"('a,'b) term × ('a,'b) term" and δ::"('a,'b) subst"
  assumes t: "t ∈ setops_sstp (b ·_sstp δ)"
  shows "∃s ∈ setops_sstp b. t = s ·_p rm_vars (set (bvars_sstp b)) δ"
proof (cases "is_NegChecks b")
  case True
  then obtain X F G where b: "b = NegChecks X F G" by (cases b) moura+
  hence "setops_sstp b = set G" "setops_sstp (b ·_sstp δ) = set (G ·_pairs rm_vars (set (bvars_sstp b)) δ)"
    by simp_all
  thus ?thesis using t subst_apply_pairs_pset_subst[of G] by blast
next
  case False
  hence "setops_sstp (b ·_sstp δ) = setops_sstp b ·_pset rm_vars (set (bvars_sstp b)) δ"
    using setops_sstp_subst' bvars_sstp_NegChecks by fastforce
  thus ?thesis using t by blast
qed

lemma setops_sst_subst:
  assumes "bvars_sst S ∩ subst_domain ϑ = {}"
  shows "setops_sst (S ·_sst ϑ) = setops_sst S ·_pset ϑ"
using assms
proof (induction S)
  case (Cons a S)
  have "bvars_sst S ∩ subst_domain ϑ = {}" and *: "set (bvars_sstp a) ∩ subst_domain ϑ = {}"
    using Cons.prems by auto
  hence IH: "setops_sst (S ·_sst ϑ) = setops_sst S ·_pset ϑ"
    using Cons.IH by auto
  show ?case
    using setops_sstp_subst[OF *] IH unfolding setops_sst_def
    by (auto simp add: subst_apply_stateful_strand_def)
qed (simp add: setops_sst_def)

lemma setops_sst_subst':
  fixes p::"('a,'b) term × ('a,'b) term" and δ::"('a,'b) subst"
  assumes "p ∈ setops_sst (S ·_sst δ)"
  shows "∃s ∈ setops_sst S. ∃X. set X ⊆ bvars_sst S ∧ p = s ·_p rm_vars (set X) δ"
using assms
proof (induction S)
  case (Cons a S)
  note 0 = setops_sst_cons[of a S] bvars_sst_Cons[of a S]
  note 1 = setops_sst_cons[of "a ·_sstp δ" "S ·_sst δ"] subst_sst_cons[of a S δ]
  have "p ∈ setops_sst (S ·_sst δ) ∨ p ∈ setops_sstp (a ·_sstp δ)" using Cons.prems 1 by auto
  thus ?case
  proof
    assume *: "p ∈ setops_sstp (a ·_sstp δ)"
    show ?thesis using setops_sstp_subst''[OF *] 0 by blast
  next
    assume *: "p ∈ setops_sst (S ·_sst δ)"
    show ?thesis using Cons.IH[OF *] 0 by blast
  qed
qed simp
```

4.1.3 Stateful Constraint Semantics

```
context intruder_model
begin

definition negchecks_model where
  "negchecks_model (I::('a,'b) subst) (D::('a,'b) dbstate) X F G ≡
      (∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶
              (list_ex (λf. fst f · (δ ∘s I) ≠ snd f · (δ ∘s I)) F ∨
               list_ex (λf. f ·p (δ ∘s I) ∉ D) G))"

fun strand_sem_stateful::
  "('fun,'var) terms ⇒ ('fun,'var) dbstate ⇒ ('fun,'var) stateful_strand ⇒ ('fun,'var) subst ⇒
bool"
  ("⟦_; _; _⟧s")
where
  "⟦M; D; []⟧s = (λI. True)"
| "⟦M; D; Send t#S⟧s = (λI. M ⊢ t · I ∧ ⟦M; D; S⟧s I)"
| "⟦M; D; Receive t#S⟧s = (λI. ⟦insert (t · I) M; D; S⟧s I)"
| "⟦M; D; Equality _ t t'#S⟧s = (λI. t · I = t' · I ∧ ⟦M; D; S⟧s I)"
| "⟦M; D; Insert t s#S⟧s = (λI. ⟦M; insert ((t,s) ·p I) D; S⟧s I)"
| "⟦M; D; Delete t s#S⟧s = (λI. ⟦M; D - {(t,s) ·p I}; S⟧s I)"
| "⟦M; D; InSet _ t s#S⟧s = (λI. (t,s) ·p I ∈ D ∧ ⟦M; D; S⟧s I)"
| "⟦M; D; NegChecks X F F'#S⟧s = (λI. negchecks_model I D X F F' ∧ ⟦M; D; S⟧s I)"


lemmas strand_sem_stateful_induct =
  strand_sem_stateful.induct[case_names Nil ConsSnd ConsRcv ConsEq
                                        ConsIns ConsDel ConsIn ConsNegChecks]

abbreviation constr_sem_stateful (infix "⊨s" 91) where "I ⊨s A ≡ ⟦{}; {}; A⟧s I"

lemma stateful_strand_sem_NegChecks_no_bvars:
  "⟦M; D; [⟨t not in s⟩]⟧s I ⟹ (t · I, s · I) ∉ D"
  "⟦M; D; [⟨t != s⟩]⟧s I ⟹ t · I ≠ s · I"
by (simp_all add: negchecks_model_def empty_dom_iff_empty_subst)

lemma strand_sem_ik_mono_stateful:
  "⟦M; D; A⟧s I ⟹ ⟦M ∪ M'; D; A⟧s I"
using ideduct_mono by (induct A arbitrary: M M' D rule: strand_sem_stateful.induct) force+

lemma strand_sem_append_stateful:
  "⟦M; D; A@B⟧s I ⟷ ⟦M; D; A⟧s I ∧ ⟦M ∪ (iksst A ·set I); dbupdsst A I D; B⟧s I"
  (is "?P ⟷ ?Q ∧ ?R")
proof -
  have 1: "?P ⟹ ?Q" by (induct A rule: strand_sem_stateful.induct) auto

  have 2: "?P ⟹ ?R"
  proof (induction A arbitrary: M D B)
    case (Cons a A) thus ?case
    proof (cases a)
      case (Receive t)
      have "insert (t · I) (M ∪ (iksst A ·set I)) = M ∪ (iksst (a#A) ·set I)"
           "dbupdsst A I D = dbupdsst (a#A) I D"
        using Receive by (auto simp add: iksst_def)
      thus ?thesis using Cons Receive by force
    qed (auto simp add: iksst_def)
  qed (simp add: iksst_def)

  have 3: "?Q ⟹ ?R ⟹ ?P"
  proof (induction A arbitrary: M D)
    case (Cons a A) thus ?case
    proof (cases a)
```

```
      case (Receive t)
      have "insert (t · I) (M ∪ (ik_sst A ·_set I)) = M ∪ (ik_sst (a#A) ·_set I)"
           "dbupd_sst A I D = dbupd_sst (a#A) I D"
        using Receive by (auto simp add: ik_sst_def)
      thus ?thesis using Cons Receive by simp
    qed (auto simp add: ik_sst_def)
  qed (simp add: ik_sst_def)

  show ?thesis by (metis 1 2 3)
qed

lemma negchecks_model_db_subset:
  fixes F F'::"(('a,'b) term × ('a,'b) term) list"
  assumes "D' ⊆ D"
  and "negchecks_model I D X F F'"
  shows "negchecks_model I D' X F F'"
proof -
  have "list_ex (λf. f ·_p δ ∘_s I ∉ D') F'"
    when "list_ex (λf. f ·_p δ ∘_s I ∉ D) F'"
    for δ::"('a,'b) subst"
    using Bex_set[of F' "λf. f ·_p δ ∘_s I ∉ D'"]
          Bex_set[of F' "λf. f ·_p δ ∘_s I ∉ D"]
          that assms(1)
    by blast
  thus ?thesis using assms(2) by (auto simp add: negchecks_model_def)
qed

lemma negchecks_model_db_supset:
  fixes F F'::"(('a,'b) term × ('a,'b) term) list"
  assumes "D' ⊆ D"
    and "∀f ∈ set F'. ∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ f ·_p (δ ∘_s I) ∉ D -
D'"
    and "negchecks_model I D' X F F'"
  shows "negchecks_model I D X F F'"
proof -
  have "list_ex (λf. f ·_p δ ∘_s I ∉ D) F'"
    when "list_ex (λf. f ·_p δ ∘_s I ∉ D') F'" "subst_domain δ = set X ∧ ground (subst_range δ)"
    for δ::"('a,'b) subst"
    using Bex_set[of F' "λf. f ·_p δ ∘_s I ∉ D'"]
          Bex_set[of F' "λf. f ·_p δ ∘_s I ∉ D"]
          that assms(1,2)
    by blast
  thus ?thesis using assms(3) by (auto simp add: negchecks_model_def)
qed

lemma negchecks_model_subst:
  fixes F F'::"(('a,'b) term × ('a,'b) term) list"
  assumes "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
  shows "negchecks_model (δ ∘_s ϑ) D X F F' ⟷ negchecks_model ϑ D X (F ·_pairs δ) (F' ·_pairs δ)"
proof -
  have 0: "σ ∘_s (δ ∘_s ϑ) = δ ∘_s (σ ∘_s ϑ)"
    when σ: "subst_domain σ = set X" "ground (subst_range σ)" for σ
    by (metis (no_types, lifting) σ subst_compose_assoc assms(1) inf_sup_aci(1)
            subst_comp_eq_if_disjoint_vars sup_inf_absorb range_vars_alt_def)

  { fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. fst f · (σ ∘_s (δ ∘_s ϑ)) ≠ snd f · (σ ∘_s (δ ∘_s ϑ))) F"
    obtain f where f: "f ∈ set F" "fst f · σ ∘_s (δ ∘_s ϑ) ≠ snd f · σ ∘_s (δ ∘_s ϑ)"
      using * by (induct F) auto
    hence "(fst f · δ) · σ ∘_s ϑ ≠ (snd f · δ) · σ ∘_s ϑ" using 0[OF σ] by simp
    moreover have "(fst f · δ, snd f · δ) ∈ set (F ·_pairs δ)"
      using f(1) by (auto simp add: subst_apply_pairs_def)
```

```
    ultimately have "list_ex (λf. fst f · (σ ∘s ϑ) ≠ snd f · (σ ∘s ϑ)) (F ·pairs δ)"
      using f(1) Bex_set by fastforce
  } moreover {
    fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. f ·p σ ∘s (δ ∘s ϑ) ∉ D) F'"
    obtain f where f: "f ∈ set F'" "f ·p σ ∘s (δ ∘s ϑ) ∉ D"
      using * by (induct F') auto
    hence "f ·p δ ·p σ ∘s ϑ ∉ D" using 0[OF σ] by (metis subst_pair_compose)
    moreover have "f ·p δ ∈ set (F' ·pairs δ)"
      using f(1) by (auto simp add: subst_apply_pairs_def)
    ultimately have "list_ex (λf. f ·p σ ∘s ϑ ∉ D) (F' ·pairs δ)"
      using f(1) Bex_set by fastforce
  } moreover {
    fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. fst f · (σ ∘s ϑ) ≠ snd f · (σ ∘s ϑ)) (F ·pairs δ)"
    obtain f where f: "f ∈ set (F ·pairs δ)" "fst f · σ ∘s ϑ ≠ snd f · σ ∘s ϑ"
      using * by (induct F) (auto simp add: subst_apply_pairs_def)
    then obtain g where g: "g ∈ set F" "f = g ·p δ" by (auto simp add: subst_apply_pairs_def)
    have "fst g · σ ∘s (δ ∘s ϑ) ≠ snd g · σ ∘s (δ ∘s ϑ)"
      using f(2) g 0[OF σ] by (simp add: prod.case_eq_if)
    hence "list_ex (λf. fst f · (σ ∘s (δ ∘s ϑ)) ≠ snd f · (σ ∘s (δ ∘s ϑ))) F"
      using g Bex_set by fastforce
  } moreover {
    fix σ::"('a,'b) subst" and t t'
    assume σ: "subst_domain σ = set X" "ground (subst_range σ)"
        and *: "list_ex (λf. f ·p (σ ∘s ϑ) ∉ D) (F' ·pairs δ)"
    obtain f where f: "f ∈ set (F' ·pairs δ)" "f ·p σ ∘s ϑ ∉ D"
      using * by (induct F') (auto simp add: subst_apply_pairs_def)
    then obtain g where g: "g ∈ set F'" "f = g ·p δ" by (auto simp add: subst_apply_pairs_def)
    have "g ·p σ ∘s (δ ∘s ϑ) ∉ D"
      using f(2) g 0[OF σ] by (simp add: prod.case_eq_if)
    hence "list_ex (λf. f ·p (σ ∘s (δ ∘s ϑ)) ∉ D) F'"
      using g Bex_set by fastforce
  } ultimately show ?thesis using assms unfolding negchecks_model_def by blast
qed

lemma strand_sem_subst_stateful:
  fixes δ::"('fun,'var) subst"
  assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarssst S = {}"
  shows "⟦M; D; S⟧s (δ ∘s ϑ) ⟷ ⟦M; D; S ·sst δ⟧s ϑ"
proof
  note [simp] = subst_sst_cons[of _ _ δ] subst_subst_compose[of _ δ ϑ]

  have "(subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"
    when δ: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
      and γ: "subst_domain γ = set X" "ground (subst_range γ)"
    for X and γ::"('fun,'var) subst"
    using δ γ unfolding range_vars_alt_def by auto
  hence 0: "γ ∘s δ = δ ∘s γ"
    when δ: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
      and γ: "subst_domain γ = set X" "ground (subst_range γ)"
    for γ X
    by (metis δ γ subst_comp_eq_if_disjoint_vars)

  show "⟦M; D; S⟧s (δ ∘s ϑ) ⟹ ⟦M; D; S ·sst δ⟧s ϑ" using assms
  proof (induction S arbitrary: M D rule: strand_sem_stateful_induct)
    case (ConsNegChecks M D X F F' S)
    hence *: "⟦M; D; S ·sst δ⟧s ϑ" and **: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
      unfolding bvarssst_def negchecks_model_def by (force, auto)
    have "negchecks_model (δ ∘s ϑ) D X F F'" using ConsNegChecks by auto
    hence "negchecks_model ϑ D X (F ·pairs δ) (F' ·pairs δ)"
```

```
      using 0[OF **] negchecks_model_subst[OF **] by blast
    moreover have "rm_vars (set X) δ = δ" using ConsNegChecks.prems(2) by force
    ultimately show ?case using * by auto
  qed simp_all

  show "⟦M; D; S ·sst δ⟧s ϑ ⟹ ⟦M; D; S⟧s (δ ∘s ϑ)" using assms
  proof (induction S arbitrary: M D rule: strand_sem_stateful_induct)
    case (ConsNegChecks M D X F F' S)
    have δ: "rm_vars (set X) δ = δ" using ConsNegChecks.prems(2) by force
    hence *: "⟦M; D; S⟧s (δ ∘s ϑ)" and **: "(subst_domain δ ∪ range_vars δ) ∩ set X = {}"
      using ConsNegChecks unfolding bvarssst_def negchecks_model_def by auto
    have "negchecks_model ϑ D X (F ·pairs δ) (F' ·pairs δ)"
      using ConsNegChecks.prems(1) δ by (auto simp add: subst_compose_assoc negchecks_model_def)
    hence "negchecks_model (δ ∘s ϑ) D X F F'"
      using 0[OF **] negchecks_model_subst[OF **] by blast
    thus ?case using * by auto
  qed simp_all
qed

end
```

4.1.4 Well-Formedness Lemmata

```
lemma wfvarsoccsst_subset_wfrestrictedvarssst[simp]:
  "wfvarsoccssst S ⊆ wfrestrictedvarssst S"
by (induction S)
   (auto simp add: wfrestrictedvarssst_def wfvarsoccssst_def
         split: stateful_strand_step.split poscheckvariant.split)

lemma wfvarsoccssst_append: "wfvarsoccssst (S@S') = wfvarsoccssst S ∪ wfvarsoccssst S'"
by (simp add: wfvarsoccssst_def)

lemma wfrestrictedvarssst_union[simp]:
  "wfrestrictedvarssst (S@T) = wfrestrictedvarssst S ∪ wfrestrictedvarssst T"
by (simp add: wfrestrictedvarssst_def)

lemma wfrestrictedvarssst_singleton:
  "wfrestrictedvarssst [s] = wfrestrictedvarssstp s"
by (simp add: wfrestrictedvarssst_def)

lemma wfsst_prefix[dest]: "wf'sst V (S@S') ⟹ wf'sst V S"
by (induct S rule: wf'sst.induct) auto

lemma wfsst_vars_mono: "wf'sst V S ⟹ wf'sst (V ∪ W) S"
proof (induction S arbitrary: V)
  case (Cons x S) thus ?case
  proof (cases x)
    case (Send t)
    hence "wf'sst (V ∪ fv t ∪ W) S" using Cons.prems(1) Cons.IH by simp
    thus ?thesis using Send by (simp add: sup_commute sup_left_commute)
  next
    case (Equality a t t')
    show ?thesis
    proof (cases a)
      case Assign
      hence "wf'sst (V ∪ fv t ∪ W) S" "fv t' ⊆ V ∪ W" using Equality Cons.prems(1) Cons.IH by auto
      thus ?thesis using Equality Assign by (simp add: sup_commute sup_left_commute)
    next
      case Check thus ?thesis using Equality Cons by auto
    qed
  next
    case (InSet a t t')
    show ?thesis
```

```
    proof (cases a)
      case Assign
      hence "wf'sst (V ∪ fv t ∪ fv t' ∪ W) S" using InSet Cons.prems(1) Cons.IH by auto
      thus ?thesis using InSet Assign by (simp add: sup_commute sup_left_commute)
    next
      case Check thus ?thesis using InSet Cons by auto
    qed
  qed auto
qed simp

lemma wfsstI[intro]: "wfrestrictedvarssst S ⊆ V ⟹ wf'sst V S"
proof (induction S)
  case (Cons x S) thus ?case
  proof (cases x)
    case (Send t)
    hence "wf'sst V S" "V ∪ fv t = V"
      using Cons
      unfolding wfrestrictedvarssst_def
      by auto
    thus ?thesis using Send by simp
  next
    case (Equality a t t')
    show ?thesis
    proof (cases a)
      case Assign
      hence "wf'sst V S" "fv t' ⊆ V"
        using Equality Cons
        unfolding wfrestrictedvarssst_def
        by auto
      thus ?thesis using wfsst_vars_mono Equality Assign by simp
    next
      case Check
      thus ?thesis
        using Equality Cons
        unfolding wfrestrictedvarssst_def
        by auto
    qed
  next
    case (InSet a t t')
    show ?thesis
    proof (cases a)
      case Assign
      hence "wf'sst V S" "fv t ∪ fv t' ⊆ V"
        using InSet Cons
        unfolding wfrestrictedvarssst_def
        by auto
      thus ?thesis using wfsst_vars_mono InSet Assign by (simp add: Un_assoc)
    next
      case Check
      thus ?thesis
        using InSet Cons
        unfolding wfrestrictedvarssst_def
        by auto
    qed
  qed (simp_all add: wfrestrictedvarssst_def)
qed (simp add: wfrestrictedvarssst_def)

lemma wfsstI'[intro]:
  assumes "⋃((λx. case x of
            Receive t ⇒ fv t
          | Equality Assign _ t' ⇒ fv t'
          | Insert t t' ⇒ fv t ∪ fv t'
          | _ ⇒ {}) ` set S) ⊆ V"
```

```
  shows "wf'sst V S"
using assms
proof (induction S)
  case (Cons x S) thus ?case
  proof (cases x)
    case (Equality a t t')
    thus ?thesis using Cons by (cases a) (auto simp add: wfsst_vars_mono)
  next
    case (InSet a t t')
    thus ?thesis using Cons by (cases a) (auto simp add: wfsst_vars_mono Un_assoc)
  qed (simp_all add: wfsst_vars_mono)
qed simp

lemma wfsst_append_exec: "wf'sst V (S@S') ⟹ wf'sst (V ∪ wfvarsoccssst S) S'"
proof (induction S arbitrary: V)
  case (Cons x S V) thus ?case
  proof (cases x)
    case (Send t)
    hence "wf'sst (V ∪ fv t ∪ wfvarsoccssst S) S'" using Cons.prems Cons.IH by simp
    thus ?thesis using Send unfolding wfvarsoccssst_def by (auto simp add: sup_assoc)
  next
    case (Equality a t t') show ?thesis
    proof (cases a)
      case Assign
      hence "wf'sst (V ∪ fv t ∪ wfvarsoccssst S) S'" using Equality Cons.prems Cons.IH by auto
      thus ?thesis using Equality Assign unfolding wfvarsoccssst_def by (auto simp add: sup_assoc)
    next
      case Check
      hence "wf'sst (V ∪ wfvarsoccssst S) S'" using Equality Cons.prems Cons.IH by auto
      thus ?thesis using Equality Check unfolding wfvarsoccssst_def by (auto simp add: sup_assoc)
    qed
  next
    case (InSet a t t') show ?thesis
    proof (cases a)
      case Assign
      hence "wf'sst (V ∪ fv t ∪ fv t' ∪ wfvarsoccssst S) S'" using InSet Cons.prems Cons.IH by auto
      thus ?thesis using InSet Assign unfolding wfvarsoccssst_def by (auto simp add: sup_assoc)
    next
      case Check
      hence "wf'sst (V ∪ wfvarsoccssst S) S'" using InSet Cons.prems Cons.IH by auto
      thus ?thesis using InSet Check unfolding wfvarsoccssst_def by (auto simp add: sup_assoc)
    qed
  qed (auto simp add: wfvarsoccssst_def)
qed (simp add: wfvarsoccssst_def)

lemma wfsst_append:
  "wf'sst X S ⟹ wf'sst Y T ⟹ wf'sst (X ∪ Y) (S@T)"
proof (induction X S rule: wf'sst.induct)
  case 1 thus ?case by (metis wfsst_vars_mono Un_commute append_Nil)
next
  case 3 thus ?case by (metis append_Cons Un_commute Un_assoc wf'sst.simps(3))
next
  case (4 V t t' S)
  hence *: "fv t' ⊆ V" and "wf'sst (V ∪ fv t ∪ Y) (S @ T)" by simp_all
  hence "wf'sst (V ∪ Y ∪ fv t) (S @ T)" by (metis Un_commute Un_assoc)
  thus ?case using * by auto
next
  case (8 V t t' S)
  hence "wf'sst (V ∪ fv t ∪ fv t' ∪ Y) (S @ T)" by simp_all
  hence "wf'sst (V ∪ Y ∪ fv t ∪ fv t') (S @ T)" by (metis Un_commute Un_assoc)
  thus ?case by auto
qed auto
```

```
lemma wf_sst_append_suffix:
  "wf'_sst V S ⟹ wfrestrictedvars_sst S' ⊆ wfrestrictedvars_sst S ∪ V ⟹ wf'_sst V (S@S')"
proof (induction V S rule: wf'_sst.induct)
  case (2 V t S)
  hence *: "fv t ⊆ V" "wf'_sst V S" by simp_all
  hence "wfrestrictedvars_sst S' ⊆ wfrestrictedvars_sst S ∪ V"
    using "2.prems"(2) unfolding wfrestrictedvars_sst_def by auto
  thus ?case using "2.IH" * by simp
next
  case (3 V t S)
  hence *: "wf'_sst (V ∪ fv t) S" by simp_all
  hence "wfrestrictedvars_sst S' ⊆ wfrestrictedvars_sst S ∪ (V ∪ fv t)"
    using "3.prems"(2) unfolding wfrestrictedvars_sst_def by auto
  thus ?case using "3.IH" * by simp
next
  case (4 V t t' S)
  hence *: "fv t' ⊆ V" "wf'_sst (V ∪ fv t) S" by simp_all
  moreover have "vars_sstp (⟨t := t'⟩) = fv t ∪ fv t'"
    by simp
  moreover have "wfrestrictedvars_sst (⟨t := t'⟩#S) = fv t ∪ fv t' ∪ wfrestrictedvars_sst S"
    unfolding wfrestrictedvars_sst_def by auto
  ultimately have "wfrestrictedvars_sst S' ⊆ wfrestrictedvars_sst S ∪ (V ∪ fv t)"
    using "4.prems"(2) by blast
  thus ?case using "4.IH" * by simp
next
  case (6 V t t' S)
  hence *: "fv t ∪ fv t' ⊆ V" "wf'_sst V S" by simp_all
  moreover have "vars_sstp (insert⟨t,t'⟩) = fv t ∪ fv t'"
    by simp
  moreover have "wfrestrictedvars_sst (insert⟨t,t'⟩#S) = fv t ∪ fv t' ∪ wfrestrictedvars_sst S"
    unfolding wfrestrictedvars_sst_def by auto
  ultimately have "wfrestrictedvars_sst S' ⊆ wfrestrictedvars_sst S ∪ V"
    using "6.prems"(2) by blast
  thus ?case using "6.IH" * by simp
next
  case (8 V t t' S)
  hence *: "wf'_sst (V ∪ fv t ∪ fv t') S" by simp_all
  moreover have "vars_sstp (select⟨t,t'⟩) = fv t ∪ fv t'"
    by simp
  moreover have "wfrestrictedvars_sst (select⟨t,t'⟩#S) = fv t ∪ fv t' ∪ wfrestrictedvars_sst S"
    unfolding wfrestrictedvars_sst_def by auto
  ultimately have "wfrestrictedvars_sst S' ⊆ wfrestrictedvars_sst S ∪ (V ∪ fv t ∪ fv t')"
    using "8.prems"(2) by blast
  thus ?case using "8.IH" * by simp
qed (simp_all add: wf_sstI wfrestrictedvars_sst_def)

lemma wf_sst_append_suffix':
  assumes "wf'_sst V S"
    and "⋃((λx. case x of
            Receive t ⇒ fv t
          | Equality Assign _ t' ⇒ fv t'
          | Insert t t' ⇒ fv t ∪ fv t'
          | _ ⇒ {}) ` set S') ⊆ wfvarsoccs_sst S ∪ V"
  shows "wf'_sst V (S@S')"
using assms
by (induction V S rule: wf'_sst.induct)
   (auto simp add: wf_sstI' wf_sst_vars_mono wfvarsoccs_sst_def)

lemma wf_sst_subst_apply:
  "wf'_sst V S ⟹ wf'_sst (fv_set (δ ` V)) (S ·_sst δ)"
proof (induction S arbitrary: V rule: wf'_sst.induct)
  case (2 V t S)
  hence "wf'_sst V S" "fv t ⊆ V" by simp_all
```

```
  hence "wf'sst (fvset (δ ' V)) (S ·sst δ)" "fv (t · δ) ⊆ fvset (δ ' V)"
    using "2.IH" subst_apply_fv_subset by simp_all
  thus ?case by (simp add: subst_apply_stateful_strand_def)
next
  case (3 V t S)
  hence "wf'sst (V ∪ fv t) S" by simp
  hence "wf'sst (fvset (δ ' (V ∪ fv t))) (S ·sst δ)" using "3.IH" by metis
  hence "wf'sst (fvset (δ ' V) ∪ fv (t · δ)) (S ·sst δ)" by (metis subst_apply_fv_union)
  thus ?case by (simp add: subst_apply_stateful_strand_def)
next
  case (4 V t t' S)
  hence "wf'sst (V ∪ fv t) S" "fv t' ⊆ V" by auto
  hence "wf'sst (fvset (δ ' (V ∪ fv t))) (S ·sst δ)" and *: "fv (t' · δ) ⊆ fvset (δ ' V)"
    using "4.IH" subst_apply_fv_subset by force+
  hence "wf'sst (fvset (δ ' V) ∪ fv (t · δ)) (S ·sst δ)" by (metis subst_apply_fv_union)
  thus ?case using * by (simp add: subst_apply_stateful_strand_def)
next
  case (6 V t t' S)
  hence "wf'sst V S" "fv t ∪ fv t' ⊆ V" by auto
  hence "wf'sst (fvset (δ ' V)) (S ·sst δ)" "fv (t · δ) ⊆ fvset (δ ' V)" "fv (t' · δ) ⊆ fvset (δ ' V)"
    using "6.IH" subst_apply_fv_subset by force+
  thus ?case by (simp add: sup_assoc subst_apply_stateful_strand_def)
next
  case (8 V t t' S)
  hence "wf'sst (V ∪ fv t ∪ fv t') S" by auto
  hence "wf'sst (fvset (δ ' (V ∪ fv t ∪ fv t'))) (S ·sst δ)"
    using "8.IH" subst_apply_fv_subset by force
  hence "wf'sst (fvset (δ ' V) ∪ fv (t · δ) ∪ fv (t' · δ)) (S ·sst δ)" by (metis subst_apply_fv_union)
  thus ?case by (simp add: subst_apply_stateful_strand_def)
qed (auto simp add: subst_apply_stateful_strand_def)

end
```

4.2 Extending the Typing Result to Stateful Constraints (Stateful_Typing)

```
theory Stateful_Typing
imports Typing_Result Stateful_Strands
begin
```

Locale setup

```
locale stateful_typed_model = typed_model arity public Ana Γ
  for arity::"'fun ⇒ nat"
    and public::"'fun ⇒ bool"
    and Ana::"('fun,'var) term ⇒ (('fun,'var) term list × ('fun,'var) term list)"
    and Γ::"('fun,'var) term ⇒ ('fun,'atom::finite) term_type"
  +
  fixes Pair::"'fun"
  assumes Pair_arity: "arity Pair = 2"
  and Ana_subst': "⋀f T δ K M. Ana (Fun f T) = (K,M) ⟹ Ana (Fun f T · δ) = (K ·list δ,M ·list δ)"
begin

lemma Ana_invar_subst'[simp]: "Ana_invar_subst 𝒮"
using Ana_subst' unfolding Ana_invar_subst_def by force

definition pair where
  "pair d ≡ case d of (t,t') ⇒ Fun Pair [t,t']"

fun trpairs::
  "(('fun,'var) term × ('fun,'var) term) list ⇒
   ('fun,'var) dbstatelist ⇒
   (('fun,'var) term × ('fun,'var) term) list list"
```

```
where
  "tr_pairs [] D = [[]]"
| "tr_pairs ((s,t)#F) D =
    concat (map (λd. map ((#) (pair (s,t), pair d)) (tr_pairs F D)) D)"
```

A translation/reduction tr from stateful constraints to (lists of) "non-stateful" constraints. The output represents a finite disjunction of constraints whose models constitute exactly the models of the input constraint. The typing result for "non-stateful" constraints is later lifted to the stateful setting through this reduction procedure.

```
fun tr::"('fun,'var) stateful_strand ⇒ ('fun,'var) dbstatelist ⇒ ('fun,'var) strand list"
where
  "tr [] D = [[]]"
| "tr (send⟨t⟩#A) D = map ((#) (send⟨t⟩_st)) (tr A D)"
| "tr (receive⟨t⟩#A) D = map ((#) (receive⟨t⟩_st)) (tr A D)"
| "tr (⟨ac: t ≐ t'⟩#A) D = map ((#) (⟨ac: t ≐ t'⟩_st)) (tr A D)"
| "tr (insert⟨t,s⟩#A) D = tr A (List.insert (t,s) D)"
| "tr (delete⟨t,s⟩#A) D =
    concat (map (λDi. map (λB. (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩_st) Di)@
                              (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩_st) [d←D. d ∉ set Di])@B)
                          (tr A [d←D. d ∉ set Di]))
                 (subseqs D))"
| "tr (⟨ac: t ∈ s⟩#A) D =
    concat (map (λB. map (λd. ⟨ac: (pair (t,s)) ≐ (pair d)⟩_st#B) D) (tr A D))"
| "tr (∀X⟨∨≠: F ∨∉: F'⟩#A) D =
    map ((@) (map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' D))) (tr A D)"
```

Type-flaw resistance of stateful constraint steps

```
fun tfr_sstp where
  "tfr_sstp (Equality _ t t') = ((∃δ. Unifier δ t t') ⟶ Γ t = Γ t')"
| "tfr_sstp (NegChecks X F F') = (
    (F' = [] ∧ (∀x ∈ fv_pairs F-set X. ∃a. Γ (Var x) = TAtom a)) ∨
    (∀f T. Fun f T ∈ subterms_set (trms_pairs F ∪ pair ` set F') ⟶
              T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)))"
| "tfr_sstp _ = True"
```

Type-flaw resistance of stateful constraints

```
definition tfr_sst where "tfr_sst S ≡ tfr_set (trms_sst S ∪ pair ` setops_sst S) ∧ list_all tfr_sstp S"
```

4.2.1 Small Lemmata

```
lemma pair_in_pair_image_iff:
  "pair (s,t) ∈ pair ` P ⟷ (s,t) ∈ P"
unfolding pair_def by fast

lemma subst_apply_pairs_pair_image_subst:
  "pair ` set (F ·_pairs ϑ) = pair ` set F ·_set ϑ"
unfolding subst_apply_pairs_def pair_def by (induct F) auto

lemma Ana_subst_subterms_cases:
  fixes ϑ::"('fun,'var) subst"
  assumes t: "t ∈ subterms_set (M ·_set ϑ)"
    and s: "s ∈ set (snd (Ana t))"
  shows "(∃u ∈ subterms_set M. t = u · ϑ ∧ s ∈ set (snd (Ana u)) ·_set ϑ) ∨ (∃x ∈ fv_set M. t ⊑ ϑ x)"
proof (cases "t ∈ subterms_set M ·_set ϑ")
  case True
  then obtain u where u: "u ∈ subterms_set M" "t = u · ϑ" by moura
  show ?thesis
  proof (cases u)
    case (Var x)
    hence "x ∈ fv_set M" using fv_subset_subterms[OF u(1)] by simp
    thus ?thesis using u(2) Var by fastforce
  next
```

```
    case (Fun f T)
    hence "set (snd (Ana t)) = set (snd (Ana u)) ·set ϑ"
      using Ana_subst'[of f T _ _ ϑ] u(2) by (cases "Ana u") auto
    thus ?thesis using s u by blast
  qed
qed (use s t subtermsset_subst in blast)

lemma tfrsstp_alt_def:
  "list_all tfrsstp S =
    ((∀ac t t'. Equality ac t t' ∈ set S ∧ (∃δ. Unifier δ t t') ⟶ Γ t = Γ t') ∧
     (∀X F F'. NegChecks X F F' ∈ set S ⟶ (
        (F' = [] ∧ (∀x ∈ fvpairs F-set X. ∃a. Γ (Var x) = TAtom a)) ∨
        (∀f T. Fun f T ∈ subtermsset (trmspairs F ∪ pair ` set F') ⟶
                  T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)))))"
  (is "?P S = ?Q S")
proof
  show "?P S ⟹ ?Q S"
  proof (induction S)
    case (Cons x S) thus ?case by (cases x) auto
  qed simp

  show "?Q S ⟹ ?P S"
  proof (induction S)
    case (Cons x S) thus ?case by (cases x) auto
  qed simp
qed

lemma fun_pair_eq[dest]: "pair d = pair d' ⟹ d = d'"
proof -
  obtain t s t' s' where "d = (t,s)" "d' = (t',s')" by moura
  thus "pair d = pair d' ⟹ d = d'" unfolding pair_def by simp
qed

lemma fun_pair_subst: "pair d · δ = pair (d ·p δ)"
using surj_pair[of d] unfolding pair_def by force

lemma fun_pair_subst_set: "pair ` M ·set δ = pair ` (M ·pset δ)"
proof
  show "pair ` M ·set δ ⊆ pair ` (M ·pset δ)"
    using fun_pair_subst[of _ δ] by fastforce

  show "pair ` (M ·pset δ) ⊆ pair ` M ·set δ"
  proof
    fix t assume t: "t ∈ pair ` (M ·pset δ)"
    then obtain p where p: "p ∈ M" "t = pair (p ·p δ)" by blast
    thus "t ∈ pair ` M ·set δ" using fun_pair_subst[of p δ] by force
  qed
qed

lemma fun_pair_eq_subst: "pair d · δ = pair d' · ϑ ⟷ d ·p δ = d' ·p ϑ"
by (metis fun_pair_subst fun_pair_eq[of "d ·p δ" "d' ·p ϑ"])

lemma setopssst_pair_image_cons[simp]:
  "pair ` setopssst (x#S) = pair ` setopssstp x ∪ pair ` setopssst S"
  "pair ` setopssst (send⟨t⟩#S) = pair ` setopssst S"
  "pair ` setopssst (receive⟨t⟩#S) = pair ` setopssst S"
  "pair ` setopssst (⟨ac: t ≐ t'⟩#S) = pair ` setopssst S"
  "pair ` setopssst (insert⟨t,s⟩#S) = {pair (t,s)} ∪ pair ` setopssst S"
  "pair ` setopssst (delete⟨t,s⟩#S) = {pair (t,s)} ∪ pair ` setopssst S"
  "pair ` setopssst (⟨ac: t ∈ s⟩#S) = {pair (t,s)} ∪ pair ` setopssst S"
  "pair ` setopssst (∀X⟨∨≠: F ∨∉: G⟩#S) = pair ` set G ∪ pair ` setopssst S"
unfolding setopssst_def by auto
```

```
lemma setopssst_pair_image_subst_cons[simp]:
  "pair ‘ setopssst (x#S ·sst ϑ) = pair ‘ setopssstp (x ·sstp ϑ) ∪ pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (send⟨t⟩#S ·sst ϑ) = pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (receive⟨t⟩#S ·sst ϑ) = pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (⟨ac: t ≐ t'⟩#S ·sst ϑ) = pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (insert⟨t,s⟩#S ·sst ϑ) = {pair (t,s) · ϑ} ∪ pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (delete⟨t,s⟩#S ·sst ϑ) = {pair (t,s) · ϑ} ∪ pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (⟨ac: t ∈ s⟩#S ·sst ϑ) = {pair (t,s) · ϑ} ∪ pair ‘ setopssst (S ·sst ϑ)"
  "pair ‘ setopssst (∀X⟨∨≠: F ∨∉: G⟩#S ·sst ϑ) =
    pair ‘ set (G ·pairs rm_vars (set X) ϑ) ∪ pair ‘ setopssst (S ·sst ϑ)"
using subst_sst_cons[of _ S ϑ] unfolding setopssst_def pair_def by auto

lemma setopssst_are_pairs: "t ∈ pair ‘ setopssst A ⟹ ∃s s'. t = pair (s,s')"
proof (induction A)
  case (Cons a A) thus ?case
    by (cases a) (auto simp add: setopssst_def)
qed (simp add: setopssst_def)

lemma fun_pair_wftrm: "wftrm t ⟹ wftrm t' ⟹ wftrm (pair (t,t'))"
using Pair_arity unfolding wftrm_def pair_def by auto

lemma wftrms_pairs: "wftrms (trmspairs F) ⟹ wftrms (pair ‘ set F)"
using fun_pair_wftrm by blast

lemma tfrsst_Nil[simp]: "tfrsst []"
by (simp add: tfrsst_def setopssst_def)

lemma tfrsst_append: "tfrsst (A@B) ⟹ tfrsst A"
proof -
  assume assms: "tfrsst (A@B)"
  let ?M = "trmssst A ∪ pair ‘ setopssst A"
  let ?N = "trmssst (A@B) ∪ pair ‘ setopssst (A@B)"
  let ?P = "λt t'. ∀x ∈ fv t ∪ fv t'. ∃a. Γ (Var x) = Var a"
  let ?Q = "λX t t'. X = [] ∨ (∀x ∈ (fv t ∪ fv t')-set X. ∃a. Γ (Var x) = Var a)"
  have *: "SMP ?M - Var‘𝒱 ⊆ SMP ?N - Var‘𝒱" "?M ⊆ ?N"
    using SMP_mono[of ?M ?N] setopssst_append[of A B]
    by auto
  { fix s t assume **: "tfrset ?N" "s ∈ SMP ?M - Var‘𝒱" "t ∈ SMP ?M - Var‘𝒱" "(∃δ. Unifier δ s t)"
    hence "s ∈ SMP ?N - Var‘𝒱" "t ∈ SMP ?N - Var‘𝒱" using * by auto
    hence "Γ s = Γ t" using **(1,4) unfolding tfrset_def by blast
  } moreover have "∀t ∈ ?N. wftrm t ⟹ ∀t ∈ ?M. wftrm t" using * by blast
  ultimately have "tfrset ?N ⟹ tfrset ?M" unfolding tfrset_def by blast
  hence "tfrset ?M" using assms unfolding tfrsst_def by metis
  thus "tfrsst A" using assms unfolding tfrsst_def by simp
qed

lemma tfrsst_append': "tfrsst (A@B) ⟹ tfrsst B"
proof -
  assume assms: "tfrsst (A@B)"
  let ?M = "trmssst B ∪ pair ‘ setopssst B"
  let ?N = "trmssst (A@B) ∪ pair ‘ setopssst (A@B)"
  let ?P = "λt t'. ∀x ∈ fv t ∪ fv t'. ∃a. Γ (Var x) = Var a"
  let ?Q = "λX t t'. X = [] ∨ (∀x ∈ (fv t ∪ fv t')-set X. ∃a. Γ (Var x) = Var a)"
  have *: "SMP ?M - Var‘𝒱 ⊆ SMP ?N - Var‘𝒱" "?M ⊆ ?N"
    using SMP_mono[of ?M ?N] setopssst_append[of A B]
    by auto
  { fix s t assume **: "tfrset ?N" "s ∈ SMP ?M - Var‘𝒱" "t ∈ SMP ?M - Var‘𝒱" "(∃δ. Unifier δ s t)"
    hence "s ∈ SMP ?N - Var‘𝒱" "t ∈ SMP ?N - Var‘𝒱" using * by auto
    hence "Γ s = Γ t" using **(1,4) unfolding tfrset_def by blast
  } moreover have "∀t ∈ ?N. wftrm t ⟹ ∀t ∈ ?M. wftrm t" using * by blast
  ultimately have "tfrset ?N ⟹ tfrset ?M" unfolding tfrset_def by blast
  hence "tfrset ?M" using assms unfolding tfrsst_def by metis
  thus "tfrsst B" using assms unfolding tfrsst_def by simp
```

```
qed

lemma tfr_sst_cons: "tfr_sst (a#A) ⟹ tfr_sst A"
using tfr_sst_append'[of "[a]" A] by simp

lemma tfr_sstp_subst:
  assumes s: "tfr_sstp s"
    and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" "set (bvars_sstp s) ∩ range_vars ϑ = {}"
  shows "tfr_sstp (s ·_sstp ϑ)"
proof (cases s)
  case (Equality a t t')
  thus ?thesis
  proof (cases "∃δ. Unifier δ (t · ϑ) (t' · ϑ)")
    case True
    hence "∃δ. Unifier δ t t'" by (metis subst_subst_compose[of _ ϑ])
    moreover have "Γ t = Γ (t · ϑ)" "Γ t' = Γ (t' · ϑ)" by (metis wt_subst_trm''[OF assms(2)])+
    ultimately have "Γ (t · ϑ) = Γ (t' · ϑ)" using s Equality by simp
    thus ?thesis using Equality True by simp
  qed simp
next
  case (NegChecks X F G)
  let ?P = "λF G. G = [] ∧ (∀x ∈ fv_pairs F-set X. ∃a. Γ (Var x) = TAtom a)"
  let ?Q = "λF G. ∀f T. Fun f T ∈ subterms_set (trms_pairs F ∪ pair ` set G) ⟶
                          T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)"
  let ?ϑ = "rm_vars (set X) ϑ"

  have "?P F G ∨ ?Q F G" using NegChecks assms(1) by simp
  hence "?P (F ·_pairs ?ϑ) (G ·_pairs ?ϑ) ∨ ?Q (F ·_pairs ?ϑ) (G ·_pairs ?ϑ)"
  proof
    assume *: "?P F G"
    have "G ·_pairs ?ϑ = []" using * by simp
    moreover have "∃a. Γ (Var x) = TAtom a" when x: "x ∈ fv_pairs (F ·_pairs ?ϑ) - set X" for x
    proof -
      obtain t t' where t: "(t,t') ∈ set (F ·_pairs ?ϑ)" "x ∈ fv t ∪ fv t' - set X"
        using x(1) by auto
      then obtain u u' where u: "(u,u') ∈ set F" "u · ?ϑ = t" "u' · ?ϑ = t'"
        unfolding subst_apply_pairs_def by auto
      obtain y where y: "y ∈ fv u ∪ fv u' - set X" "x ∈ fv (?ϑ y)"
        using t(2) u(2,3) rm_vars_fv_obtain by fast
      hence a: "∃a. Γ (Var y) = TAtom a" using u * by auto

      have a': "Γ (Var y) = Γ (?ϑ y)"
        using wt_subst_trm''[OF wt_subst_rm_vars[OF ϑ(1), of "set X"], of "Var y"]
        by simp

      have "(∃z. ?ϑ y = Var z) ∨ (∃c. ?ϑ y = Fun c [])"
      proof (cases "?ϑ y ∈ subst_range ϑ")
        case True thus ?thesis
          using a a' ϑ(2) const_type_inv_wf
          by (cases "?ϑ y") fastforce+
      qed fastforce
      hence "?ϑ y = Var x" using y(2) by fastforce
      hence "Γ (Var x) = Γ (Var y)" using a' by simp
      thus ?thesis using a by presburger
    qed
    ultimately show ?thesis by simp
  next
    assume *: "?Q F G"
    have **: "set X ∩ range_vars ?ϑ = {}"
      using ϑ(3) NegChecks rm_vars_img_fv_subset[of "set X" ϑ] by auto
    have "?Q (F ·_pairs ?ϑ) (G ·_pairs ?ϑ)"
      using ineq_subterm_inj_cond_subst[OF ** *]
            trms_pairs_subst[of F "rm_vars (set X) ϑ"]
```

```
          subst_apply_pairs_pair_image_subst[of G "rm_vars (set X) ϑ"]
      by (metis (no_types, lifting) image_Un)
    thus ?thesis by simp
  qed
  thus ?thesis using NegChecks by simp
qed simp_all

lemma tfr_sstp_all_wt_subst_apply:
  assumes S: "list_all tfr_sstp S"
    and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" "bvars_sst S ∩ range_vars ϑ = {}"
  shows "list_all tfr_sstp (S ·_sst ϑ)"
proof -
  have "set (bvars_sstp s) ∩ range_vars ϑ = {}" when "s ∈ set S" for s
    using that ϑ(3) unfolding bvars_sst_def range_vars_alt_def by fastforce
  thus ?thesis
    using tfr_sstp_subst[OF _ ϑ(1,2)] S
    unfolding list_all_iff
    by (auto simp add: subst_apply_stateful_strand_def)
qed

lemma tr_pairs_empty_case:
  assumes "tr_pairs F D = []"
  shows "D = []" "F ≠ []"
proof -
  show "F ≠ []" using assms by (auto intro: ccontr)

  have "tr_pairs F (a#A) ≠ []" for a A
    by (induct F "a#A" rule: tr_pairs.induct) fastforce+
  thus "D = []" using assms by (cases D) simp_all
qed

lemma tr_pairs_elem_length_eq:
  assumes "G ∈ set (tr_pairs F D)"
  shows "length G = length F"
using assms by (induct F D arbitrary: G rule: tr_pairs.induct) auto

lemma tr_pairs_index:
  assumes "G ∈ set (tr_pairs F D)" "i < length F"
  shows "∃d ∈ set D. G ! i = (pair (F ! i), pair d)"
using assms
proof (induction F D arbitrary: i G rule: tr_pairs.induct)
  case (2 s t F D)
  obtain d G' where G:
      "d ∈ set D" "G' ∈ set (tr_pairs F D)"
      "G = (pair (s,t), pair d)#G'"
    using "2.prems"(1) by moura
  show ?case
    using "2.IH"[OF G(1,2)] "2.prems"(2) G(1,3)
    by (cases i) auto
qed simp

lemma tr_pairs_cons:
  assumes "G ∈ set (tr_pairs F D)" "d ∈ set D"
  shows "(pair (s,t), pair d)#G ∈ set (tr_pairs ((s,t)#F) D)"
using assms by auto

lemma tr_pairs_has_pair_lists:
  assumes "G ∈ set (tr_pairs F D)" "g ∈ set G"
  shows "∃f ∈ set F. ∃d ∈ set D. g = (pair f, pair d)"
using assms
proof (induction F D arbitrary: G rule: tr_pairs.induct)
  case (2 s t F D)
  obtain d G' where G:
```

```
      "d ∈ set D" "G' ∈ set (tr_pairs F D)"
      "G = (pair (s,t), pair d)#G'"
    using "2.prems"(1) by moura
  show ?case
    using "2.IH"[OF G(1,2)] "2.prems"(2) G(1,3)
    by (cases "g ∈ set G'") auto
qed simp

lemma tr_pairs_is_pair_lists:
  assumes "f ∈ set F" "d ∈ set D"
  shows "∃G ∈ set (tr_pairs F D). (pair f, pair d) ∈ set G"
  (is "?P F D f d")
proof -
  have "∀f ∈ set F. ∀d ∈ set D. ?P F D f d"
  proof (induction F D rule: tr_pairs.induct)
    case (2 s t F D)
    hence IH: "∀f ∈ set F. ∀d ∈ set D. ?P F D f d" by metis
    moreover have "∀d ∈ set D. ?P ((s,t)#F) D (s,t) d"
    proof
      fix d assume d: "d ∈ set D"
      then obtain G where G: "G ∈ set (tr_pairs F D)"
        using tr_pairs_empty_case(1) by force
      hence "(pair (s, t), pair d)#G ∈ set (tr_pairs ((s,t)#F) D)"
        using d by auto
      thus "?P ((s,t)#F) D (s,t) d" using d G by auto
    qed
    ultimately show ?case by fastforce
  qed simp
  thus ?thesis by (metis assms)
qed

lemma tr_pairs_db_append_subset:
  "set (tr_pairs F D) ⊆ set (tr_pairs F (D@E))" (is ?A)
  "set (tr_pairs F E) ⊆ set (tr_pairs F (D@E))" (is ?B)
proof -
  show ?A
  proof (induction F D rule: tr_pairs.induct)
    case (2 s t F D)
    show ?case
    proof
      fix G assume "G ∈ set (tr_pairs ((s,t)#F) D)"
      then obtain d G' where G':
          "d ∈ set D" "G' ∈ set (tr_pairs F D)" "G = (pair (s,t), pair d)#G'"
        by moura
      have "d ∈ set (D@E)" "G' ∈ set (tr_pairs F (D@E))" using "2.IH"[OF G'(1)] G'(1,2) by auto
      thus "G ∈ set (tr_pairs ((s,t)#F) (D@E))" using G'(3) by auto
    qed
  qed simp

  show ?B
  proof (induction F E rule: tr_pairs.induct)
    case (2 s t F E)
    show ?case
    proof
      fix G assume "G ∈ set (tr_pairs ((s,t)#F) E)"
      then obtain d G' where G':
          "d ∈ set E" "G' ∈ set (tr_pairs F E)" "G = (pair (s,t), pair d)#G'"
        by moura
      have "d ∈ set (D@E)" "G' ∈ set (tr_pairs F (D@E))" using "2.IH"[OF G'(1)] G'(1,2) by auto
      thus "G ∈ set (tr_pairs ((s,t)#F) (D@E))" using G'(3) by auto
    qed
  qed simp
qed
```

```
lemma trpairs_trms_subset:
  "G ∈ set (trpairs F D) ⟹ trmspairs G ⊆ pair ‘ set F ∪ pair ‘ set D"
proof (induction F D arbitrary: G rule: trpairs.induct)
  case (2 s t F D G)
  obtain d G’ where G:
      "d ∈ set D" "G’ ∈ set (trpairs F D)" "G = (pair (s,t), pair d)#G’"
    using "2.prems"(1) by moura

  show ?case using "2.IH"[OF G(1,2)] G(1,3) by auto
qed simp

lemma trpairs_trms_subset’:
  "⋃(trmspairs ‘ set (trpairs F D)) ⊆ pair ‘ set F ∪ pair ‘ set D"
using trpairs_trms_subset by blast

lemma tr_trms_subset:
  "A’ ∈ set (tr A D) ⟹ trmsst A’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set D"
proof (induction A D arbitrary: A’ rule: tr.induct)
  case 1 thus ?case by simp
next
  case (2 t A D)
  then obtain A’’ where A’’: "A’ = send⟨t⟩st#A’’" "A’’ ∈ set (tr A D)" by moura
  hence "trmsst A’’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set D" by (metis "2.IH")
  thus ?case using A’’ by (auto simp add: setopssst_def)
next
  case (3 t A D)
  then obtain A’’ where A’’: "A’ = receive⟨t⟩st#A’’" "A’’ ∈ set (tr A D)" by moura
  hence "trmsst A’’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set D" by (metis "3.IH")
  thus ?case using A’’ by (auto simp add: setopssst_def)
next
  case (4 ac t t’ A D)
  then obtain A’’ where A’’: "A’ = ⟨ac: t ≐ t’⟩st#A’’" "A’’ ∈ set (tr A D)" by moura
  hence "trmsst A’’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set D" by (metis "4.IH")
  thus ?case using A’’ by (auto simp add: setopssst_def)
next
  case (5 t s A D)
  hence "A’ ∈ set (tr A (List.insert (t,s) D))" by simp
  hence "trmsst A’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set (List.insert (t, s) D)"
    by (metis "5.IH")
  thus ?case by (auto simp add: setopssst_def)
next
  case (6 t s A D)
  from 6 obtain Di A’’ B C where A’’:
      "Di ∈ set (subseqs D)" "A’’ ∈ set (tr A [d←D. d ∉ set Di])" "A’ = (B@C)@A’’"
      "B = map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di"
      "C = map (λd. Inequality [] [(pair (t,s) , pair d)]) [d←D. d ∉ set Di]"
    by moura
  hence "trmsst A’’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set [d←D. d ∉ set Di]"
    by (metis "6.IH")
  hence "trmsst A’’ ⊆ trmssst (Delete t s#A) ∪ pair ‘ setopssst (Delete t s#A) ∪ pair ‘ set D"
    by (auto simp add: setopssst_def)
  moreover have "trmsst (B@C) ⊆ insert (pair (t,s)) (pair ‘ set D)"
    using A’’(4,5) subseqs_set_subset[OF A’’(1)] by auto
  moreover have "pair (t,s) ∈ pair ‘ setopssst (Delete t s#A)" by (simp add: setopssst_def)
  ultimately show ?case using A’’(3) trmsst_append[of "B@C" A’] by auto
next
  case (7 ac t s A D)
  from 7 obtain d A’’ where A’’:
      "d ∈ set D" "A’’ ∈ set (tr A D)"
      "A’ = ⟨ac: (pair (t,s)) ≐ (pair d)⟩st#A’’"
    by moura
  hence "trmsst A’’ ⊆ trmssst A ∪ pair ‘ setopssst A ∪ pair ‘ set D" by (metis "7.IH")
```

```
  moreover have "trms_st A' = {pair (t,s), pair d} ∪ trms_st A''"
    using A''(1,3) by auto
  ultimately show ?case using A''(1) by (auto simp add: setops_sst_def)
next
  case (8 X F F' A D)
  from 8 obtain A'' where A'':
      "A'' ∈ set (tr A D)" "A' = (map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' D))@A''"
    by moura

  define B where "B ≡ ⋃(trms_pairs ` set (tr_pairs F' D))"

  have "trms_st A'' ⊆ trms_sst A ∪ pair ` setops_sst A ∪ pair ` set D" by (metis A''(1) "8.IH")
  hence "trms_st A' ⊆ B ∪ trms_pairs F ∪ trms_sst A ∪ pair ` setops_sst A ∪ pair ` set D"
    using A'' B_def by auto
  moreover have "B ⊆ pair ` set F' ∪ pair ` set D"
    using tr_pairs_trms_subset'[of F' D] B_def by simp
  moreover have "pair ` setops_sst (∀X⟨∨≠: F ∨∉: F'⟩#A) = pair ` set F' ∪ pair ` setops_sst A"
    by (auto simp add: setops_sst_def)
  ultimately show ?case by auto
qed

lemma tr_pairs_vars_subset:
  "G ∈ set (tr_pairs F D) ⟹ fv_pairs G ⊆ fv_pairs F ∪ fv_pairs D"
proof (induction F D arbitrary: G rule: tr_pairs.induct)
  case (2 s t F D G)
  obtain d G' where G:
      "d ∈ set D" "G' ∈ set (tr_pairs F D)" "G = (pair (s,t), pair d)#G'"
    using "2.prems"(1) by moura

  show ?case using "2.IH"[OF G(1,2)] G(1,3) unfolding pair_def by auto
qed simp

lemma tr_pairs_vars_subset': "⋃(fv_pairs ` set (tr_pairs F D)) ⊆ fv_pairs F ∪ fv_pairs D"
using tr_pairs_vars_subset[of _ F D] by blast

lemma tr_vars_subset:
  assumes "A' ∈ set (tr A D)"
  shows "fv_st A' ⊆ fv_sst A ∪ (⋃(t,t') ∈ set D. fv t ∪ fv t')" (is ?P)
  and "bvars_st A' ⊆ bvars_sst A" (is ?Q)
proof -
  show ?P using assms
  proof (induction A arbitrary: A' D rule: strand_sem_stateful_induct)
    case (ConsIn A' D ac t s A)
    then obtain A'' d where *:
        "d ∈ set D" "A' = ⟨ac: (pair (t,s)) ≐ (pair d)⟩_st#A''"
        "A'' ∈ set (tr A D)"
      by moura
    hence "fv_st A'' ⊆ fv_sst A ∪ (⋃(t,t')∈set D. fv t ∪ fv t')" by (metis ConsIn.IH)
    thus ?case using * unfolding pair_def by auto
  next
    case (ConsDel A' D t s A)
    define Dfv where "Dfv ≡ λD::('fun,'var) dbstatelist. (⋃(t,t')∈set D. fv t ∪ fv t')"
    define fltD where "fltD ≡ λDi. filter (λd. d ∉ set Di) D"
    define constr where
      "constr ≡ λDi. (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩_st) Di)@
                     (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩_st) (fltD Di))"
    from ConsDel obtain A'' Di where *:
        "Di ∈ set (subseqs D)" "A' = (constr Di)@A''" "A'' ∈ set (tr A (fltD Di))"
      unfolding constr_def fltD_def by moura
    hence "fv_st A'' ⊆ fv_sst A ∪ Dfv (fltD Di)"
      unfolding Dfv_def constr_def fltD_def by (metis ConsDel.IH)
    moreover have "Dfv (fltD Di) ⊆ Dfv D" unfolding Dfv_def constr_def fltD_def by auto
    moreover have "Dfv Di ⊆ Dfv D"
```

```
      using subseqs_set_subset(1)[OF *(1)] unfolding Dfv_def constr_def fltD_def by fast
    moreover have "fv_st (constr Di) ⊆ fv t ∪ fv s ∪ (Dfv Di ∪ Dfv (fltD Di))"
      unfolding Dfv_def constr_def fltD_def pair_def by auto
    moreover have "fv_sst (Delete t s#A) = fv t ∪ fv s ∪ fv_sst A" by auto
    moreover have "fv_st A' = fv_st (constr Di) ∪ fv_st A''" using * by force
    ultimately have "fv_st A' ⊆ fv_sst (Delete t s#A) ∪ Dfv D" by auto
    thus ?case unfolding Dfv_def fltD_def constr_def by simp
  next
    case (ConsNegChecks A' D X F F' A)
    then obtain A'' where A'':
        "A'' ∈ set (tr A D)" "A' = (map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' D))@A''"
      by moura

    define B where "B ≡ ⋃(fv_pairs ` set (tr_pairs F' D))"

    have 1: "fv_st (map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' D)) ⊆ (B ∪ fv_pairs F) - set X"
      unfolding B_def by auto

    have 2: "B ⊆ fv_pairs F' ∪ fv_pairs D"
      using tr_pairs_vars_subset'[of F' D]
      unfolding B_def by simp

    have "fv_st A' ⊆ ((fv_pairs F' ∪ fv_pairs D ∪ fv_pairs F) - set X) ∪ fv_st A''"
      using 1 2 A''(2) by fastforce
    thus ?case using ConsNegChecks.IH[OF A''(1)] by auto
  qed fastforce+

  show ?Q using assms by (induct A arbitrary: A' D rule: strand_sem_stateful_induct) fastforce+
qed

lemma tr_vars_disj:
  assumes "A' ∈ set (tr A D)" "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvars_sst A = {}"
  and "fv_sst A ∩ bvars_sst A = {}"
  shows "fv_st A' ∩ bvars_st A' = {}"
  using assms tr_vars_subset by fast

lemma wf_fun_pair_ineqs_map:
  assumes "wf_st X A"
  shows "wf_st X (map (λd. ∀Y⟨∨≠: [(pair (t, s), pair d)]⟩_st) D@A)"
using assms by (induct D) auto

lemma wf_fun_pair_negchecks_map:
  assumes "wf_st X A"
  shows "wf_st X (map (λG. ∀Y⟨∨≠: (F@G)⟩_st) M@A)"
using assms by (induct M) auto

lemma wf_fun_pair_eqs_ineqs_map:
  fixes A::"('fun,'var) strand"
  assumes "wf_st X A" "Di ∈ set (subseqs D)" "∀(t,t') ∈ set D. fv t ∪ fv t' ⊆ X"
  shows "wf_st X ((map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩_st) Di)@
                (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩_st) [d←D. d ∉ set Di])@A)"
proof -
  let ?c1 = "map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩_st) Di"
  let ?c2 = "map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩_st) [d←D. d ∉ set Di]"
  have 1: "wf_st X (?c2@A)" using wf_fun_pair_ineqs_map[OF assms(1)] by simp
  have 2: "∀(t,t') ∈ set Di. fv t ∪ fv t' ⊆ X"
    using assms(2,3) by (meson contra_subsetD subseqs_set_subset(1))
  have "wf_st X (?c1@B)" when "wf_st X B" for B::"('fun,'var) strand"
    using 2 that by (induct Di) auto
  thus ?thesis using 1 by simp
qed

lemma trms_sst_wt_subst_ex:
```

```
  assumes ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
    and t: "t ∈ trms_sst (S ·sst ϑ)"
  shows "∃s δ. s ∈ trms_sst S ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"
using t
proof (induction S)
  case (Cons s S) thus ?case
  proof (cases "t ∈ trms_sst (S ·sst ϑ)")
    case False
    hence "t ∈ trms_sstp (s ·sstp ϑ)"
      using Cons.prems trms_sst_subst_cons[of s S ϑ]
      by auto
    then obtain u where u: "u ∈ trms_sstp s" "t = u · rm_vars (set (bvars_sstp s)) ϑ"
      using trms_sstp_subst'' by blast
    thus ?thesis
      using trms_sst_subst_cons[of s S ϑ]
            wt_subst_rm_vars[OF ϑ(1), of "set (bvars_sstp s)"]
            wf_trms_subst_rm_vars'[OF ϑ(2), of "set (bvars_sstp s)"]
      by fastforce
  qed auto
qed simp

lemma setops_sst_wt_subst_ex:
  assumes ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
    and t: "t ∈ pair ` setops_sst (S ·sst ϑ)"
  shows "∃s δ. s ∈ pair ` setops_sst S ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"
using t
proof (induction S)
  case (Cons x S) thus ?case
  proof (cases x)
    case (Insert t' s)
    hence "t = pair (t',s) · ϑ ∨ t ∈ pair ` setops_sst (S ·sst ϑ)"
      using Cons.prems subst_sst_cons[of _ S ϑ]
      unfolding pair_def by (force simp add: setops_sst_def)
    thus ?thesis
      using Insert Cons.IH ϑ by (cases "t = pair (t', s) · ϑ") (fastforce, auto)
  next
    case (Delete t' s)
    hence "t = pair (t',s) · ϑ ∨ t ∈ pair ` setops_sst (S ·sst ϑ)"
      using Cons.prems subst_sst_cons[of _ S ϑ]
      unfolding pair_def by (force simp add: setops_sst_def)
    thus ?thesis
      using Delete Cons.IH ϑ by (cases "t = pair (t', s) · ϑ") (fastforce, auto)
  next
    case (InSet ac t' s)
    hence "t = pair (t',s) · ϑ ∨ t ∈ pair ` setops_sst (S ·sst ϑ)"
      using Cons.prems subst_sst_cons[of _ S ϑ]
      unfolding pair_def by (force simp add: setops_sst_def)
    thus ?thesis
      using InSet Cons.IH ϑ by (cases "t = pair (t', s) · ϑ") (fastforce, auto)
  next
    case (NegChecks X F F')
    hence "t ∈ pair ` set (F' ·pairs rm_vars (set X) ϑ) ∨ t ∈ pair ` setops_sst (S ·sst ϑ)"
      using Cons.prems subst_sst_cons[of _ S ϑ]
      unfolding pair_def by (force simp add: setops_sst_def)
    thus ?thesis
    proof
      assume "t ∈ pair ` set (F' ·pairs rm_vars (set X) ϑ)"
      then obtain s where s: "t = s · rm_vars (set X) ϑ" "s ∈ pair ` set F'"
        using subst_apply_pairs_pair_image_subst[of F' "rm_vars (set X) ϑ"] by auto
      thus ?thesis
        using NegChecks setops_sst_pair_image_cons(8)[of X F F' S]
              wt_subst_rm_vars[OF ϑ(1), of "set X"]
              wf_trms_subst_rm_vars'[OF ϑ(2), of "set X"]
```

```
        by fast
    qed (use Cons.IH in auto)
  qed (auto simp add: setops_sst_def subst_sst_cons[of _ S ϑ])
qed (simp add: setops_sst_def)

lemma setops_sst_wf_trms:
  "wf_trms (trms_sst A) ⟹ wf_trms (pair ` setops_sst A)"
  "wf_trms (trms_sst A) ⟹ wf_trms (trms_sst A ∪ pair ` setops_sst A)"
proof -
  show "wf_trms (trms_sst A) ⟹ wf_trms (pair ` setops_sst A)"
  proof (induction A)
    case (Cons a A)
    hence 0: "wf_trms (trms_sstp a)" "wf_trms (pair ` setops_sst A)" by auto
    thus ?case
    proof (cases a)
      case (NegChecks X F F')
      hence "wf_trms (trms_pairs F')" using 0 by simp
      thus ?thesis using NegChecks wf_trms_pairs[of F'] 0 by (auto simp add: setops_sst_def)
    qed (auto simp add: setops_sst_def dest: fun_pair_wf_trm)
  qed (auto simp add: setops_sst_def)
  thus "wf_trms (trms_sst A) ⟹ wf_trms (trms_sst A ∪ pair ` setops_sst A)" by fast
qed

lemma SMP_MP_split:
  assumes "t ∈ SMP M"
    and M: "∀m ∈ M. is_Fun m"
  shows "(∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t ∈ M ·_set δ) ∨
         t ∈ SMP ((subterms_set M ∪ ⋃((set ∘ fst ∘ Ana) ` M)) - M)"
  (is "?P t ∨ ?Q t")
using assms(1)
proof (induction t rule: SMP.induct)
  case (MP t)
  have "wt_subst Var" "wf_trms (subst_range Var)" "M ·_set Var = M" by simp_all
  thus ?case using MP by metis
next
  case (Subterm t t')
  show ?case using Subterm.IH
  proof
    assume "?P t"
    then obtain s δ where s: "s ∈ M" "t = s · δ" and δ: "wt_subst δ" "wf_trms (subst_range δ)" by
moura
    then obtain f T where fT: "s = Fun f T" using M by fast

    have "(∃s'. s' ⊑ s ∧ t' = s' · δ) ∨ (∃x ∈ fv s. t' ⊏ δ x)"
      using subterm_subst_unfold[OF Subterm.hyps(2)[unfolded s(2)]] by blast
    thus ?thesis
    proof
      assume "∃s'. s' ⊑ s ∧ t' = s' · δ"
      then obtain s' where s': "s' ⊑ s" "t' = s' · δ" by moura
      show ?thesis
      proof (cases "s' ∈ M")
        case True thus ?thesis using s' δ by blast
      next
        case False
        hence "s' ∈ (subterms_set M ∪ ⋃((set ∘ fst ∘ Ana) ` M)) - M" using s'(1) s(1) by force
        thus ?thesis using SMP.Substitution[OF SMP.MP[of s'] δ] s' by presburger
      qed
    next
      assume "∃x ∈ fv s. t' ⊏ δ x"
      then obtain x where x: "x ∈ fv s" "t' ⊏ δ x" by moura
      have "Var x ∉ M" using M by blast
      hence "Var x ∈ (subterms_set M ∪ ⋃((set ∘ fst ∘ Ana) ` M)) - M"
        using s(1) var_is_subterm[OF x(1)] by blast
```

```
      hence "δ x ∈ SMP ((subterms_set M ∪ ⋃((set ∘ fst ∘ Ana) ` M)) - M)"
        using SMP.Substitution[OF SMP.MP[of "Var x"] δ] by auto
      thus ?thesis using SMP.Subterm x(2) by presburger
    qed
  qed (metis SMP.Subterm[OF _ Subterm.hyps(2)])
next
  case (Substitution t δ)
  show ?case using Substitution.IH
  proof
    assume "?P t"
    then obtain ϑ where "wt_subst ϑ" "wf_trms (subst_range ϑ)" "t ∈ M ·_set ϑ" by moura
    hence "wt_subst (ϑ ∘_s δ)" "wf_trms (subst_range (ϑ ∘_s δ))" "t · δ ∈ M ·_set (ϑ ∘_s δ)"
      using wt_subst_compose[of ϑ, OF _ Substitution.hyps(2)]
            wf_trm_subst_compose[of ϑ _ δ, OF _ wf_trm_subst_rangeD[OF Substitution.hyps(3)]]
            wf_trm_subst_range_iff
      by (argo, blast, auto)
    thus ?thesis by blast
  next
    assume "?Q t" thus ?thesis using SMP.Substitution[OF _ Substitution.hyps(2,3)] by meson
  qed
next
  case (Ana t K T k)
  show ?case using Ana.IH
  proof
    assume "?P t"
    then obtain ϑ where ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" "t ∈ M ·_set ϑ" by moura
    then obtain s where s: "s ∈ M" "t = s · ϑ" by auto
    then obtain f S where fT: "s = Fun f S" using M by (cases s) auto
    obtain K' T' where s_Ana: "Ana s = (K', T')" by (metis surj_pair)
    hence "set K = set K' ·_set ϑ" "set T = set T' ·_set ϑ"
      using Ana_subst'[of f S K' T'] fT Ana.hyps(2) s(2) by auto
    then obtain k' where k': "k' ∈ set K'" "k = k' · ϑ" using Ana.hyps(3) by fast
    show ?thesis
    proof (cases "k' ∈ M")
      case True thus ?thesis using k' ϑ(1,2) by blast
    next
      case False
      hence "k' ∈ (subterms_set M ∪ ⋃((set ∘ fst ∘ Ana) ` M)) - M" using k'(1) s_Ana s(1) by force
      thus ?thesis using SMP.Substitution[OF SMP.MP[of k'] ϑ(1,2)] k'(2) by presburger
    qed
  next
    assume "?Q t" thus ?thesis using SMP.Ana[OF _ Ana.hyps(2,3)] by meson
  qed
qed

lemma setops_subterm_trms:
  assumes t: "t ∈ pair ` setops_sst S"
    and s: "s ⊏ t"
  shows "s ∈ subterms_set (trms_sst S)"
proof -
  obtain u u' where u: "pair (u,u') ∈ pair ` setops_sst S" "t = pair (u,u')"
    using t setops_sst_are_pairs[of _ S] by blast
  hence "s ⊑ u ∨ s ⊑ u'" using s unfolding pair_def by auto
  thus ?thesis using u setops_sst_member_iff[of u u' S] unfolding trms_sst_def by force
qed

lemma setops_subterms_cases:
  assumes t: "t ∈ subterms_set (pair ` setops_sst S)"
  shows "t ∈ subterms_set (trms_sst S) ∨ t ∈ pair ` setops_sst S"
proof -
  obtain s s' where s: "pair (s,s') ∈ pair ` setops_sst S" "t ⊑ pair (s,s')"
    using t setops_sst_are_pairs[of _ S] by blast
  hence "t ∈ pair ` setops_sst S ∨ t ⊑ s ∨ t ⊑ s'" unfolding pair_def by auto
```

```
  thus ?thesis using s setops_sst_member_iff[of s s' S] unfolding trms_sst_def by force
qed

lemma setops_SMP_cases:
  assumes "t ∈ SMP (pair ` setops_sst S)"
    and "∀p. Ana (pair p) = ([], [])"
  shows "(∃δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ t ∈ pair ` setops_sst S ·_set δ) ∨ t ∈ SMP (trms_sst
S)"
proof -
  have 0: "⋃((set ∘ fst ∘ Ana) ` pair ` setops_sst S) = {}"
  proof (induction S)
    case (Cons x S) thus ?case
      using assms(2) by (cases x) (auto simp add: setops_sst_def)
  qed (simp add: setops_sst_def)

  have 1: "∀m ∈ pair ` setops_sst S. is_Fun m"
  proof (induction S)
    case (Cons x S) thus ?case
      unfolding pair_def by (cases x) (auto simp add: assms(2) setops_sst_def)
  qed (simp add: setops_sst_def)

  have 2:
      "subterms_set (pair ` setops_sst S) ∪
       ⋃((set ∘ fst ∘ Ana) ` (pair ` setops_sst S)) - pair ` setops_sst S
        ⊆ subterms_set (trms_sst S)"
    using 0 setops_subterms_cases by fast

  show ?thesis
    using SMP_MP_split[OF assms(1) 1] SMP_mono[OF 2] SMP_subterms_eq[of "trms_sst S"]
    by blast
qed

lemma tfr_setops_if_tfr_trms:
  assumes "Pair ∉ ⋃(funs_term ` SMP (trms_sst S))"
    and "∀p. Ana (pair p) = ([], [])"
    and "∀s ∈ pair ` setops_sst S. ∀t ∈ pair ` setops_sst S. (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
    and "∀s ∈ pair ` setops_sst S. ∀t ∈ pair ` setops_sst S.
          (∃σ ϑ ϱ. wt_subst σ ∧ wt_subst ϑ ∧ wf_trms (subst_range σ) ∧ wf_trms (subst_range ϑ) ∧
                   Unifier ϱ (s · σ) (t · ϑ))
          ⟶ (∃δ. Unifier δ s t)"
    and tfr: "tfr_set (trms_sst S)"
  shows "tfr_set (trms_sst S ∪ pair ` setops_sst S)"
proof -
  have 0: "t ∈ SMP (trms_sst S) - range Var ∨ t ∈ SMP (pair ` setops_sst S) - range Var"
    when "t ∈ SMP (trms_sst S ∪ pair ` setops_sst S) - range Var" for t
    using that SMP_union by blast

  have 1: "s ∈ SMP (trms_sst S) - range Var"
      when st: "s ∈ SMP (pair ` setops_sst S) - range Var"
               "t ∈ SMP (trms_sst S) - range Var"
               "∃δ. Unifier δ s t"
        for s t
  proof -
    have "(∃δ. s ∈ pair ` setops_sst S ·_set δ) ∨ s ∈ SMP (trms_sst S) - range Var"
      using st setops_SMP_cases[of s S] assms(2) by blast
    moreover {
      fix δ assume δ: "s ∈ pair ` setops_sst S ·_set δ"
      then obtain s' where s': "s' ∈ pair ` setops_sst S" "s = s' · δ" by blast
      then obtain u u' where u: "s' = Fun Pair [u,u']"
        using setops_sst_are_pairs[of s'] unfolding pair_def by fast
      hence *: "s = Fun Pair [u · δ, u' · δ]" using δ s' by simp

      obtain f T where fT: "t = Fun f T" using st(2) by (cases t) auto
```

```
      hence "f ≠ Pair" using st(2) assms(1) by auto
      hence False using st(3) * fT s' u by fast
    } ultimately show ?thesis by meson
  qed

  have 2: "Γ s = Γ t"
      when "s ∈ SMP (trms_sst S) - range Var"
           "t ∈ SMP (trms_sst S) - range Var"
           "∃δ. Unifier δ s t"
       for s t
    using that tfr unfolding tfr_set_def by blast

  have 3: "Γ s = Γ t"
      when st: "s ∈ SMP (pair ` setops_sst S) - range Var"
               "t ∈ SMP (pair ` setops_sst S) - range Var"
               "∃δ. Unifier δ s t"
      for s t
  proof -
    let ?P = "λs δ. wt_subst δ ∧ wf_trms (subst_range δ) ∧ s ∈ pair ` setops_sst S ·_set δ"
    have "(∃δ. ?P s δ) ∨ s ∈ SMP (trms_sst S) - range Var"
         "(∃δ. ?P t δ) ∨ t ∈ SMP (trms_sst S) - range Var"
      using setops_SMP_cases[of _ S] assms(2) st(1,2) by auto
    hence "(∃δ δ'. ?P s δ ∧ ?P t δ') ∨ Γ s = Γ t" by (metis 1 2 st)
    moreover {
      fix δ δ' assume *: "?P s δ" "?P t δ'"
      then obtain s' t' where **:
          "s' ∈ pair ` setops_sst S" "t' ∈ pair ` setops_sst S" "s = s' · δ" "t = t' · δ'"
        by blast
      hence "∃ϑ. Unifier ϑ s' t'" using st(3) assms(4) * by blast
      hence "Γ s' = Γ t'" using assms(3) ** by blast
      hence "Γ s = Γ t" using * **(3,4) wt_subst_trm''[of δ s'] wt_subst_trm''[of δ' t'] by argo
    } ultimately show ?thesis by blast
  qed

  show ?thesis using 0 1 2 3 unfolding tfr_set_def by metis
qed
```

4.2.2 The Typing Result for Stateful Constraints

```
context
begin
private lemma tr_wf':
  assumes "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvars_sst A = {}"
  and "∀(t,t') ∈ set D. fv t ∪ fv t' ⊆ X"
  and "wf'_sst X A" "fv_sst A ∩ bvars_sst A = {}"
  and "A' ∈ set (tr A D)"
  shows "wf_st X A'"
proof -
  define P where
    "P = (λ(D::('fun,'var) dbstatelist) (A::('fun,'var) stateful_strand).
          (∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvars_sst A = {}) ∧ fv_sst A ∩ bvars_sst A = {})"

  have "P D A" using assms(1,4) by (simp add: P_def)
  with assms(5,3,2) show ?thesis
  proof (induction A arbitrary: A' D X rule: wf'_sst.induct)
    case 1 thus ?case by simp
  next
    case (2 X t A A')
    then obtain A'' where A'': "A' = receive⟨t⟩_st#A''" "A'' ∈ set (tr A D)" "fv t ⊆ X"
      by moura
    have *: "wf'_sst X A" "∀(s,s') ∈ set D. fv s ∪ fv s' ⊆ X" "P D A"
      using 2(1,2,3,4) apply (force, force)
      using 2(5) unfolding P_def by force
```

```
  show ?case using "2.IH"[OF A''(2) *] A''(1,3) by simp
next
  case (3 X t A A')
  then obtain A'' where A'': "A' = send⟨t⟩st#A''" "A'' ∈ set (tr A D)"
    by moura
  have *: "wf'sst (X ∪ fv t) A" "∀(s,s') ∈ set D. fv s ∪ fv s' ⊆ X ∪ fv t" "P D A"
    using 3(1,2,3,4) apply (force, force)
    using 3(5) unfolding P_def by force
  show ?case using "3.IH"[OF A''(2) *] A''(1) by simp
next
  case (4 X t t' A A')
  then obtain A'' where A'': "A' = ⟨assign: t ≐ t'⟩st#A''" "A'' ∈ set (tr A D)" "fv t' ⊆ X"
    by moura
  have *: "wf'sst (X ∪ fv t) A" "∀(s,s') ∈ set D. fv s ∪ fv s' ⊆ X ∪ fv t" "P D A"
    using 4(1,2,3,4) apply (force, force)
    using 4(5) unfolding P_def by force
  show ?case using "4.IH"[OF A''(2) *] A''(1,3) by simp
next
  case (5 X t t' A A')
  then obtain A'' where A'': "A' = ⟨check: t ≐ t'⟩st#A''" "A'' ∈ set (tr A D)"
    by moura
  have *: "wf'sst X A" "P D A"
    using 5(3) apply force
    using 5(5) unfolding P_def by force
  show ?case using "5.IH"[OF A''(2) *(1) 5(4) *(2)] A''(1) by simp
next
  case (6 X t s A A')
  hence A': "A' ∈ set (tr A (List.insert (t,s) D))" "fv t ⊆ X" "fv s ⊆ X" by auto
  have *: "wf'sst X A" "∀(s,s') ∈ set (List.insert (t,s) D). fv s ∪ fv s' ⊆ X" using 6 by auto
  have **: "P (List.insert (t,s) D) A" using 6(5) unfolding P_def by force
  show ?case using "6.IH"[OF A'(1) * **] A'(2,3) by simp
next
  case (7 X t s A A')
  let ?constr = "λDi. (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@
                      (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di])"
  from 7 obtain Di A'' where A'':
      "A' = ?constr Di@A''" "A'' ∈ set (tr A [d←D. d ∉ set Di])"
      "Di ∈ set (subseqs D)"
    by moura
  have *: "wf'sst X A" "∀(t',s') ∈ set [d←D. d ∉ set Di]. fv t' ∪ fv s' ⊆ X"
    using 7 by auto
  have **: "P [d←D. d ∉ set Di] A" using 7 unfolding P_def by force
  have ***: "∀(t, t') ∈ set D. fv t ∪ fv t' ⊆ X" using 7 by auto
  show ?case
    using "7.IH"[OF A''(2) * **] A''(1) wf_fun_pair_eqs_ineqs_map[OF _ A''(3) ***]
    by simp
next
  case (8 X t s A A')
  then obtain d A'' where A'':
      "A' = ⟨assign: (pair (t,s)) ≐ (pair d)⟩st#A''"
      "A'' ∈ set (tr A D)" "d ∈ set D"
    by moura
  have *: "wf'sst (X ∪ fv t ∪ fv s) A" "∀(t',s')∈set D. fv t' ∪ fv s' ⊆ X ∪ fv t ∪ fv s" "P D A"
    using 8(1,2,3,4) apply (force, force)
    using 8(5) unfolding P_def by force
  have **: "fv (pair d) ⊆ X" using A''(3) "8.prems"(3) unfolding pair_def by fastforce
  have ***: "fv (pair (t,s)) = fv s ∪ fv t" unfolding pair_def by auto
  show ?case using "8.IH"[OF A''(2) *] A''(1) ** *** unfolding pair_def by (simp add: Un_assoc)
next
  case (9 X t s A A')
  then obtain d A'' where A'':
      "A' = ⟨check: (pair (t,s)) ≐ (pair d)⟩st#A''"
      "A'' ∈ set (tr A D)" "d ∈ set D"
```

```
      by moura
    have *: "wf'sst X A""P D A"
      using 9(3) apply force
      using 9(5) unfolding P_def by force
    have **: "fv (pair d) ⊆ X" using A''(3) "9.prems"(3) unfolding pair_def by fastforce
    have ***: "fv (pair (t,s)) = fv s ∪ fv t" unfolding pair_def by auto
    show ?case using "9.IH"[OF A''(2) *(1) 9(4) *(2)] A''(1) ** *** by (simp add: Un_assoc)
  next
    case (10 X Y F F' A A')
    from 10 obtain A'' where A'':
        "A' = (map (λG. ∀Y⟨∨≠: (F@G)⟩st) (trpairs F' D))@A''" "A'' ∈ set (tr A D)"
      by moura

    have *: "wf'sst X A" "∀(t',s') ∈ set D. fv t' ∪ fv s' ⊆ X" using 10 by auto

    have "bvarssst A ⊆ bvarssst (∀Y⟨∨≠: F ∨∉: F'⟩#A)" "fvsst A ⊆ fvsst (∀Y⟨∨≠: F ∨∉: F'⟩#A)" by
auto
    hence **:  "P D A" using 10 unfolding P_def by blast

    show ?case using "10.IH"[OF A''(2) * **] A''(1) wf_fun_pair_negchecks_map by simp
  qed
qed

private lemma tr_wftrms:
  assumes "A' ∈ set (tr A [])" "wftrms (trmssst A)"
  shows "wftrms (trmsst A')"
using tr_trms_subset[OF assms(1)] setopssst_wftrms(2)[OF assms(2)]
by auto

lemma tr_wf:
  assumes "A' ∈ set (tr A [])"
    and "wfsst A"
    and "wftrms (trmssst A)"
  shows "wfst {} A'"
    and "wftrms (trmsst A')"
    and "fvst A' ∩ bvarsst A' = {}"
using tr_wf'[OF _ _ _ _ assms(1)]
      tr_wftrms[OF assms(1,3)]
      tr_vars_disj[OF assms(1)]
      assms(2)
by fastforce+

private lemma tr_tfrsstp:
  assumes "A' ∈ set (tr A D)" "list_all tfrsstp A"
  and "fvsst A ∩ bvarssst A = {}" (is "?P0 A D")
  and "∀(t,s) ∈ set D. (fv t ∪ fv s) ∩ bvarssst A = {}" (is "?P1 A D")
  and "∀t ∈ pair ` setopssst A ∪ pair ` set D. ∀t' ∈ pair ` setopssst A ∪ pair ` set D.
        (∃δ. Unifier δ t t') ⟶ Γ t = Γ t'" (is "?P3 A D")
  shows "list_all tfrstp A'"
proof -
  have sublmm: "list_all tfrsstp A" "?P0 A D" "?P1 A D" "?P3 A D"
    when p: "list_all tfrsstp (a#A)" "?P0 (a#A) D" "?P1 (a#A) D" "?P3 (a#A) D"
    for a A D
    using p(1) apply (simp add: tfrsst_def)
    using p(2) fvsst_cons_subset bvarssst_cons_subset apply fast
    using p(3) bvarssst_cons_subset apply fast
    using p(4) setopssst_cons_subset by fast

  show ?thesis using assms
  proof (induction A D arbitrary: A' rule: tr.induct)
    case 1 thus ?case by simp
  next
    case (2 t A D)
```

```
  note prems = "2.prems"
  note IH = "2.IH"
  from prems(1) obtain A'' where A'': "A' = send⟨t⟩st#A''" "A'' ∈ set (tr A D)"
    by moura
  have "list_all tfrstp A''" using IH[OF A''(2)] prems(5) sublmm[OF prems(2,3,4,5)] by meson
  thus ?case using A''(1) by simp
next
  case (3 t A D)
  note prems = "3.prems"
  note IH = "3.IH"
  from prems(1) obtain A'' where A'': "A' = receive⟨t⟩st#A''" "A'' ∈ set (tr A D)"
    by moura
  have "list_all tfrstp A''" using IH[OF A''(2)] prems(5) sublmm[OF prems(2,3,4,5)] by meson
  thus ?case using A''(1) by simp
next
  case (4 ac t t' A D)
  note prems = "4.prems"
  note IH = "4.IH"
  from prems(1) obtain A'' where A'':
      "A' = ⟨ac: t ≐ t'⟩st#A''" "A'' ∈ set (tr A D)"
    by moura
  have "list_all tfrstp A''" using IH[OF A''(2)] prems(5) sublmm[OF prems(2,3,4,5)] by meson
  moreover have "(∃δ. Unifier δ t t') ⟹ Γ t = Γ t'" using prems(2) by (simp add: tfrsst_def)
  ultimately show ?case using A''(1) by auto
next
  case (5 t s A D)
  note prems = "5.prems"
  note IH = "5.IH"
  from prems(1) have A': "A' ∈ set (tr A (List.insert (t,s) D))" by simp

  have 1: "list_all tfrsstp A" using sublmm[OF prems(2,3,4,5)] by simp

  have "pair ` setopssst (Insert t s#A) ∪ pair`set D =
        pair ` setopssst A ∪ pair`set (List.insert (t,s) D)"
    by (simp add: setopssst_def)
  hence 3: "?P3 A (List.insert (t,s) D)" using prems(5) by metis
  moreover have "?P1 A (List.insert (t, s) D)" using prems(3,4) bvarssst_cons_subset[of A] by auto
  ultimately have "list_all tfrstp A'" using IH[OF A' sublmm(1,2)[OF prems(2,3,4,5)] _ 3] by metis
  thus ?case using A'(1) by auto
next
  case (6 t s A D)
  note prems = "6.prems"
  note IH = "6.IH"

  define constr where constr:
    "constr ≡ (λDi. (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@
                     (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di]))"

  from prems(1) obtain Di A'' where A'':
      "A' = constr Di@A''" "A'' ∈ set (tr A [d←D. d ∉ set Di])"
      "Di ∈ set (subseqs D)"
    unfolding constr by auto

  define Q1 where "Q1 ≡ (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
      ∀x ∈ (fvpairs F) - set X. ∃a. Γ (Var x) = TAtom a)"

  define Q2 where "Q2 ≡ (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
      ∀f T. Fun f T ∈ subtermsset (trmspairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))"

  have "set [d←D. d ∉ set Di] ⊆ set D"
       "pair ` setopssst A ∪ pair ` set [d←D. d ∉ set Di]
       ⊆ pair ` setopssst (Delete t s#A) ∪ pair ` set D"
    by (auto simp add: setopssst_def)
```

```
  hence *: "?P3 A [d←D. d ∉ set Di]" using prems(5) by blast
  have **: "?P1 A [d←D. d ∉ set Di]" using prems(4,5) by auto
  have 1: "list_all tfr_stp A''"
    using IH[OF A''(3,2) sublmm(1,2)[OF prems(2,3,4,5)] ** *]
    by metis

  have 2: "⟨ac: u ≐ u'⟩_st ∈ set A'' ∨
          (∃d ∈ set Di. u = pair (t,s) ∧ u' = pair d)"
    when "⟨ac: u ≐ u'⟩_st ∈ set A'" for ac u u'
    using that A''(1) unfolding constr by force
  have 3: "Inequality X U ∈ set A' ⟹ Inequality X U ∈ set A'' ∨
          (∃d ∈ set [d←D. d ∉ set Di].
             U = [(pair (t,s), pair d)] ∧ Q2 [(pair (t,s), pair d)] X)"
      for X U
    using A''(1) unfolding Q2_def constr by force
  have 4:
      "∀d∈set D. (∃δ. Unifier δ (pair (t,s)) (pair d)) ⟶ Γ (pair (t,s)) = Γ (pair d)"
    using prems(5) by (simp add: setops_sst_def)

  { fix ac u u'
    assume a: "⟨ac: u ≐ u'⟩_st ∈ set A'" "∃δ. Unifier δ u u'"
    hence "⟨ac: u ≐ u'⟩_st ∈ set A'' ∨ (∃d ∈ set Di. u = pair (t,s) ∧ u' = pair d)"
      using 2 by metis
    hence "Γ u = Γ u'"
      using 1(1) 4 subseqs_set_subset[OF A''(3)] a(2) tfr_stp_list_all_alt_def[of A'']
      by blast
  } moreover {
    fix u U
    assume "∀U⟨∨≠: u⟩_st ∈ set A'"
    hence "∀U⟨∨≠: u⟩_st ∈ set A'' ∨
          (∃d ∈ set [d←D. d ∉ set Di]. u = [(pair (t,s), pair d)] ∧ Q2 u U)"
      using 3 by metis
    hence "Q1 u U ∨ Q2 u U"
      using 1 4 subseqs_set_subset[OF A''(3)] tfr_stp_list_all_alt_def[of A'']
      unfolding Q1_def Q2_def
      by blast
  } ultimately show ?case using tfr_stp_list_all_alt_def[of A'] unfolding Q1_def Q2_def by blast
next
  case (7 ac t s A D)
  note prems = "7.prems"
  note IH = "7.IH"

  from prems(1) obtain d A'' where A'':
      "A' = ⟨ac: (pair (t,s)) ≐ (pair d)⟩_st#A''"
      "A'' ∈ set (tr A D)" "d ∈ set D"
    by moura

  have "list_all tfr_stp A''"
    using IH[OF A''(2) sublmm(1,2,3)[OF prems(2,3,4,5)] sublmm(4)[OF prems(2,3,4,5)]]
    by metis
  moreover have "(∃δ. Unifier δ (pair (t,s)) (pair d)) ⟹ Γ (pair (t,s)) = Γ (pair d)"
    using prems(2,5) A''(3) unfolding tfr_sst_def by (simp add: setops_sst_def)
  ultimately show ?case using A''(1) by fastforce
next
  case (8 X F F' A D)
  note prems = "8.prems"
  note IH = "8.IH"

  define constr where "constr = (map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' D))"

  define Q1 where "Q1 ≡ (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
      ∀x ∈ (fv_pairs F) - set X. ∃a. Γ (Var x) = TAtom a)"
```

```
    define Q2 where "Q2 ≡ (λ(M::('fun,'var) terms) X.
        ∀f T. Fun f T ∈ subterms_set M ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))"

    have Q2_subset: "Q2 M' X" when "M' ⊆ M" "Q2 M X" for X M M'
      using that unfolding Q2_def by auto

    have Q2_supset: "Q2 (M ∪ M') X" when "Q2 M X" "Q2 M' X" for X M M'
      using that unfolding Q2_def by auto

    from prems(1) obtain A'' where A'': "A' = constr@A''" "A'' ∈ set (tr A D)"
      using constr_def by moura

    have 0: "F' = [] ⟹ constr = [∀X⟨∨≠: F⟩_st]" unfolding constr_def by simp

    have 1: "list_all tfr_stp A''"
      using IH[OF A''(2) sublmm(1,2,3)[OF prems(2,3,4,5)] sublmm(4)[OF prems(2,3,4,5)]]
      by metis

    have 2: "(F' = [] ∧ Q1 F X) ∨ Q2 (trms_pairs F ∪ pair ` set F') X"
      using prems(2) unfolding Q1_def Q2_def by simp

    have 3: "list_all tfr_stp constr" when "F' = []" "Q1 F X"
      using that 0 2 tfr_stp_list_all_alt_def[of constr] unfolding Q1_def by auto

    { fix c assume "c ∈ set constr"
      hence "∃G ∈ set (tr_pairs F' D). c = ∀X⟨∨≠: (F@G)⟩_st" unfolding constr_def by force
    } moreover {
      fix G
      assume G: "G ∈ set (tr_pairs F' D)"
         and c: "∀X⟨∨≠: (F@G)⟩_st ∈ set constr"
         and e: "Q2 (trms_pairs F ∪ pair ` set F') X"

      have d_Q2: "Q2 (pair ` set D) X" unfolding Q2_def
      proof (intro allI impI)
        fix f T assume "Fun f T ∈ subterms_set (pair ` set D)"
        then obtain d where d: "d ∈ set D" "Fun f T ∈ subterms (pair d)" by auto
        hence "fv (pair d) ∩ set X = {}" using prems(4) unfolding pair_def by force
        thus "T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)"
          by (metis fv_disj_Fun_subterm_param_cases d(2))
      qed

      have "trms_pairs (F@G) ⊆ trms_pairs F ∪ pair ` set F' ∪ pair ` set D"
        using tr_pairs_trms_subset[OF G] by auto
      hence "Q2 (trms_pairs (F@G)) X" using Q2_subset[OF _ Q2_supset[OF e d_Q2]] by metis
      hence "tfr_stp (∀X⟨∨≠: (F@G)⟩_st)" by (metis Q2_def tfr_stp.simps(2))
    } ultimately have 4: "list_all tfr_stp constr" when "Q2 (trms_pairs F ∪ pair ` set F') X"
      using that Ball_set by blast

    have 5: "list_all tfr_stp constr" using 2 3 4 by metis

    show ?case using 1 5 A''(1) by simp
  qed
qed

lemma tr_tfr:
  assumes "A' ∈ set (tr A [])" and "tfr_sst A" and "fv_sst A ∩ bvars_sst A = {}"
  shows "tfr_st A'"
proof -
  have *: "trms_st A' ⊆ trms_sst A ∪ pair ` setops_sst A" using tr_trms_subset[OF assms(1)] by simp
  hence "SMP (trms_st A') ⊆ SMP (trms_sst A ∪ pair ` setops_sst A)" using SMP_mono by simp
  moreover have "tfr_set (trms_sst A ∪ pair ` setops_sst A)" using assms(2) unfolding tfr_sst_def by
fast
  ultimately have 1: "tfr_set (trms_st A')" by (metis tfr_subset(2)[OF _ *])
```

```
  have **: "list_all tfr_sstp A" using assms(2) unfolding tfr_sst_def by fast
  have "pair ‘ setops_sst A ⊆ SMP (trms_sst A ∪ pair ‘ setops_sst A) - Var‘𝒱"
    using setops_sst_are_pairs unfolding pair_def by auto
  hence ***: "∀t ∈ pair‘setops_sst A. ∀t’ ∈ pair‘setops_sst A. (∃δ. Unifier δ t t’) ⟶ Γ t = Γ t’"
    using assms(2) unfolding tfr_sst_def tfr_set_def by blast
  have 2: "list_all tfr_stp A’"
    using tr_tfr_sstp[OF assms(1) ** assms(3)] *** unfolding pair_def by fastforce

  show ?thesis by (metis 1 2 tfr_st_def)
qed

private lemma fun_pair_ineqs:
  assumes "d ·p δ ·p ϑ ≠ d’ ·p ℐ"
  shows "pair d · δ · ϑ ≠ pair d’ · ℐ"
proof -
  have "d ·p (δ ∘s ϑ) ≠ d’ ·p ℐ" using assms subst_pair_compose by metis
  hence "pair d · (δ ∘s ϑ) ≠ pair d’ · ℐ" using fun_pair_eq_subst by metis
  thus ?thesis by simp
qed

private lemma tr_Delete_constr_iff_aux1:
  assumes "∀d ∈ set Di. (t,s) ·p ℐ = d ·p ℐ"
  and "∀d ∈ set D - set Di. (t,s) ·p ℐ ≠ d ·p ℐ"
  shows "⟦M; (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@
           (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di])⟧d ℐ"
proof -
  from assms(2) have
    "⟦M; map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di]⟧d ℐ"
  proof (induction D)
    case (Cons d D)
    hence IH: "⟦M; map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D . d ∉ set Di]⟧d ℐ" by auto
    thus ?case
    proof (cases "d ∈ set Di")
      case False
      hence "(t,s) ·p ℐ ≠ d ·p ℐ" using Cons by simp
      hence "pair (t,s) · ℐ ≠ pair d · ℐ" using fun_pair_eq_subst by metis
      moreover have "⋀t (δ::(’fun,’var) subst). subst_domain δ = {} ⟹ t · δ = t" by auto
      ultimately have "∀δ. subst_domain δ = {} ⟶ pair (t,s) · δ · ℐ ≠ pair d · δ · ℐ" by metis
      thus ?thesis using IH by (simp add: ineq_model_def)
    qed simp
  qed simp
  moreover {
    fix B assume "⟦M; B⟧d ℐ"
    with assms(1) have "⟦M; (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@B⟧d ℐ"
      unfolding pair_def by (induction Di) auto
  } ultimately show ?thesis by metis
qed

private lemma tr_Delete_constr_iff_aux2:
  assumes "ground M"
  and "⟦M; (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@
         (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di])⟧d ℐ"
  shows "(∀d ∈ set Di. (t,s) ·p ℐ = d ·p ℐ) ∧ (∀d ∈ set D - set Di. (t,s) ·p ℐ ≠ d ·p ℐ)"
proof -
  let ?c1 = "map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di"
  let ?c2 = "map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di]"

  have "M ·set ℐ = M" using assms(1) subst_all_ground_ident by metis
  moreover have "ik_st ?c1 = {}" by auto
  ultimately have *:
      "⟦M; map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di⟧d ℐ"
      "⟦M; map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di]⟧d ℐ"
```

```
    using strand_sem_split(3,4)[of M ?c1 ?c2 I] assms(2) by auto

  from *(1) have 1: "∀d ∈ set Di. (t,s) ·p I = d ·p I" unfolding pair_def by (induct Di) auto
  from *(2) have 2: "∀d ∈ set D - set Di. (t,s) ·p I ≠ d ·p I"
  proof (induction D arbitrary: Di)
    case (Cons d D) thus ?case
    proof (cases "d ∈ set Di")
      case False
      hence IH: "∀d ∈ set D - set Di. (t,s) ·p I ≠ d ·p I" using Cons by force
      have "⋀t (δ::('fun,'var) subst). subst_domain δ = {} ∧ ground (subst_range δ) ⟷ δ = Var"
        by auto
      moreover have "ineq_model I [] [((pair (t,s)), (pair d))]"
        using False Cons.prems by simp
      ultimately have "pair (t,s) · I ≠ pair d · I" by (simp add: ineq_model_def)
      thus ?thesis using IH unfolding pair_def by force
    qed simp
  qed simp

  show ?thesis by (metis 1 2)
qed

private lemma tr_Delete_constr_iff:
  fixes I::"('fun,'var) subst"
  assumes "ground M"
  shows "set Di ·pset I ⊆ {(t,s) ·p I} ∧ (t,s) ·p I ∉ (set D - set Di) ·pset I ⟷
         ⟦M; (map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@
              (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di])⟧d I"
proof -
  let ?constr = "(map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di)@
                 (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di])"
  { assume "set Di ·pset I ⊆ {(t,s) ·p I}" "(t,s) ·p I ∉ (set D - set Di) ·pset I"
    hence "∀d ∈ set Di. (t,s) ·p I = d ·p I" "∀d ∈ set D - set Di. (t,s) ·p I ≠ d ·p I"
      by auto
    hence "⟦M; ?constr⟧d I" using tr_Delete_constr_iff_aux1 by simp
  } moreover {
    assume "⟦M; ?constr⟧d I"
    hence "∀d ∈ set Di. (t,s) ·p I = d ·p I" "∀d ∈ set D - set Di. (t,s) ·p I ≠ d ·p I"
      using assms tr_Delete_constr_iff_aux2 by auto
    hence "set Di ·pset I ⊆ {(t,s) ·p I} ∧ (t,s) ·p I ∉ (set D - set Di) ·pset I" by force
  } ultimately show ?thesis by metis
qed

private lemma tr_NotInSet_constr_iff:
  fixes I::"('fun,'var) subst"
  assumes "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ set X = {}"
  shows "(∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ (t,s) ·p δ ·p I ∉ set D ·pset I)
         ⟷ ⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st) D⟧d I"
proof -
  { assume "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ (t,s) ·p δ ·p I ∉ set D ·pset I"
    with assms have "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st) D⟧d I"
    proof (induction D)
      case (Cons d D)
      obtain t' s' where d: "d = (t',s')" by moura
      have "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st) D⟧d I"
           "map (λd. ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st) (d#D) =
            ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st#map (λd. ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st) D"
        using Cons by auto
      moreover have
          "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ pair (t, s) · δ · I ≠ pair d · I"
        using fun_pair_ineqs[of I _ "(t,s)" I d] Cons.prems(2) by auto
      moreover have "(fv t' ∪ fv s') ∩ set X = {}" using Cons.prems(1) d by auto
      hence "∀δ. subst_domain δ = set X ⟶ pair d · δ = pair d" using d unfolding pair_def by auto
      ultimately show ?case by (simp add: ineq_model_def)
```

```
    qed simp
  } moreover {
    fix δ::"('fun,'var) subst"
    assume "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair d)]⟩st) D⟧d ℐ"
      and δ: "subst_domain δ = set X" "ground (subst_range δ)"
    with assms have "(t,s) ·p δ ·p ℐ ∉ set D ·pset ℐ"
    proof (induction D)
      case (Cons d D)
      obtain t' s' where d: "d = (t',s')" by moura
      have "(t,s) ·p δ ·p ℐ ∉ set D ·pset ℐ"
           "pair (t,s) · δ · ℐ ≠ pair d · δ · ℐ"
        using Cons d by (auto simp add: ineq_model_def simp del: subst_range.simps)
      moreover have "pair d · δ = pair d"
        using Cons.prems(1) fun_pair_subst[of d δ] d δ(1) unfolding pair_def by auto
      ultimately show ?case unfolding pair_def by force
    qed simp
  } ultimately show ?thesis by metis
qed

lemma tr_NegChecks_constr_iff:
  "(∀G∈set L. ineq_model ℐ X (F@G)) ⟷ ⟦M; map (λG. ∀X⟨∨≠: (F@G)⟩st) L⟧d ℐ" (is ?A)
  "negchecks_model ℐ D X F F' ⟷ ⟦M; D; [∀X⟨∨≠: F ∨∉: F'⟩]⟧s ℐ" (is ?B)
proof -
  show ?A by (induct L) auto
  show ?B by simp
qed

lemma tr_pairs_sem_equiv:
  fixes ℐ::"('fun,'var) subst"
  assumes "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ set X = {}"
  shows "negchecks_model ℐ (set D ·pset ℐ) X F F' ⟷
         (∀G ∈ set (tr_pairs F' D). ineq_model ℐ X (F@G))"
proof -
  define P where
    "P ≡ λδ::('fun,'var) subst. subst_domain δ = set X ∧ ground (subst_range δ)"

  define Ineq where
    "Ineq ≡ λ(δ::('fun,'var) subst) F. list_ex (λf. fst f · δ ∘s ℐ ≠ snd f · δ ∘s ℐ) F"

  define Ineq' where
    "Ineq' ≡ λ(δ::('fun,'var) subst) F. list_ex (λf. fst f · δ ∘s ℐ ≠ snd f · ℐ) F"

  define Notin where
    "Notin ≡ λ(δ::('fun,'var) subst) D F'. list_ex (λf. f ·p δ ∘s ℐ ∉ set D ·pset ℐ) F'"

  have sublmm:
      "((s,t) ·p δ ∘s ℐ ∉ set D ·pset ℐ) ⟷ (list_all (λd. Ineq' δ [(pair (s,t),pair d)]) D)"
    for s t δ D
    unfolding pair_def by (induct D) (auto simp add: Ineq'_def)

  have "Notin δ D F' ⟷ (∀G ∈ set (tr_pairs F' D). Ineq' δ G)"
    (is "?A ⟷ ?B")
    when "P δ" for δ
  proof
    show "?A ⟹ ?B"
    proof (induction F' D rule: tr_pairs.induct)
      case (2 s t F' D)
      show ?case
      proof (cases "Notin δ D F'")
        case False
        hence "(s,t) ·p δ ∘s ℐ ∉ set D ·pset ℐ"
          using "2.prems"
          by (auto simp add: Notin_def)
```

```
        hence "pair (s,t) · δ ∘s I ≠ pair d · I" when "d ∈ set D" for d
          using that sublmm Ball_set[of D "λd. Ineq' δ [(pair (s,t), pair d)]"]
          by (simp add: Ineq'_def)
        moreover have "∃d ∈ set D. ∃G'. G = (pair (s,t), pair d)#G'"
          when "G ∈ set (tr_pairs ((s,t)#F') D)" for G
          using that tr_pairs_index[OF that, of 0] by force
        ultimately show ?thesis by (simp add: Ineq'_def)
      qed (auto dest: "2.IH" simp add: Ineq'_def)
    qed (simp add: Notin_def)

    have "¬?A ⟹ ¬?B"
    proof (induction F' D rule: tr_pairs.induct)
      case (2 s t F' D)
      then obtain G where G: "G ∈ set (tr_pairs F' D)" "¬Ineq' δ G"
        by (auto simp add: Notin_def)

      obtain d where d: "d ∈ set D" "pair (s,t) · δ ∘s I = pair d · I"
        using "2.prems"
        unfolding pair_def by (auto simp add: Notin_def)
      thus ?case
        using G(2) tr_pairs_cons[OF G(1) d(1)]
        by (auto simp add: Ineq'_def)
    qed (simp add: Ineq'_def)
    thus "?B ⟹ ?A" by metis
  qed
  hence *: "(∀δ. P δ ⟶ Ineq δ F ∨ Notin δ D F') ⟷
           (∀G ∈ set (tr_pairs F' D). ∀δ. P δ ⟶ Ineq δ F ∨ Ineq' δ G)"
    by auto

  have "snd g · δ = snd g"
    when "G ∈ set (tr_pairs F' D)" "g ∈ set G" "P δ"
    for δ g G
    using assms that(3) tr_pairs_has_pair_lists[OF that(1,2)]
    unfolding pair_def by (fastforce simp add: P_def)
  hence **: "Ineq' δ G = Ineq δ G"
    when "G ∈ set (tr_pairs F' D)" "P δ"
    for δ G
    using Bex_set[of G "λf. fst f · δ ∘s I ≠ snd f · I"]
          Bex_set[of G "λf. fst f · δ ∘s I ≠ snd f · δ ∘s I"]
          that
    by (simp add: Ineq_def Ineq'_def)

  show ?thesis
    using * **
    by (simp add: Ineq_def Ineq'_def Notin_def P_def negchecks_model_def ineq_model_def)
qed

lemma tr_sem_equiv':
  assumes "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvars_sst A = {}"
    and "fv_sst A ∩ bvars_sst A = {}"
    and "ground M"
    and I: "interpretation_subst I"
  shows "⟦M; set D ·pset I; A⟧s I ⟷ (∃A' ∈ set (tr A D). ⟦M; A'⟧d I)" (is "?P ⟷ ?Q")
proof
  have I_grounds: "⋀t. fv (t · I) = {}" by (rule interpretation_grounds[OF I])
  have "∃A' ∈ set (tr A D). ⟦M; A'⟧d I" when ?P using that assms(1,2,3)
  proof (induction A arbitrary: D rule: strand_sem_stateful_induct)
    case (ConsRcv M D t A)
    have "⟦insert (t · I) M; set D ·pset I; A⟧s I"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvars_sst A = {}"
         "fv_sst A ∩ bvars_sst A = {}" "ground (insert (t · I) M)"
      using I ConsRcv.prems unfolding fv_sst_def bvars_sst_def by force+
    then obtain A' where A': "A' ∈ set (tr A D)" "⟦insert (t · I) M; A'⟧d I" by (metis ConsRcv.IH)
```

```
    thus ?case by auto
  next
    case (ConsSnd M D t A)
    have "⟦M; set D ·pset ℐ; A⟧s ℐ"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground M"
      and *: "M ⊢ t · ℐ"
      using ℐ ConsSnd.prems unfolding fvsst_def bvarssst_def by force+
    then obtain A' where A': "A' ∈ set (tr A D)" "⟦M; A'⟧d ℐ" by (metis ConsSnd.IH)
    thus ?case using * by auto
  next
    case (ConsEq M D ac t t' A)
    have "⟦M; set D ·pset ℐ; A⟧s ℐ"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground M"
      and *: "t · ℐ = t' · ℐ"
      using ℐ ConsEq.prems unfolding fvsst_def bvarssst_def by force+
    then obtain A' where A': "A' ∈ set (tr A D)" "⟦M; A'⟧d ℐ" by (metis ConsEq.IH)
    thus ?case using * by auto
  next
    case (ConsIns M D t s A)
    have "⟦M; set (List.insert (t,s) D) ·pset ℐ; A⟧s ℐ"
         "∀(t,t') ∈ set (List.insert (t,s) D). (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground M"
      using ConsIns.prems unfolding fvsst_def bvarssst_def by force+
    then obtain A' where A': "A' ∈ set (tr A (List.insert (t,s) D))" "⟦M; A'⟧d ℐ"
      by (metis ConsIns.IH)
    thus ?case by auto
  next
    case (ConsDel M D t s A)
    have *: "⟦M; (set D ·pset ℐ) - {(t,s) ·p ℐ}; A⟧s ℐ"
            "∀(t,t')∈set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
            "fvsst A ∩ bvarssst A = {}" "ground M"
      using ConsDel.prems unfolding fvsst_def bvarssst_def by force+
    then obtain Di where Di:
        "Di ⊆ set D" "Di ·pset ℐ ⊆ {(t,s) ·p ℐ}" "(t,s) ·p ℐ ∉ (set D - Di) ·pset ℐ"
      using subset_subst_pairs_diff_exists'[of "set D"] by moura
    hence **: "(set D ·pset ℐ) - {(t,s) ·p ℐ} = (set D - Di) ·pset ℐ" by blast

    obtain Di' where Di': "set Di' = Di" "Di' ∈ set (subseqs D)"
      using subset_sublist_exists[OF Di(1)] by moura
    hence ***: "(set D ·pset ℐ) - {(t,s) ·p ℐ} = (set [d←D. d ∉ set Di'] ·pset ℐ)"
      using Di ** by auto

    define constr where "constr ≡
        map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di'@
        map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←D. d ∉ set Di']"

    have ****: "∀(t,t')∈set [d←D. d ∉ set Di']. (fv t ∪ fv t') ∩ bvarssst A = {}"
      using *(2) Di(1) Di'(1) subseqs_set_subset[OF Di'(2)] by simp
    have "set D - Di = set [d←D. d ∉ set Di']" using Di Di' by auto
    hence *****: "⟦M; set [d←D. d ∉ set Di'] ·pset ℐ; A⟧s ℐ"
      using *(1) ** by metis
    obtain A' where A': "A' ∈ set (tr A [d←D. d ∉ set Di'])" "⟦M; A'⟧d ℐ"
      using ConsDel.IH[OF ***** **** *(3,4)] by moura
    hence constr_sat: "⟦M; constr⟧d ℐ"
      using Di Di' *(1) *** tr_Delete_constr_iff[OF *(4), of ℐ Di' t s D]
      unfolding constr_def by auto

    have "constr@A' ∈ set (tr (Delete t s#A) D)" using A'(1) Di' unfolding constr_def by auto
    moreover have "ikst constr = {}" unfolding constr_def by auto
    hence "⟦M ·set ℐ; constr⟧d ℐ" "⟦M ∪ (ikst constr ·set ℐ); A'⟧d ℐ"
      using constr_sat A'(2) subst_all_ground_ident[OF *(4)] by simp_all
```

```
    ultimately show ?case
      using strand_sem_append(2)[of _ _ I]
            subst_all_ground_ident[OF *(4), of I]
      by metis
  next
    case (ConsIn M D ac t s A)
    have "⟦M; set D ·pset I; A⟧s I"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground M"
      and *: "(t,s) ·p I ∈ set D ·pset I"
      using I ConsIn.prems unfolding fvsst_def bvarssst_def by force+
    then obtain A' where A': "A' ∈ set (tr A D)" "⟦M; A'⟧d I" by (metis ConsIn.IH)
    moreover obtain d where "d ∈ set D" "pair (t,s) · I = pair d · I"
      using * unfolding pair_def by auto
    ultimately show ?case using * by auto
  next
    case (ConsNegChecks M D X F F' A)
    let ?ineqs = "(map (λG. ∀X⟨∨≠: (F@G)⟩st) (trpairs F' D))"
    have 1: "⟦M; set D ·pset I; A⟧s I" "ground M" using ConsNegChecks by auto
    have 2: "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}" "fvsst A ∩ bvarssst A = {}"
      using ConsNegChecks.prems(2,3) I unfolding fvsst_def bvarssst_def by fastforce+

    have 3: "negchecks_model I (set D ·pset I) X F F'" using ConsNegChecks.prems(1) by simp
    from 1 2 obtain A' where A': "A' ∈ set (tr A D)" "⟦M; A'⟧d I" by (metis ConsNegChecks.IH)

    have 4: "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ set X = {}"
      using ConsNegChecks.prems(2) unfolding bvarssst_def by auto

    have "⟦M; ?ineqs⟧d I"
      using 3 trpairs_sem_equiv[OF 4] tr_NegChecks_constr_iff
      by metis
    moreover have "ikst ?ineqs = {}" by auto
    moreover have "M ·set I = M" using 1(2) I by (simp add: subst_all_ground_ident)
    ultimately show ?case
      using strand_sem_append(2)[of M ?ineqs I A'] A'
      by force
  qed simp
  thus "?P ⟹ ?Q" by metis

  have "(∃A' ∈ set (tr A D). ⟦M; A'⟧d I) ⟹ ?P" using assms(1,2,3)
  proof (induction A arbitrary: D rule: strand_sem_stateful_induct)
    case (ConsRcv M D t A)
    have "∃A' ∈ set (tr A D). ⟦insert (t · I) M; A'⟧d I"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground (insert (t · I) M)"
      using I ConsRcv.prems unfolding fvsst_def bvarssst_def by force+
    hence "⟦insert (t · I) M; set D ·pset I; A⟧s I" by (metis ConsRcv.IH)
    thus ?case by auto
  next
    case (ConsSnd M D t A)
    have "∃A' ∈ set (tr A D). ⟦M; A'⟧d I"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground M"
      and *: "M ⊢ t · I"
      using I ConsSnd.prems unfolding fvsst_def bvarssst_def by force+
    hence "⟦M; set D ·pset I; A⟧s I" by (metis ConsSnd.IH)
    thus ?case using * by auto
  next
    case (ConsEq M D ac t t' A)
    have "∃A' ∈ set (tr A D). ⟦M; A'⟧d I"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
         "fvsst A ∩ bvarssst A = {}" "ground M"
      and *: "t · I = t' · I"
```

```
      using ℐ ConsEq.prems unfolding fv_sst_def bvars_sst_def by force+
    hence "⟦M; set D ·_pset ℐ; A⟧_s ℐ" by (metis ConsEq.IH)
    thus ?case using * by auto
  next
    case (ConsIns M D t s A)
    hence "∃A' ∈ set (tr A (List.insert (t,s) D)). ⟦M; A'⟧_d ℐ"
          "∀(t,t') ∈ set (List.insert (t,s) D). (fv t ∪ fv t') ∩ bvars_sst A = {}"
          "fv_sst A ∩ bvars_sst A = {}" "ground M"
      unfolding fv_sst_def bvars_sst_def by auto+
    hence "⟦M; set (List.insert (t,s) D) ·_pset ℐ; A⟧_s ℐ" by (metis ConsIns.IH)
    thus ?case by auto
  next
    case (ConsDel M D t s A)
    define constr where "constr ≡
      λDi. map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩_st) Di@
           map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩_st) [d←D. d ∉ set Di]"
    let ?flt = "λDi. filter (λd. d ∉ set Di) D"

    have "∃Di ∈ set (subseqs D). ∃B' ∈ set (tr A (?flt Di)). B = constr Di@B'"
      when "B ∈ set (tr (delete⟨t,s⟩#A) D)" for B
      using that unfolding constr_def by auto
    then obtain A' Di where A':
        "constr Di@A' ∈ set (tr (Delete t s#A) D)"
        "A' ∈ set (tr A (?flt Di))"
        "Di ∈ set (subseqs D)"
        "⟦M; constr Di@A'⟧_d ℐ"
      using ConsDel.prems(1) by blast

    have 1: "∀(t,t')∈set (?flt Di). (fv t ∪ fv t') ∩ bvars_sst A = {}" using ConsDel.prems(2) by
auto
    have 2: "fv_sst A ∩ bvars_sst A = {}" using ConsDel.prems(3) by force+
    have "ik_st (constr Di) = {}" unfolding constr_def by auto
    hence 3: "⟦M; A'⟧_d ℐ"
      using subst_all_ground_ident[OF ConsDel.prems(4)] A'(4)
            strand_sem_split(4)[of M "constr Di" A' ℐ]
      by simp
    have IH: "⟦M; set (?flt Di) ·_pset ℐ; A⟧_s ℐ"
      by (metis ConsDel.IH[OF _ 1 2 ConsDel.prems(4)] 3 A'(2))

    have "⟦M; constr Di⟧_d ℐ"
      using subst_all_ground_ident[OF ConsDel.prems(4)] strand_sem_split(3) A'(4)
      by metis
    hence *: "set Di ·_pset ℐ ⊆ {(t,s) ·_p ℐ}" "(t,s) ·_p ℐ ∉ (set D - set Di) ·_pset ℐ"
      using tr_Delete_constr_iff[OF ConsDel.prems(4), of ℐ Di t s D] unfolding constr_def by auto
    have 4: "set (?flt Di) ·_pset ℐ = (set D ·_pset ℐ) - {((t,s) ·_p ℐ)}"
    proof
      show "set (?flt Di) ·_pset ℐ ⊆ (set D ·_pset ℐ) - {((t,s) ·_p ℐ)}"
      proof
        fix u u' assume u: "(u,u') ∈ set (?flt Di) ·_pset ℐ"
        then obtain v v' where v: "(v,v') ∈ set D - set Di" "(v,v') ·_p ℐ = (u,u')" by auto
        hence "(u,u') ≠ (t,s) ·_p ℐ" using * by force
        thus "(u,u') ∈  (set D ·_pset ℐ) - {((t,s) ·_p ℐ)}"
          using u v * subseqs_set_subset[OF A'(3)] by auto
      qed
      show "(set D ·_pset ℐ) - {((t,s) ·_p ℐ)} ⊆ set (?flt Di) ·_pset ℐ"
        using * subseqs_set_subset[OF A'(3)] by force
    qed

    show ?case using 4 IH by simp
  next
    case (ConsIn M D ac t s A)
    have "∃A' ∈ set (tr A D). ⟦M; A'⟧_d ℐ"
         "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvars_sst A = {}"
```

```
          "fvsst A ∩ bvarssst A = {}" "ground M"
        and *: "(t,s) ·p 𝓘 ∈ set D ·pset 𝓘"
        using ConsIn.prems(1,2,3,4) apply (fastforce, fastforce, fastforce, fastforce)
        using ConsIn.prems(1) tr.simps(7)[of ac t s A D] unfolding pair_def by fastforce
      hence "⟦M; set D ·pset 𝓘; A⟧s 𝓘" by (metis ConsIn.IH)
      moreover obtain d where "d ∈ set D" "pair (t,s) · 𝓘 = pair d · 𝓘"
        using * unfolding pair_def by auto
      ultimately show ?case using * by auto
    next
      case (ConsNegChecks M D X F F' A)
      let ?ineqs = "(map (λG. ∀X⟨∨≠: (F@G)⟩st) (trpairs F' D))"

      obtain B where B:
          "?ineqs@B ∈ set (tr (NegChecks X F F'#A) D)" "⟦M; ?ineqs@B⟧d 𝓘" "B ∈ set (tr A D)"
        using ConsNegChecks.prems(1) by moura
      moreover have "M ·set 𝓘 = M"
        using ConsNegChecks.prems(4) 𝓘 by (simp add: subst_all_ground_ident)
      moreover have "ikst ?ineqs = {}" by auto
      ultimately have "⟦M; B⟧d 𝓘" using strand_sem_split(4)[of M ?ineqs B 𝓘] by simp
      moreover have "∀(t,t')∈set D. (fv t ∪ fv t') ∩ bvarssst A = {}" "fvsst A ∩ bvarssst A = {}"
        using ConsNegChecks.prems(2,3) unfolding fvsst_def bvarssst_def by force+
      ultimately have "⟦M; set D ·pset 𝓘; A⟧s 𝓘"
        by (metis ConsNegChecks.IH B(3) ConsNegChecks.prems(4))
      moreover have "∀(t, t')∈set D. (fv t ∪ fv t') ∩ set X = {}"
        using ConsNegChecks.prems(2) unfolding bvarssst_def by force
      ultimately show ?case
        using trpairs_sem_equiv tr_NegChecks_constr_iff
              B(2) strand_sem_split(3)[of M ?ineqs B 𝓘] ‹M ·set 𝓘 = M›
        by simp
    qed simp
    thus "?Q ⟹ ?P" by metis
qed

lemma tr_sem_equiv:
  assumes "fvsst A ∩ bvarssst A = {}" and "interpretationsubst 𝓘"
  shows "𝓘 ⊨s A ⟷ (∃A' ∈ set (tr A []). (𝓘 ⊨ ⟨A'⟩))"
using tr_sem_equiv'[OF _ assms(1) _ assms(2), of "[]" "{}"]
unfolding constr_sem_d_def
by auto

theorem stateful_typing_result:
  assumes "wfsst 𝒜"
    and "tfrsst 𝒜"
    and "wftrms (trmssst 𝒜)"
    and "interpretationsubst 𝓘"
    and "𝓘 ⊨s 𝒜"
  obtains 𝓘τ
    where "interpretationsubst 𝓘τ"
    and "𝓘τ ⊨s 𝒜"
    and "wtsubst 𝓘τ"
    and "wftrms (subst_range 𝓘τ)"
proof -
  obtain 𝒜' where 𝒜':
      "𝒜' ∈ set (tr 𝒜 [])" "𝓘 ⊨ ⟨𝒜'⟩"
    using tr_sem_equiv[of 𝒜] assms(1,4,5)
    by auto

  have *: "wfst {} 𝒜'"
          "fvst 𝒜' ∩ bvarsst 𝒜' = {}"
          "tfrst 𝒜'" "wftrms (trmsst 𝒜')"
    using tr_wf[OF 𝒜'(1) assms(1,3)]
          tr_tfr[OF 𝒜'(1) assms(2)] assms(1)
    by metis+
```

```
  obtain ℐτ where ℐτ:
      "interpretation_subst ℐτ" "⟦{}; 𝒜'⟧d ℐτ"
      "wt_subst ℐτ" "wf_trms (subst_range ℐτ)"
    using wt_attack_if_tfr_attack_d
          * Ana_invar_subst' assms(4)
          𝒜'(2)
    unfolding constr_sem_d_def
    by moura

  thus ?thesis
    using that tr_sem_equiv[of 𝒜] assms(1,3) 𝒜'(1)
    unfolding constr_sem_d_def
    by auto
qed

end

end
```

4.2.3 Proving type-flaw resistance automatically

```
definition pair' where
  "pair' pair_fun d ≡ case d of (t,t') ⇒ Fun pair_fun [t,t']"

fun comp_tfr_sstp where
  "comp_tfr_sstp Γ pair_fun (⟨_: t ≐ t'⟩) = (mgu t t' ≠ None ⟶ Γ t = Γ t')"
| "comp_tfr_sstp Γ pair_fun (∀X⟨∨≠: F ∨∉: F'⟩) = (
    (F' = [] ∧ (∀x ∈ fv_pairs F - set X. is_Var (Γ (Var x)))) ∨
    (∀u ∈ subterms_set (trms_pairs F ∪ pair' pair_fun ` set F').
      is_Fun u ⟶ (args u = [] ∨ (∃s ∈ set (args u). s ∉ Var ` set X))))"
| "comp_tfr_sstp _ _ _ = True"

definition comp_tfr_sst where
  "comp_tfr_sst arity Ana Γ pair_fun M S ≡
    list_all (comp_tfr_sstp Γ pair_fun) S ∧
    list_all (wf_trm' arity) (trms_list_sst S) ∧
    has_all_wt_instances_of Γ (trms_sst S ∪ pair' pair_fun ` setops_sst S) (set M) ∧
    comp_tfr_set arity Ana Γ M"

locale stateful_typed_model' = stateful_typed_model arity public Ana Γ Pair
  for arity::"'fun ⇒ nat"
    and public::"'fun ⇒ bool"
    and Ana::"('fun,(('fun,'atom::finite) term_type × nat)) term
             ⇒ (('fun,(('fun,'atom) term_type × nat)) term list
                × ('fun,(('fun,'atom) term_type × nat)) term list)"
    and Γ::"('fun,(('fun,'atom) term_type × nat)) term ⇒ ('fun,'atom) term_type"
    and Pair::"'fun"
  +
  assumes Γ_Var_fst': "⋀τ n m. Γ (Var (τ,n)) = Γ (Var (τ,m))"
    and Ana_const': "⋀c T. arity c = 0 ⟹ Ana (Fun c T) = ([], [])"
begin

sublocale typed_model'
by (unfold_locales, rule Γ_Var_fst', metis Ana_const', metis Ana_subst')

lemma pair_code:
  "pair d = pair' Pair d"
by (simp add: pair_def pair'_def)

lemma tfr_sstp_is_comp_tfr_sstp: "tfr_sstp a = comp_tfr_sstp Γ Pair a"
proof (cases a)
  case (Equality ac t t')
```

```
  thus ?thesis
    using mgu_always_unifies[of t _ t'] mgu_gives_MGU[of t t']
    by auto
next
  case (NegChecks X F F')
  thus ?thesis
    using tfr_sstp.simps(2)[of X F F']
          comp_tfr_sstp.simps(2)[of Γ Pair X F F']
          Fun_range_case(2)[of "subterms_set (trms_pairs F ∪ pair ` set F')"]
    unfolding is_Var_def pair_code[symmetric]
    by auto
qed auto

lemma tfr_sst_if_comp_tfr_sst:
  assumes "comp_tfr_sst arity Ana Γ Pair M S"
  shows "tfr_sst S"
unfolding tfr_sst_def
proof
  have comp_tfr_set_M: "comp_tfr_set arity Ana Γ M"
    using assms unfolding comp_tfr_sst_def by blast

  have wf_trms_M: "wf_trms (set M)"
      and wf_trms_S: "wf_trms (trms_sst S ∪ pair ` setops_sst S)"
      and S_trms_instance_M: "has_all_wt_instances_of Γ (trms_sst S ∪ pair ` setops_sst S) (set M)"
    using assms setops_sst_wf_trms(2)[of S] trms_list_sst_is_trms_sst[of S]
    unfolding comp_tfr_sst_def comp_tfr_set_def list_all_iff pair_code[symmetric] wf_trm_code[symmetric]
              finite_SMP_representation_def
    by (meson, meson, blast, meson)

  show "tfr_set (trms_sst S ∪ pair ` setops_sst S)"
    using tfr_subset(3)[OF tfr_set_if_comp_tfr_set[OF comp_tfr_set_M] SMP_SMP_subset]
          SMP_I'[OF wf_trms_S wf_trms_M S_trms_instance_M]
    by blast

  have "list_all (comp_tfr_sstp Γ Pair) S" by (metis assms comp_tfr_sst_def)
  thus "list_all tfr_sstp S" by (induct S) (simp_all add: tfr_sstp_is_comp_tfr_sstp)
qed

lemma tfr_sst_if_comp_tfr_sst':
  assumes "comp_tfr_sst arity Ana Γ Pair (SMP0 Ana Γ (trms_list_sst S@map pair (setops_list_sst S))) S"
  shows "tfr_sst S"
by (rule tfr_sst_if_comp_tfr_sst[OF assms])

end

end
```

5 The Parallel Composition Result for Non-Stateful Protocols

In this chapter, we formalize and prove a compositionality result for security protocols. This work is an extension of the work described in [4] and [1, chapter 5].

5.1 Labeled Strands (Labeled_Strands)

theory *Labeled_Strands*
imports *Strands_and_Constraints*
begin

5.1.1 Definitions: Labeled Strands and Constraints

```
datatype 'l strand_label =
  LabelN (the_LabelN: "'l") ("ln _")
| LabelS ("⋆")
```

Labeled strands are strands whose steps are equipped with labels

```
type_synonym ('a,'b,'c) labeled_strand_step = "'c strand_label × ('a,'b) strand_step"
type_synonym ('a,'b,'c) labeled_strand = "('a,'b,'c) labeled_strand_step list"

abbreviation is_LabelN where "is_LabelN n x ≡ fst x = ln n"
abbreviation is_LabelS where "is_LabelS x ≡ fst x = ⋆"

definition unlabel where "unlabel S ≡ map snd S"
definition proj where "proj n S ≡ filter (λs. is_LabelN n s ∨ is_LabelS s) S"
abbreviation proj_unl where "proj_unl n S ≡ unlabel (proj n S)"

abbreviation wfrestrictedvars_lst where "wfrestrictedvars_lst S ≡ wfrestrictedvars_st (unlabel S)"

abbreviation subst_apply_labeled_strand_step (infix "·_lstp" 51) where
  "x ·_lstp ϑ ≡ (case x of (l, s) ⇒ (l, s ·_stp ϑ))"

abbreviation subst_apply_labeled_strand (infix "·_lst" 51) where
  "S ·_lst ϑ ≡ map (λx. x ·_lstp ϑ) S"

abbreviation trms_lst where "trms_lst S ≡ trms_st (unlabel S)"
abbreviation trms_proj_lst where "trms_proj_lst n S ≡ trms_st (proj_unl n S)"

abbreviation vars_lst where "vars_lst S ≡ vars_st (unlabel S)"
abbreviation vars_proj_lst where "vars_proj_lst n S ≡ vars_st (proj_unl n S)"

abbreviation bvars_lst where "bvars_lst S ≡ bvars_st (unlabel S)"
abbreviation fv_lst where "fv_lst S ≡ fv_st (unlabel S)"

abbreviation wf_lst where "wf_lst V S ≡ wf_st V (unlabel S)"
```

5.1.2 Lemmata: Projections

```
lemma is_LabelS_proj_iff_not_is_LabelN:
  "list_all is_LabelS (proj l A) ⟷ ¬list_ex (is_LabelN l) A"
by (induct A) (auto simp add: proj_def)

lemma proj_subset_if_no_label:
```

```
  assumes "¬list_ex (is_LabelN l) A"
  shows "set (proj l A) ⊆ set (proj l' A)"
    and "set (proj_unl l A) ⊆ set (proj_unl l' A)"
using assms by (induct A) (auto simp add: unlabel_def proj_def)

lemma proj_in_setD:
  assumes a: "a ∈ set (proj l A)"
  obtains k b where "a = (k, b)" "k = (ln l) ∨ k = ⋆"
using that a unfolding proj_def by (cases a) auto

lemma proj_set_mono:
  assumes "set A ⊆ set B"
  shows "set (proj n A) ⊆ set (proj n B)"
    and "set (proj_unl n A) ⊆ set (proj_unl n B)"
using assms unfolding proj_def unlabel_def by auto

lemma unlabel_nil[simp]: "unlabel [] = []"
by (simp add: unlabel_def)

lemma unlabel_mono: "set A ⊆ set B ⟹ set (unlabel A) ⊆ set (unlabel B)"
by (auto simp add: unlabel_def)

lemma unlabel_in: "(l,x) ∈ set A ⟹ x ∈ set (unlabel A)"
unfolding unlabel_def by force

lemma unlabel_mem_has_label: "x ∈ set (unlabel A) ⟹ ∃l. (l,x) ∈ set A"
unfolding unlabel_def by auto

lemma proj_nil[simp]: "proj n [] = []" "proj_unl n [] = []"
unfolding unlabel_def proj_def by auto

lemma singleton_lst_proj[simp]:
  "proj_unl l [(ln l, a)] = [a]"
  "l ≠ l' ⟹ proj_unl l' [(ln l, a)] = []"
  "proj_unl l [(⋆, a)] = [a]"
  "unlabel [(l'', a)] = [a]"
unfolding proj_def unlabel_def by simp_all

lemma unlabel_nil_only_if_nil[simp]: "unlabel A = [] ⟹ A = []"
unfolding unlabel_def by auto

lemma unlabel_Cons[simp]:
  "unlabel ((l,a)#A) = a#unlabel A"
  "unlabel (b#A) = snd b#unlabel A"
unfolding unlabel_def by simp_all

lemma unlabel_append[simp]: "unlabel (A@B) = unlabel A@unlabel B"
unfolding unlabel_def by auto

lemma proj_Cons[simp]:
  "proj n ((ln n,a)#A) = (ln n,a)#proj n A"
  "proj n ((⋆,a)#A) = (⋆,a)#proj n A"
  "m ≠ n ⟹ proj n ((ln m,a)#A) = proj n A"
  "l = (ln n) ⟹ proj n ((l,a)#A) = (l,a)#proj n A"
  "l = ⋆ ⟹ proj n ((l,a)#A) = (l,a)#proj n A"
  "fst b ≠ ⋆ ⟹ fst b ≠ (ln n) ⟹ proj n (b#A) = proj n A"
unfolding proj_def by auto

lemma proj_append[simp]:
  "proj l (A'@B') = proj l A'@proj l B'"
  "proj_unl l (A@B) = proj_unl l A@proj_unl l B"
unfolding proj_def unlabel_def by auto
```

```
lemma proj_unl_cons[simp]:
  "proj_unl l ((ln l, a)#A) = a#proj_unl l A"
  "l ≠ l' ⟹ proj_unl l' ((ln l, a)#A) = proj_unl l' A"
  "proj_unl l ((⋆, a)#A) = a#proj_unl l A"
unfolding proj_def unlabel_def by simp_all

lemma trms_unlabel_proj[simp]:
  "trms_stp (snd (ln l, x)) ⊆ trms_proj_lst l [(ln l, x)]"
by auto

lemma trms_unlabel_star[simp]:
  "trms_stp (snd (⋆, x)) ⊆ trms_proj_lst l [(⋆, x)]"
by auto

lemma trms_lst_union[simp]: "trms_lst A = (⋃l. trms_proj_lst l A)"
proof (induction A)
  case (Cons a A)
  obtain l s where ls: "a = (l,s)" by moura
  have "trms_lst [a] = (⋃l. trms_proj_lst l [a])"
  proof -
    have *: "trms_lst [a] = trms_stp s" using ls by simp
    show ?thesis
    proof (cases l)
      case (LabelN n)
      hence "trms_proj_lst n [a] = trms_stp s" using ls by simp
      moreover have "∀m. n ≠ m ⟶ trms_proj_lst m [a] = {}" using ls LabelN by auto
      ultimately show ?thesis using * ls by fastforce
    next
      case LabelS
      hence "∀l. trms_proj_lst l [a] = trms_stp s" using ls by auto
      thus ?thesis using * ls by fastforce
    qed
  qed
  moreover have "∀l. trms_proj_lst l (a#A) = trms_proj_lst l [a] ∪ trms_proj_lst l A"
    unfolding unlabel_def proj_def by auto
  hence "(⋃l. trms_proj_lst l (a#A)) = (⋃l. trms_proj_lst l [a]) ∪ (⋃l. trms_proj_lst l A)" by auto
  ultimately show ?case using Cons.IH ls by auto
qed simp

lemma trms_lst_append[simp]: "trms_lst (A@B) = trms_lst A ∪ trms_lst B"
by (metis trms_st_append unlabel_append)

lemma trms_proj_lst_append[simp]: "trms_proj_lst l (A@B) = trms_proj_lst l A ∪ trms_proj_lst l B"
by (metis (no_types, lifting) filter_append proj_def trms_lst_append)

lemma trms_proj_lst_subset[simp]:
  "trms_proj_lst l A ⊆ trms_proj_lst l (A@B)"
  "trms_proj_lst l B ⊆ trms_proj_lst l (A@B)"
using trms_proj_lst_append[of l] by blast+

lemma trms_lst_subset[simp]:
  "trms_lst A ⊆ trms_lst (A@B)"
  "trms_lst B ⊆ trms_lst (A@B)"
proof (induction A)
  case (Cons a A)
  obtain l s where *: "a = (l,s)" by moura
  { case 1 thus ?case using Cons * by auto }
  { case 2 thus ?case using Cons * by auto }
qed simp_all

lemma vars_lst_union: "vars_lst A = (⋃l. vars_proj_lst l A)"
proof (induction A)
  case (Cons a A)
```

```
  obtain l s where ls: "a = (l,s)" by moura
  have "vars_lst [a] = (⋃l. vars_proj_lst l [a])"
  proof -
    have *: "vars_lst [a] = vars_stp s" using ls by auto
    show ?thesis
    proof (cases l)
      case (LabelN n)
      hence "vars_proj_lst n [a] = vars_stp s" using ls by simp
      moreover have "∀m. n ≠ m ⟶ vars_proj_lst m [a] = {}" using ls LabelN by auto
      ultimately show ?thesis using * ls by fast
    next
      case LabelS
      hence "∀l. vars_proj_lst l [a] = vars_stp s" using ls by auto
      thus ?thesis using * ls by fast
    qed
  qed
  moreover have "∀l. vars_proj_lst l (a#A) = vars_proj_lst l [a] ∪ vars_proj_lst l A"
    unfolding unlabel_def proj_def by auto
  hence "(⋃l. vars_proj_lst l (a#A)) = (⋃l. vars_proj_lst l [a]) ∪ (⋃l. vars_proj_lst l A)"
    using strand_vars_split(1) by auto
  ultimately show ?case using Cons.IH ls strand_vars_split(1) by auto
qed simp

lemma unlabel_Cons_inv:
  "unlabel A = b#B ⟹ ∃A'. (∃n. A = (ln n, b)#A') ∨ A = (⋆, b)#A'"
proof -
  assume *: "unlabel A = b#B"
  then obtain l A' where "A = (l,b)#A'" unfolding unlabel_def by moura
  thus "∃A'. (∃l. A = (ln l, b)#A') ∨ A = (⋆, b)#A'" by (metis strand_label.exhaust)
qed

lemma unlabel_snoc_inv:
  "unlabel A = B@[b] ⟹ ∃A'. (∃n. A = A'@[(ln n, b)]) ∨ A = A'@[(⋆, b)]"
proof -
  assume *: "unlabel A = B@[b]"
  then obtain A' l where "A = A'@[(l,b)]"
    unfolding unlabel_def by (induct A rule: List.rev_induct) auto
  thus "∃A'. (∃n. A = A'@[(ln n, b)]) ∨ A = A'@[(⋆, b)]" by (cases l) auto
qed

lemma proj_idem[simp]: "proj l (proj l A) = proj l A"
unfolding proj_def by auto

lemma proj_ik_st_is_proj_rcv_set:
  "ik_st (proj_unl n A) = {t. (ln n, Receive t) ∈ set A ∨ (⋆, Receive t) ∈ set A} "
using ik_st_is_rcv_set unfolding unlabel_def proj_def by force

lemma unlabel_ik_st_is_rcv_set:
  "ik_st (unlabel A) = {t | l t. (l, Receive t) ∈ set A}"
using ik_st_is_rcv_set unfolding unlabel_def by force

lemma proj_ik_union_is_unlabel_ik:
  "ik_st (unlabel A) = (⋃l. ik_st (proj_unl l A))"
proof
  show "(⋃l. ik_st (proj_unl l A)) ⊆ ik_st (unlabel A)"
    using unlabel_ik_st_is_rcv_set[of A] proj_ik_st_is_proj_rcv_set[of _ A] by auto

  show "ik_st (unlabel A) ⊆ (⋃l. ik_st (proj_unl l A))"
  proof
    fix t assume "t ∈ ik_st (unlabel A)"
    then obtain l where "(l, Receive t) ∈ set A"
      using ik_st_is_rcv_set unlabel_mem_has_label[of _ A]
      by moura
```

```
    thus "t ∈ (⋃l. ik_st (proj_unl l A))" using proj_ik_st_is_proj_rcv_set[of _ A] by (cases l) auto
  qed
qed

lemma proj_ik_append[simp]:
  "ik_st (proj_unl l (A@B)) = ik_st (proj_unl l A) ∪ ik_st (proj_unl l B)"
using proj_append(2)[of l A B] ik_append by auto

lemma proj_ik_append_subst_all:
  "ik_st (proj_unl l (A@B)) ·_set I = (ik_st (proj_unl l A) ·_set I) ∪ (ik_st (proj_unl l B) ·_set I)"
using proj_ik_append[of l] by auto

lemma ik_proj_subset[simp]: "ik_st (proj_unl n A) ⊆ trms_proj_lst n A"
by auto

lemma prefix_proj:
  "prefix A B ⟹ prefix (unlabel A) (unlabel B)"
  "prefix A B ⟹ prefix (proj n A) (proj n B)"
  "prefix A B ⟹ prefix (proj_unl n A) (proj_unl n B)"
unfolding prefix_def unlabel_def proj_def by auto
```

5.1.3 Lemmata: Well-formedness

```
lemma wfvarsoccs_st_proj_union:
  "wfvarsoccs_st (unlabel A) = (⋃l. wfvarsoccs_st (proj_unl l A))"
proof (induction A)
  case (Cons a A)
  obtain l s where ls: "a = (l,s)" by moura
  have "wfvarsoccs_st (unlabel [a]) = (⋃l. wfvarsoccs_st (proj_unl l [a]))"
  proof -
    have *: "wfvarsoccs_st (unlabel [a]) = wfvarsoccs_stp s" using ls by auto
    show ?thesis
    proof (cases l)
      case (LabelN n)
      hence "wfvarsoccs_st (proj_unl n [a]) = wfvarsoccs_stp s" using ls by simp
      moreover have "∀m. n ≠ m ⟶ wfvarsoccs_st (proj_unl m [a]) = {}" using ls LabelN by auto
      ultimately show ?thesis using * ls by fast
    next
      case LabelS
      hence "∀l. wfvarsoccs_st (proj_unl l [a]) = wfvarsoccs_stp s" using ls by auto
      thus ?thesis using * ls by fast
    qed
  qed
  moreover have
      "wfvarsoccs_st (proj_unl l (a#A)) =
       wfvarsoccs_st (proj_unl l [a]) ∪ wfvarsoccs_st (proj_unl l A)"
    for l
    unfolding unlabel_def proj_def by auto
  hence "(⋃l. wfvarsoccs_st (proj_unl l (a#A))) =
         (⋃l. wfvarsoccs_st (proj_unl l [a])) ∪ (⋃l. wfvarsoccs_st (proj_unl l A))"
    using strand_vars_split(1) by auto
  ultimately show ?case using Cons.IH ls strand_vars_split(1) by auto
qed simp

lemma wf_if_wf_proj:
  assumes "∀l. wf_st V (proj_unl l A)"
  shows "wf_st V (unlabel A)"
using assms
proof (induction A arbitrary: V rule: List.rev_induct)
  case (snoc a A)
  hence IH: "wf_st V (unlabel A)" using proj_append(2)[of _ A] by auto
  obtain b l where b: "a = (ln l, b) ∨ a = (⋆, b)" by (cases a, metis strand_label.exhaust)
  hence *: "wf_st V (proj_unl l A@[b])"
```

```
    by (metis snoc.prems proj_append(2) singleton_lst_proj(1) proj_unl_cons(1,3))
  thus ?case using IH b snoc.prems proj_append(2)[of l A "[a]"] unlabel_append[of A "[a]"]
  proof (cases b)
    case (Receive t)
    have "fv t ⊆ wfvarsoccs_st (unlabel A) ∪ V"
    proof
      fix x assume "x ∈ fv t"
      hence "x ∈ V ∪ wfvarsoccs_st (proj_unl l A)" using wf_append_exec[OF *] b Receive by auto
      thus "x ∈ wfvarsoccs_st (unlabel A) ∪ V" using wfvarsoccs_st_proj_union[of A] by auto
    qed
    hence "fv t ⊆ wfrestrictedvars_st (unlabel A) ∪ V"
      using vars_snd_rcv_strand_subset2(4)[of "unlabel A"] by blast
    hence "wf_st V (unlabel A@[Receive t])" by (rule wf_rcv_append'''[OF IH])
    thus ?thesis using b Receive unlabel_append[of A "[a]"] by auto
  next
    case (Equality ac s t)
    have "fv t ⊆ wfvarsoccs_st (unlabel A) ∪ V" when "ac = Assign"
    proof
      fix x assume "x ∈ fv t"
      hence "x ∈ V ∪ wfvarsoccs_st (proj_unl l A)" using wf_append_exec[OF *] b Equality that by auto
      thus "x ∈ wfvarsoccs_st (unlabel A) ∪ V" using wfvarsoccs_st_proj_union[of A] by auto
    qed
    hence "fv t ⊆ wfrestrictedvars_lst A ∪ V" when "ac = Assign"
      using vars_snd_rcv_strand_subset2(4)[of "unlabel A"] that by blast
    hence "wf_st V (unlabel A@[Equality ac s t])"
      by (cases ac) (metis wf_eq_append'''[OF IH], metis wf_eq_check_append''[OF IH])
    thus ?thesis using b Equality unlabel_append[of A "[a]"] by auto
  qed auto
qed simp

end
```

5.2 Parallel Compositionality of Security Protocols (Parallel_Compositionality)

```
theory Parallel_Compositionality
imports Typing_Result Labeled_Strands
begin
```

5.2.1 Definitions: Labeled Typed Model Locale

```
locale labeled_typed_model = typed_model arity public Ana Γ
  for arity::"'fun ⇒ nat"
    and public::"'fun ⇒ bool"
    and Ana::"('fun,'var) term ⇒ (('fun,'var) term list × ('fun,'var) term list)"
    and Γ::"('fun,'var) term ⇒ ('fun,'atom::finite) term_type"
  +
  fixes label_witness1 and label_witness2::"'lbl"
  assumes at_least_2_labels: "label_witness1 ≠ label_witness2"
begin
```

The Ground Sub-Message Patterns (GSMP)

```
definition GSMP::"('fun,'var) terms ⇒ ('fun,'var) terms" where
  "GSMP P ≡ {t ∈ SMP P. fv t = {}}"

definition typing_cond where
  "typing_cond 𝒜 ≡
    wf_st {} 𝒜 ∧
    fv_st 𝒜 ∩ bvars_st 𝒜 = {} ∧
    tfr_st 𝒜 ∧
    wf_trms (trms_st 𝒜) ∧
    Ana_invar_subst (ik_st 𝒜 ∪ assignment_rhs_st 𝒜)"
```

5.2.2 Definitions: GSMP Disjointedness and Parallel Composability

```
definition GSMP_disjoint where
  "GSMP_disjoint P1 P2 Secrets ≡ GSMP P1 ∩ GSMP P2 ⊆ Secrets ∪ {m. {} ⊢c m}"

definition declassified_lst where
  "declassified_lst (𝒜::('fun,'var,'lbl) labeled_strand) ℐ ≡ {t. (⋆, Receive t) ∈ set 𝒜} ·set ℐ"

definition par_comp where
  "par_comp (𝒜::('fun,'var,'lbl) labeled_strand) (Secrets::('fun,'var) terms) ≡
    (∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (trms_proj_lst l1 𝒜) (trms_proj_lst l2 𝒜) Secrets) ∧
    (∀s ∈ Secrets. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Secrets) ∧
    ground Secrets"

definition strand_leaks_lst where
  "strand_leaks_lst 𝒜 Sec ℐ ≡ (∃t ∈ Sec - declassified_lst 𝒜 ℐ. ∃l. (ℐ ⊨ ⟨proj_unl l 𝒜@[Send t]⟩))"
```

5.2.3 Definitions: Homogeneous and Numbered Intruder Deduction Variants

```
definition proj_specific where
  "proj_specific n t 𝒜 Secrets ≡ t ∈ GSMP (trms_proj_lst n 𝒜) - (Secrets ∪ {m. {} ⊢c m})"

definition heterogeneous_lst where
  "heterogeneous_lst t 𝒜 Secrets ≡ (
    (∃l1 l2. ∃s1 ∈ subterms t. ∃s2 ∈ subterms t.
      l1 ≠ l2 ∧ proj_specific l1 s1 𝒜 Secrets ∧ proj_specific l2 s2 𝒜 Secrets))"

abbreviation homogeneous_lst where
  "homogeneous_lst t 𝒜 Secrets ≡ ¬heterogeneous_lst t 𝒜 Secrets"

definition intruder_deduct_hom::
  "('fun,'var) terms ⇒ ('fun,'var,'lbl) labeled_strand ⇒ ('fun,'var) terms ⇒ ('fun,'var) term
  ⇒ bool" ("⟨_;_;_⟩ ⊢hom _" 50)
where
  "⟨M; 𝒜; Sec⟩ ⊢hom t ≡ ⟨M; λt. homogeneous_lst t 𝒜 Sec ∧ t ∈ GSMP (trms_lst 𝒜)⟩ ⊢r t"

lemma intruder_deduct_hom_AxiomH[simp]:
  assumes "t ∈ M"
  shows "⟨M; 𝒜; Sec⟩ ⊢hom t"
using intruder_deduct_restricted.AxiomR[of t M] assms
unfolding intruder_deduct_hom_def
by blast

lemma intruder_deduct_hom_ComposeH[simp]:
  assumes "length X = arity f" "public f" "⋀x. x ∈ set X ⟹ ⟨M; 𝒜; Sec⟩ ⊢hom x"
  and "homogeneous_lst (Fun f X) 𝒜 Sec" "Fun f X ∈ GSMP (trms_lst 𝒜)"
  shows "⟨M; 𝒜; Sec⟩ ⊢hom Fun f X"
proof -
  let ?Q = "λt. homogeneous_lst t 𝒜 Sec ∧ t ∈ GSMP (trms_lst 𝒜)"
  show ?thesis
    using intruder_deduct_restricted.ComposeR[of X f M ?Q] assms
    unfolding intruder_deduct_hom_def
    by blast
qed

lemma intruder_deduct_hom_DecomposeH:
  assumes "⟨M; 𝒜; Sec⟩ ⊢hom t" "Ana t = (K, T)" "⋀k. k ∈ set K ⟹ ⟨M; 𝒜; Sec⟩ ⊢hom k" "t_i ∈ set
T"
  shows "⟨M; 𝒜; Sec⟩ ⊢hom t_i"
proof -
  let ?Q = "λt. homogeneous_lst t 𝒜 Sec ∧ t ∈ GSMP (trms_lst 𝒜)"
  show ?thesis
    using intruder_deduct_restricted.DecomposeR[of M ?Q t] assms
```

```
    unfolding intruder_deduct_hom_def
    by blast
qed

lemma intruder_deduct_hom_induct[consumes 1, case_names AxiomH ComposeH DecomposeH]:
  assumes "⟨M; 𝒜; Sec⟩ ⊢hom t" "⋀t. t ∈ M ⟹ P M t"
          "⋀X f. ⟦length X = arity f; public f;
                  ⋀x. x ∈ set X ⟹ ⟨M; 𝒜; Sec⟩ ⊢hom x;
                  ⋀x. x ∈ set X ⟹ P M x;
                  homogeneouslst (Fun f X) 𝒜 Sec;
                  Fun f X ∈ GSMP (trmslst 𝒜)
                  ⟧ ⟹ P M (Fun f X)"
          "⋀t K T ti. ⟦⟨M; 𝒜; Sec⟩ ⊢hom t; P M t; Ana t = (K, T);
                       ⋀k. k ∈ set K ⟹ ⟨M; 𝒜; Sec⟩ ⊢hom k;
                       ⋀k. k ∈ set K ⟹ P M k; ti ∈ set T⟧ ⟹ P M ti"
        shows "P M t"
proof -
  let ?Q = "λt. homogeneouslst t 𝒜 Sec ∧ t ∈ GSMP (trmslst 𝒜)"
  show ?thesis
    using intruder_deduct_restricted_induct[of M ?Q t "λM Q t. P M t"] assms
    unfolding intruder_deduct_hom_def
    by blast
qed

lemma ideduct_hom_mono:
  "⟦⟨M; 𝒜; Sec⟩ ⊢hom t; M ⊆ M'⟧ ⟹ ⟨M'; 𝒜; Sec⟩ ⊢hom t"
using ideduct_restricted_mono[of M _ t M']
unfolding intruder_deduct_hom_def
by fast
```

5.2.4 Lemmata: GSMP

```
lemma GSMP_disjoint_empty[simp]:
  "GSMP_disjoint {} A Sec" "GSMP_disjoint A {} Sec"
unfolding GSMP_disjoint_def GSMP_def by fastforce+

lemma GSMP_mono:
  assumes "N ⊆ M"
  shows "GSMP N ⊆ GSMP M"
using SMP_mono[OF assms] unfolding GSMP_def by fast

lemma GSMP_SMP_mono:
  assumes "SMP N ⊆ SMP M"
  shows "GSMP N ⊆ GSMP M"
using assms unfolding GSMP_def by fast

lemma GSMP_subterm:
  assumes "t ∈ GSMP M" "t' ⊑ t"
  shows "t' ∈ GSMP M"
using SMP.Subterm[of t M t'] ground_subterm[of t t'] assms unfolding GSMP_def by auto

lemma GSMP_subterms: "subtermsset (GSMP M) = GSMP M"
using GSMP_subterm[of _ M] by blast

lemma GSMP_Ana_key:
  assumes "t ∈ GSMP M" "Ana t = (K,T)" "k ∈ set K"
  shows "k ∈ GSMP M"
using SMP.Ana[of t M K T k] Ana_keys_fv[of t K T] assms unfolding GSMP_def by auto

lemma GSMP_append[simp]: "GSMP (trmslst (A@B)) = GSMP (trmslst A) ∪ GSMP (trmslst B)"
using SMP_union[of "trmslst A" "trmslst B"] trmslst_append[of A B] unfolding GSMP_def by auto

lemma GSMP_union: "GSMP (A ∪ B) = GSMP A ∪ GSMP B"
```

```
using SMP_union[of A B] unfolding GSMP_def by auto

lemma GSMP_Union: "GSMP (trms_lst A) = (⋃l. GSMP (trms_proj_lst l A))"
proof -
  define P where "P ≡ (λl. trms_proj_lst l A)"
  define Q where "Q ≡ trms_lst A"
  have "SMP (⋃l. P l) = (⋃l. SMP (P l))" "Q = (⋃l. P l)"
    unfolding P_def Q_def by (metis SMP_Union, metis trms_lst_union)
  hence "GSMP Q = (⋃l. GSMP (P l))" unfolding GSMP_def by auto
  thus ?thesis unfolding P_def Q_def by metis
qed

lemma in_GSMP_in_proj: "t ∈ GSMP (trms_lst A) ⟹ ∃n. t ∈ GSMP (trms_proj_lst n A)"
using GSMP_Union[of A] by blast

lemma in_proj_in_GSMP: "t ∈ GSMP (trms_proj_lst n A) ⟹ t ∈ GSMP (trms_lst A)"
using GSMP_Union[of A] by blast

lemma GSMP_disjointE:
  assumes A: "GSMP_disjoint (trms_proj_lst n A) (trms_proj_lst m A) Sec"
  shows "GSMP (trms_proj_lst n A) ∩ GSMP (trms_proj_lst m A) ⊆ Sec ∪ {m. {} ⊢c m}"
using assms unfolding GSMP_disjoint_def by auto

lemma GSMP_disjoint_term:
  assumes "GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec"
  shows "t ∉ GSMP (trms_proj_lst l 𝒜) ∨ t ∉ GSMP (trms_proj_lst l' 𝒜) ∨ t ∈ Sec ∨ {} ⊢c t"
using assms unfolding GSMP_disjoint_def by blast

lemma GSMP_wt_subst_subset:
  assumes "t ∈ GSMP (M ·set ℐ)" "wt_subst ℐ" "wf_trms (subst_range ℐ)"
  shows "t ∈ GSMP M"
using SMP_wt_subst_subset[OF _ assms(2,3), of t M] assms(1) unfolding GSMP_def by simp

lemma GSMP_wt_substI:
  assumes "t ∈ M" "wt_subst I" "wf_trms (subst_range I)" "interpretation_subst I"
  shows "t · I ∈ GSMP M"
proof -
  have "t ∈ SMP M" using assms(1) by auto
  hence *: "t · I ∈ SMP M" using SMP.Substitution assms(2,3) wf_trm_subst_range_iff[of I] by simp
  moreover have "fv (t · I) = {}"
    using assms(1) interpretation_grounds_all'[OF assms(4)]
    by auto
  ultimately show ?thesis unfolding GSMP_def by simp
qed

lemma GSMP_disjoint_subset:
  assumes "GSMP_disjoint L R S" "L' ⊆ L" "R' ⊆ R"
  shows "GSMP_disjoint L' R' S"
using assms(1) SMP_mono[OF assms(2)] SMP_mono[OF assms(3)]
by (auto simp add: GSMP_def GSMP_disjoint_def)

lemma GSMP_disjoint_fst_specific_not_snd_specific:
  assumes "GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec" "l ≠ l'"
  and "proj_specific l m 𝒜 Sec"
  shows "¬proj_specific l' m 𝒜 Sec"
using assms by (fastforce simp add: GSMP_disjoint_def proj_specific_def)

lemma GSMP_disjoint_snd_specific_not_fst_specific:
  assumes "GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec"
  and "proj_specific l' m 𝒜 Sec"
  shows "¬proj_specific l m 𝒜 Sec"
using assms by (auto simp add: GSMP_disjoint_def proj_specific_def)
```

```
lemma GSMP_disjoint_intersection_not_specific:
  assumes "GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec"
  and "t ∈ Sec ∨ {} ⊢c t"
  shows "¬proj_specific l t 𝒜 Sec" "¬proj_specific l t 𝒜 Sec"
using assms by (auto simp add: GSMP_disjoint_def proj_specific_def)
```

5.2.5 Lemmata: Intruder Knowledge and Declassification

```
lemma ik_proj_subst_GSMP_subset:
  assumes I: "wt_subst I" "wf_trms (subst_range I)" "interpretation_subst I"
  shows "ik_st (proj_unl n A) ·set I ⊆ GSMP (trms_proj_lst n A)"
proof
  fix t assume "t ∈ ik_st (proj_unl n A) ·set I"
  hence *: "t ∈ trms_proj_lst n A ·set I" by auto
  then obtain s where "s ∈ trms_proj_lst n A" "t = s · I" by auto
  hence "t ∈ SMP (trms_proj_lst n A)" using SMP_I I(1,2) wf_trm_subst_range_iff[of I] by simp
  moreover have "fv t = {}"
    using * interpretation_grounds_all'[OF I(3)]
    by auto
  ultimately show "t ∈ GSMP (trms_proj_lst n A)" unfolding GSMP_def by simp
qed

lemma declassified_proj_ik_subset: "declassified_lst A I ⊆ ik_st (proj_unl n A) ·set I"
proof (induction A)
  case (Cons a A) thus ?case
    using proj_ik_append[of n "[a]" A] by (auto simp add: declassified_lst_def)
qed (simp add: declassified_lst_def)

lemma declassified_proj_GSMP_subset:
  assumes I: "wt_subst I" "wf_trms (subst_range I)" "interpretation_subst I"
  shows "declassified_lst A I ⊆ GSMP (trms_proj_lst n A)"
by (rule subset_trans[OF declassified_proj_ik_subset ik_proj_subst_GSMP_subset[OF I]])

lemma declassified_subterms_proj_GSMP_subset:
  assumes I: "wt_subst I" "wf_trms (subst_range I)" "interpretation_subst I"
  shows "subterms_set (declassified_lst A I) ⊆ GSMP (trms_proj_lst n A)"
proof
  fix t assume t: "t ∈ subterms_set (declassified_lst A I)"
  then obtain t' where t': "t' ∈ declassified_lst A I" "t ⊑ t'" by moura
  hence "t' ∈ GSMP (trms_proj_lst n A)" using declassified_proj_GSMP_subset[OF assms] by blast
  thus "t ∈ GSMP (trms_proj_lst n A)"
    using SMP.Subterm[of t' "trms_proj_lst n A" t] ground_subterm[OF _ t'(2)] t'(2)
    unfolding GSMP_def by fast
qed

lemma declassified_secrets_subset:
  assumes A: "∀n m. n ≠ m ⟶ GSMP_disjoint (trms_proj_lst n A) (trms_proj_lst m A) Sec"
  and I: "wt_subst I" "wf_trms (subst_range I)" "interpretation_subst I"
  shows "declassified_lst A I ⊆ Sec ∪ {m. {} ⊢c m}"
using declassified_proj_GSMP_subset[OF I] A at_least_2_labels
unfolding GSMP_disjoint_def by blast

lemma declassified_subterms_secrets_subset:
  assumes A: "∀n m. n ≠ m ⟶ GSMP_disjoint (trms_proj_lst n A) (trms_proj_lst m A) Sec"
  and I: "wt_subst I" "wf_trms (subst_range I)" "interpretation_subst I"
  shows "subterms_set (declassified_lst A I) ⊆ Sec ∪ {m. {} ⊢c m}"
using declassified_subterms_proj_GSMP_subset[OF I, of A label_witness1]
      declassified_subterms_proj_GSMP_subset[OF I, of A label_witness2]
      A at_least_2_labels
unfolding GSMP_disjoint_def by fast

lemma declassified_proj_eq: "declassified_lst A I = declassified_lst (proj n A) I"
unfolding declassified_lst_def proj_def by auto
```

```
lemma declassified_append: "declassified_lst (A@B) I = declassified_lst A I ∪ declassified_lst B I"
unfolding declassified_lst_def by auto

lemma declassified_prefix_subset: "prefix A B ⟹ declassified_lst A I ⊆ declassified_lst B I"
using declassified_append unfolding prefix_def by auto
```

5.2.6 Lemmata: Homogeneous and Heterogeneous Terms

```
lemma proj_specific_secrets_anti_mono:
  assumes "proj_specific l t 𝒜 Sec" "Sec' ⊆ Sec"
  shows "proj_specific l t 𝒜 Sec'"
using assms unfolding proj_specific_def by fast

lemma heterogeneous_secrets_anti_mono:
  assumes "heterogeneous_lst t 𝒜 Sec" "Sec' ⊆ Sec"
  shows "heterogeneous_lst t 𝒜 Sec'"
using assms proj_specific_secrets_anti_mono unfolding heterogeneous_lst_def by metis

lemma homogeneous_secrets_mono:
  assumes "homogeneous_lst t 𝒜 Sec'" "Sec' ⊆ Sec"
  shows "homogeneous_lst t 𝒜 Sec"
using assms heterogeneous_secrets_anti_mono by blast

lemma heterogeneous_supterm:
  assumes "heterogeneous_lst t 𝒜 Sec" "t ⊑ t'"
  shows "heterogeneous_lst t' 𝒜 Sec"
proof -
  obtain l1 l2 s1 s2 where *:
      "l1 ≠ l2"
      "s1 ⊑ t" "proj_specific l1 s1 𝒜 Sec"
      "s2 ⊑ t" "proj_specific l2 s2 𝒜 Sec"
    using assms(1) unfolding heterogeneous_lst_def by moura
  thus ?thesis
    using term.order_trans[OF *(2) assms(2)] term.order_trans[OF *(4) assms(2)]
    by (auto simp add: heterogeneous_lst_def)
qed

lemma homogeneous_subterm:
  assumes "homogeneous_lst t 𝒜 Sec" "t' ⊑ t"
  shows "homogeneous_lst t' 𝒜 Sec"
by (metis assms heterogeneous_supterm)

lemma proj_specific_subterm:
  assumes "t ⊑ t'" "proj_specific l t' 𝒜 Sec"
  shows "proj_specific l t 𝒜 Sec ∨ t ∈ Sec ∨ {} ⊢c t"
using GSMP_subterm[OF _ assms(1)] assms(2) by (auto simp add: proj_specific_def)

lemma heterogeneous_term_is_Fun:
  assumes "heterogeneous_lst t A S" shows "∃f T. t = Fun f T"
using assms by (cases t) (auto simp add: GSMP_def heterogeneous_lst_def proj_specific_def)

lemma proj_specific_is_homogeneous:
  assumes 𝒜: "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec"
  and t: "proj_specific l m 𝒜 Sec"
  shows "homogeneous_lst m 𝒜 Sec"
proof
  assume "heterogeneous_lst m 𝒜 Sec"
  then obtain s l' where s: "s ∈ subterms m" "proj_specific l' s 𝒜 Sec" "l ≠ l'"
    unfolding heterogeneous_lst_def by moura
  hence "s ∈ GSMP (trms_proj_lst l 𝒜)" "s ∈ GSMP (trms_proj_lst l' 𝒜)"
    using t by (auto simp add: GSMP_def proj_specific_def)
  hence "s ∈ Sec ∨ {} ⊢c s"
```

```
    using 𝒜 s(3) by (auto simp add: GSMP_disjoint_def)
  thus False using s(2) by (auto simp add: proj_specific_def)
qed

lemma deduct_synth_homogeneous:
  assumes "{} ⊢c t"
  shows "homogeneous_lst t 𝒜 Sec"
proof -
  have "∀s ∈ subterms t. {} ⊢c s" using deduct_synth_subterm[OF assms] by auto
  thus ?thesis unfolding heterogeneous_lst_def proj_specific_def by auto
qed

lemma GSMP_proj_is_homogeneous:
  assumes "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_proj_lst l A) (trms_proj_lst l' A) Sec"
  and "t ∈ GSMP (trms_proj_lst l A)" "t ∉ Sec"
  shows "homogeneous_lst t A Sec"
proof
  assume "heterogeneous_lst t A Sec"
  then obtain s l' where s: "s ∈ subterms t" "proj_specific l' s A Sec" "l ≠ l'"
    unfolding heterogeneous_lst_def by moura
  hence "s ∈ GSMP (trms_proj_lst l A)" "s ∈ GSMP (trms_proj_lst l' A)"
    using assms by (auto simp add: GSMP_def proj_specific_def)
  hence "s ∈ Sec ∨ {} ⊢c s" using assms(1) s(3) by (auto simp add: GSMP_disjoint_def)
  thus False using s(2) by (auto simp add: proj_specific_def)
qed

lemma homogeneous_is_not_proj_specific:
  assumes "homogeneous_lst m 𝒜 Sec"
  shows "∃l::'lbl. ¬proj_specific l m 𝒜 Sec"
proof -
  let ?P = "λl s. proj_specific l s 𝒜 Sec"
  have "∀l1 l2. ∀s1∈subterms m. ∀s2∈subterms m. (l1 ≠ l2 ⟶ (¬?P l1 s1 ∨ ¬?P l2 s2))"
    using assms heterogeneous_lst_def by metis
  then obtain l1 l2 where "l1 ≠ l2" "¬?P l1 m ∨ ¬?P l2 m"
    by (metis term.order_refl at_least_2_labels)
  thus ?thesis by metis
qed

lemma secrets_are_homogeneous:
  assumes "∀s ∈ Sec. P s ⟶ (∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Sec)" "s ∈ Sec" "P s"
  shows "homogeneous_lst s 𝒜 Sec"
using assms by (auto simp add: heterogeneous_lst_def proj_specific_def)

lemma GSMP_is_homogeneous:
  assumes 𝒜: "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec"
  and t: "t ∈ GSMP (trms_lst 𝒜)" "t ∉ Sec"
  shows "homogeneous_lst t 𝒜 Sec"
proof -
  obtain n where n: "t ∈ GSMP (trms_proj_lst n 𝒜)" using in_GSMP_in_proj[OF t(1)] by moura
  show ?thesis using GSMP_proj_is_homogeneous[OF 𝒜 n t(2)] by metis
qed

lemma GSMP_intersection_is_homogeneous:
  assumes 𝒜: "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_proj_lst l 𝒜) (trms_proj_lst l' 𝒜) Sec"
    and t: "t ∈ GSMP (trms_proj_lst l 𝒜) ∩ GSMP (trms_proj_lst l' 𝒜)" "l ≠ l'"
  shows "homogeneous_lst t 𝒜 Sec"
proof -
  define M where "M ≡ GSMP (trms_proj_lst l 𝒜)"
  define M' where "M' ≡ GSMP (trms_proj_lst l' 𝒜)"

  have t_in: "t ∈ M ∩ M'" "t ∈ GSMP (trms_lst 𝒜)"
    using t(1) in_proj_in_GSMP[of t _ 𝒜]
    unfolding M_def M'_def by blast+
```

```
  have "M ∩ M' ⊆ Sec ∪ {m. {} ⊢c m}"
    using 𝒜 GSMP_disjointE[of l 𝒜 l' Sec] t(2)
    unfolding M_def M'_def by presburger
  moreover have "subtermsset (M ∩ M') = M ∩ M'"
    using GSMP_subterms unfolding M_def M'_def by blast
  ultimately have *: "subtermsset (M ∩ M') ⊆ Sec ∪ {m. {} ⊢c m}"
    by blast

  show ?thesis
  proof (cases "t ∈ Sec")
    case True thus ?thesis
      using * secrets_are_homogeneous[of Sec "λt. t ∈ M ∩ M'", OF _ _ t_in(1)]
      by fast
  qed (metis GSMP_is_homogeneous[OF 𝒜 t_in(2)])
qed

lemma GSMP_is_homogeneous':
  assumes 𝒜: "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_projlst l 𝒜) (trms_projlst l' 𝒜) Sec"
  and t: "t ∈ GSMP (trmslst 𝒜)"
         "t ∉ Sec - ⋃{GSMP (trms_projlst l1 𝒜) ∩ GSMP (trms_projlst l2 𝒜) | l1 l2. l1 ≠ l2}"
  shows "homogeneouslst t 𝒜 Sec"
using GSMP_is_homogeneous[OF 𝒜 t(1)] GSMP_intersection_is_homogeneous[OF 𝒜] t(2)
by blast

lemma declassified_secrets_are_homogeneous:
  assumes 𝒜: "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_projlst l 𝒜) (trms_projlst l' 𝒜) Sec"
    and ℐ: "wtsubst ℐ" "wftrms (subst_range ℐ)" "interpretationsubst ℐ"
    and s: "s ∈ declassifiedlst 𝒜 ℐ"
  shows "homogeneouslst s 𝒜 Sec"
proof -
  have s_in: "s ∈ GSMP (trmslst 𝒜)"
    using declassified_proj_GSMP_subset[OF ℐ, of 𝒜 label_witness1]
          in_proj_in_GSMP[of s label_witness1 𝒜] s
    by blast

  show ?thesis
  proof (cases "s ∈ Sec")
    case True thus ?thesis
      using declassified_subterms_secrets_subset[OF 𝒜 ℐ]
            secrets_are_homogeneous[of Sec "λs. s ∈ declassifiedlst 𝒜 ℐ", OF _ _ s]
      by fast
  qed (metis GSMP_is_homogeneous[OF 𝒜 s_in])
qed

lemma Ana_keys_homogeneous:
  assumes 𝒜: "∀l l'. l ≠ l' ⟶ GSMP_disjoint (trms_projlst l 𝒜) (trms_projlst l' 𝒜) Sec"
  and t: "t ∈ GSMP (trmslst 𝒜)"
  and k: "Ana t = (K,T)" "k ∈ set K"
         "k ∉ Sec - ⋃{GSMP (trms_projlst l1 𝒜) ∩ GSMP (trms_projlst l2 𝒜) | l1 l2. l1 ≠ l2}"
  shows "homogeneouslst k 𝒜 Sec"
proof (cases "k ∈ ⋃{GSMP (trms_projlst l1 𝒜) ∩ GSMP (trms_projlst l2 𝒜) | l1 l2. l1 ≠ l2}")
  case False
  hence "k ∉ Sec" using k(3) by fast
  moreover have "k ∈ GSMP (trmslst 𝒜)"
    using t SMP.Ana[OF _ k(1,2)] Ana_keys_fv[OF k(1)] k(2)
    unfolding GSMP_def by auto
  ultimately show ?thesis using GSMP_is_homogeneous[OF 𝒜, of k] by metis
qed (use GSMP_intersection_is_homogeneous[OF 𝒜] in blast)
```

5.2.7 Lemmata: Intruder Deduction Equivalences

```
lemma deduct_if_hom_deduct: "⟨M;A;S⟩ ⊢hom m ⟹ M ⊢ m"
```

```
using deduct_if_restricted_deduct unfolding intruder_deduct_hom_def by blast

lemma hom_deduct_if_hom_ik:
  assumes "⟨M;A;Sec⟩ ⊢hom m" "∀m ∈ M. homogeneouslst m A Sec ∧ m ∈ GSMP (trmslst A)"
  shows "homogeneouslst m A Sec ∧ m ∈ GSMP (trmslst A)"
proof -
  let ?Q = "λm. homogeneouslst m A Sec ∧ m ∈ GSMP (trmslst A)"
  have "?Q t'" when "?Q t" "t' ⊑ t" for t t'
    using homogeneous_subterm[OF _ that(2)] GSMP_subterm[OF _ that(2)] that(1)
    by blast
  thus ?thesis
    using assms(1) restricted_deduct_if_restricted_ik[OF _ assms(2)]
    unfolding intruder_deduct_hom_def
    by blast
qed

lemma deduct_hom_if_synth:
  assumes hom: "homogeneouslst m 𝒜 Sec" "m ∈ GSMP (trmslst 𝒜)"
  and m: "M ⊢c m"
  shows "⟨M; 𝒜; Sec⟩ ⊢hom m"
proof -
  let ?Q = "λm. homogeneouslst m 𝒜 Sec ∧ m ∈ GSMP (trmslst 𝒜)"
  have "?Q t'" when "?Q t" "t' ⊑ t" for t t'
    using homogeneous_subterm[OF _ that(2)] GSMP_subterm[OF _ that(2)] that(1)
    by blast
  thus ?thesis
    using assms deduct_restricted_if_synth[of ?Q]
    unfolding intruder_deduct_hom_def
    by blast
qed

lemma hom_deduct_if_deduct:
  assumes 𝒜: "par_comp 𝒜 Sec"
  and M: "∀m∈M. homogeneouslst m 𝒜 Sec ∧ m ∈ GSMP (trmslst 𝒜)"
  and m: "M ⊢ m" "m ∈ GSMP (trmslst 𝒜)"
shows "⟨M; 𝒜; Sec⟩ ⊢hom m"
proof -
  let ?P = "λx. homogeneouslst x 𝒜 Sec ∧ x ∈ GSMP (trmslst 𝒜)"

  have GSMP_hom: "homogeneouslst t 𝒜 Sec" when "t ∈ GSMP (trmslst 𝒜)" for t
    using 𝒜 GSMP_is_homogeneous[of 𝒜 Sec t]
          secrets_are_homogeneous[of Sec "λx. True" t 𝒜] that
    unfolding par_comp_def by blast

  have P_Ana: "?P k" when "?P t" "Ana t = (K, T)" "k ∈ set K" for t K T k
    using GSMP_Ana_key[OF _ that(2,3), of "trmslst 𝒜"] 𝒜 that GSMP_hom
    by presburger

  have P_subterm: "?P t'" when "?P t" "t' ⊑ t" for t t'
    using GSMP_subterm[of _ "trmslst 𝒜"] homogeneous_subterm[of _ 𝒜 Sec] that
    by blast

  have P_m: "?P m"
    using GSMP_hom[OF m(2)] m(2)
    by metis

  show ?thesis
    using restricted_deduct_if_deduct'[OF M _ _ m(1) P_m] P_Ana P_subterm
    unfolding intruder_deduct_hom_def
    by fast
qed
```

5.2.8 Lemmata: Deduction Reduction of Parallel Composable Constraints

```
lemma par_comp_hom_deduct:
  assumes 𝒜: "par_comp 𝒜 Sec"
  and M: "∀l. ∀m ∈ M l. homogeneous_lst m 𝒜 Sec"
         "∀l. M l ⊆ GSMP (trms_proj_lst l 𝒜)"
         "∀l. Discl ⊆ M l"
         "Discl ⊆ Sec ∪ {m. {} ⊢c m}"
  and Sec: "∀l. ∀s ∈ Sec - Discl. ¬(⟨M l; 𝒜; Sec⟩ ⊢hom s)"
  and t: "⟨⋃l. M l; 𝒜; Sec⟩ ⊢hom t"
  shows "t ∉ Sec - Discl" (is ?A)
        "∀l. t ∈ GSMP (trms_proj_lst l 𝒜) ⟶ ⟨M l; 𝒜; Sec⟩ ⊢hom t" (is ?B)
proof -
  have M': "∀l. ∀m ∈ M l. m ∈ GSMP (trms_lst 𝒜)"
  proof (intro allI ballI)
    fix l m show "m ∈ M l ⟹ m ∈ GSMP (trms_lst 𝒜)" using M(2) in_proj_in_GSMP[of m l 𝒜] by blast
  qed

  show ?A ?B using t
  proof (induction t rule: intruder_deduct_hom_induct)
    case (AxiomH t)
    then obtain lt where t_in_proj_ik: "t ∈ M lt" by moura
    show t_not_Sec: "t ∉ Sec - Discl"
    proof
      assume "t ∈ Sec - Discl"
      hence "∀l. ¬(⟨M l;𝒜;Sec⟩ ⊢hom t)" using Sec by auto
      thus False using intruder_deduct_hom_AxiomH[OF t_in_proj_ik] by metis
    qed

    have 1: "∀l. t ∈ M l ⟶ t ∈ GSMP (trms_proj_lst l 𝒜)"
      using M(2,3) AxiomH by auto

    have 3: "⋀l1 l2. l1 ≠ l2 ⟹ t ∈ GSMP (trms_proj_lst l1 𝒜) ∩ GSMP (trms_proj_lst l2 𝒜)
                                ⟹ {} ⊢c t ∨ t ∈ Discl"
      using 𝒜 t_not_Sec by (auto simp add: par_comp_def GSMP_disjoint_def)

    have 4: "homogeneous_lst t 𝒜 Sec" "t ∈ GSMP (trms_lst 𝒜)" using M(1) M' t_in_proj_ik by auto

    { fix l assume "t ∈ Discl"
      hence "t ∈ M l" using M(3) by auto
      hence "⟨M l; 𝒜; Sec⟩ ⊢hom t" by auto
    } hence 5: "∀l. t ∈ Discl ⟶ ⟨M l; 𝒜; Sec⟩ ⊢hom t" by metis

    show "∀l. t ∈ GSMP (trms_proj_lst l 𝒜) ⟶ ⟨M l; 𝒜; Sec⟩ ⊢hom t"
      by (metis (lifting) Int_iff empty_subsetI
          1 3 4 5 t_in_proj_ik
          intruder_deduct_hom_AxiomH[of t _ 𝒜 Sec]
          deduct_hom_if_synth[of t 𝒜 Sec "{}"]
          ideduct_hom_mono[of "{}" 𝒜 Sec t])
  next
    case (ComposeH T f)
    show "∀l. Fun f T ∈ GSMP (trms_proj_lst l 𝒜) ⟶ ⟨M l; 𝒜; Sec⟩ ⊢hom Fun f T"
    proof (intro allI impI)
      fix l
      assume "Fun f T ∈ GSMP (trms_proj_lst l 𝒜)"
      hence "⋀t. t ∈ set T ⟹ t ∈ GSMP (trms_proj_lst l 𝒜)"
        using GSMP_subterm[OF _ subtermeqI''] by auto
      thus "⟨M l; 𝒜; Sec⟩ ⊢hom Fun f T"
        using ComposeH.IH(2) intruder_deduct_hom_ComposeH[OF ComposeH.hyps(1,2) _ ComposeH.hyps(4,5)]
        by simp
    qed
    thus "Fun f T ∉ Sec - Discl"
      using Sec ComposeH.hyps(5) trms_lst_union[of 𝒜] GSMP_Union[of 𝒜]
```

```
      by (metis (no_types, lifting) UN_iff)
  next
    case (DecomposeH t K T t_i)
    have ti_subt: "t_i ⊑ t" using Ana_subterm[OF DecomposeH.hyps(2)] ‹t_i ∈ set T› by auto
    have t: "homogeneous_lst t 𝒜 Sec" "t ∈ GSMP (trms_lst 𝒜)"
      using DecomposeH.hyps(1) hom_deduct_if_hom_ik M(1) M'
      by auto
    have ti: "homogeneous_lst t_i 𝒜 Sec" "t_i ∈ GSMP (trms_lst 𝒜)"
      using intruder_deduct_hom_DecomposeH[OF DecomposeH.hyps] hom_deduct_if_hom_ik M(1) M' by auto
    { fix l assume *: "t_i ∈ GSMP (trms_proj_lst l 𝒜)" "t ∈ GSMP (trms_proj_lst l 𝒜)"
      hence "⋀k. k ∈ set K ⟹ ⟨M l;𝒜;Sec⟩ ⊢_hom k"
        using GSMP_Ana_key[OF _ DecomposeH.hyps(2)] DecomposeH.IH(4) by auto
      hence "⟨M l;𝒜;Sec⟩ ⊢_hom t_i" "t_i ∉ Sec - Discl"
        using Sec DecomposeH.IH(2) *(2)
              intruder_deduct_hom_DecomposeH[OF _ DecomposeH.hyps(2) _ ‹t_i ∈ set T›]
        by force+
    } moreover {
      fix l1 l2 assume *: "t_i ∈ GSMP (trms_proj_lst l1 𝒜)" "t ∈ GSMP (trms_proj_lst l2 𝒜)" "l1 ≠ l2"
      have "GSMP_disjoint (trms_proj_lst l1 𝒜) (trms_proj_lst l2 𝒜) Sec"
        using *(3) 𝒜 by (simp add: par_comp_def)
      hence "t_i ∈ Sec ∪ {m. {} ⊢_c m}"
        using GSMP_subterm[OF *(2) ti_subt] *(1) by (auto simp add: GSMP_disjoint_def)
      moreover have "⋀k. k ∈ set K ⟹ ⟨M l2;𝒜;Sec⟩ ⊢_hom k"
        using *(2) GSMP_Ana_key[OF _ DecomposeH.hyps(2)] DecomposeH.IH(4) by auto
      ultimately have "t_i ∉ Sec - Discl" "{} ⊢_c t_i ∨ t_i ∈ Discl"
        using Sec DecomposeH.IH(2) *(2)
              intruder_deduct_hom_DecomposeH[OF _ DecomposeH.hyps(2) _ ‹t_i ∈ set T›]
         by (metis (lifting), metis (no_types, lifting) DiffI Un_iff mem_Collect_eq)
      hence "⟨M l1;𝒜;Sec⟩ ⊢_hom t_i" "⟨M l2;𝒜;Sec⟩ ⊢_hom t_i" "t_i ∉ Sec - Discl"
        using M(3,4) deduct_hom_if_synth[THEN ideduct_hom_mono] ti
        by (meson intruder_deduct_hom_AxiomH empty_subsetI subsetCE)+
    } moreover have
        "∃l. t_i ∈ GSMP (trms_proj_lst l 𝒜)"
        "∃l. t ∈ GSMP (trms_proj_lst l 𝒜)"
      using in_GSMP_in_proj[of _ 𝒜] ti(2) t(2) by presburger+
    ultimately show
        "t_i ∉ Sec - Discl"
        "∀l. t_i ∈ GSMP (trms_proj_lst l 𝒜) ⟶ ⟨M l; 𝒜; Sec⟩ ⊢_hom t_i"
      by (metis (no_types, lifting))+
  qed
qed

lemma par_comp_deduct_proj:
  assumes 𝒜: "par_comp 𝒜 Sec"
  and M: "∀l. ∀m∈M l. homogeneous_lst m 𝒜 Sec"
         "∀l. M l ⊆ GSMP (trms_proj_lst l 𝒜)"
         "∀l. Discl ⊆ M l"
  and t: "(⋃l. M l) ⊢ t" "t ∈ GSMP (trms_proj_lst l 𝒜)"
  and Discl: "Discl ⊆ Sec ∪ {m. {} ⊢_c m}"
  shows "M l ⊢ t ∨ (∃s ∈ Sec - Discl. ∃l. M l ⊢ s)"
using t
proof (induction t rule: intruder_deduct_induct)
  case (Axiom t)
  then obtain l' where t_in_ik_proj: "t ∈ M l'" by moura
  show ?case
  proof (cases "t ∈ Sec - Discl ∨ {} ⊢_c t")
    case True
    note T = True
    show ?thesis
    proof (cases "t ∈ Sec - Discl")
      case True thus ?thesis using intruder_deduct.Axiom[OF t_in_ik_proj] by metis
    next
      case False thus ?thesis using T ideduct_mono[of "{}" t] by auto
```

```
      qed
    next
      case False
      hence "t ∉ Sec - Discl" "¬{} ⊢c t" "t ∈ GSMP (trms_projlst l 𝒜)" using Axiom by auto
      hence "(∀l'. l ≠ l' ⟶ t ∉ GSMP (trms_projlst l' 𝒜)) ∨ t ∈ Discl"
        using 𝒜 unfolding GSMP_disjoint_def par_comp_def by auto
      hence "(∀l'. l ≠ l' ⟶ t ∉ GSMP (trms_projlst l' 𝒜)) ∨ t ∈ M l ∨ {} ⊢c t" using M by auto
      thus ?thesis using Axiom deduct_if_synth[THEN ideduct_mono] t_in_ik_proj
        by (metis (no_types, lifting) False M(2) intruder_deduct.Axiom subsetCE)
    qed
  next
    case (Compose T f)
    hence "Fun f T ∈ GSMP (trms_projlst l 𝒜)" using Compose.prems by auto
    hence "⋀t. t ∈ set T ⟹ t ∈ GSMP (trms_projlst l 𝒜)" unfolding GSMP_def by auto
    hence IH: "⋀t. t ∈ set T ⟹ M l ⊢ t ∨ (∃s ∈ Sec - Discl. ∃l. M l ⊢ s)"
      using Compose.IH by auto
    show ?case
    proof (cases "∀t ∈ set T. M l ⊢ t")
      case True thus ?thesis by (metis intruder_deduct.Compose[OF Compose.hyps(1,2)])
    qed (metis IH)
  next
    case (Decompose t K T ti)
    have hom_ik: "∀l. ∀m∈M l. homogeneouslst m 𝒜 Sec ∧ m ∈ GSMP (trmslst 𝒜)"
    proof (intro allI ballI conjI)
      fix l m assume m: "m ∈ M l"
      thus "homogeneouslst m 𝒜 Sec" using M(1) by simp
      show "m ∈ GSMP (trmslst 𝒜)" using in_proj_in_GSMP[of m l 𝒜] M(2) m by blast
    qed

    have par_comp_unfold:
        "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (trms_projlst l1 𝒜) (trms_projlst l2 𝒜) Sec"
      using 𝒜 by (auto simp add: par_comp_def)

    note ti_GSMP = in_proj_in_GSMP[OF Decompose.prems(1)]

    have "⟨⋃l. M l; 𝒜; Sec⟩ ⊢hom ti"
      using intruder_deduct.Decompose[OF Decompose.hyps]
            hom_deduct_if_deduct[OF 𝒜, of "⋃l. M l"] hom_ik ti_GSMP
      by blast
    hence "(⟨M l; 𝒜; Sec⟩ ⊢hom ti) ∨ (∃s ∈ Sec-Discl. ∃l. ⟨M l;𝒜;Sec⟩ ⊢hom s)"
      using par_comp_hom_deduct(2)[OF 𝒜 M Discl(1)] Decompose.prems(1)
      by blast
    thus ?case using deduct_if_hom_deduct[of _ 𝒜 Sec] by auto
  qed
```

5.2.9 Theorem: Parallel Compositionality for Labeled Constraints

```
lemma par_comp_prefix: assumes "par_comp (A@B) M" shows "par_comp A M"
proof -
  let ?L = "λl. trms_projlst l A ∪ trms_projlst l B"
  have "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (?L l1) (?L l2) M"
    using assms unfolding par_comp_def
    by (metis trmsst_append proj_append(2) unlabel_append)
  hence "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (trms_projlst l1 A) (trms_projlst l2 A) M"
    using SMP_union by (auto simp add: GSMP_def GSMP_disjoint_def)
  thus ?thesis using assms unfolding par_comp_def by blast
qed

theorem par_comp_constr_typed:
  assumes 𝒜: "par_comp 𝒜 Sec"
  and ℐ: "ℐ ⊨ ⟨unlabel 𝒜⟩" "interpretationsubst ℐ" "wtsubst ℐ" "wftrms (subst_range ℐ)"
  shows "(∀l. (ℐ ⊨ ⟨proj_unl l 𝒜⟩)) ∨ (∃𝒜'. prefix 𝒜' 𝒜 ∧ (strand_leakslst 𝒜' Sec ℐ))"
proof -
```

```
let ?L = "λ𝒜'. ∃t ∈ Sec - declassified_lst 𝒜' ℐ. ∃l. ⟦{}; proj_unl l 𝒜'@[Send t]⟧_d ℐ"
have "⟦{}; unlabel 𝒜⟧_d ℐ" using ℐ by (simp add: constr_sem_d_def)
with 𝒜 have "(∀l. ⟦{}; proj_unl l 𝒜⟧_d ℐ) ∨ (∃𝒜'. prefix 𝒜' 𝒜 ∧ ?L 𝒜')"
proof (induction "unlabel 𝒜" arbitrary: 𝒜 rule: List.rev_induct)
  case Nil
  hence "𝒜 = []" using unlabel_nil_only_if_nil by simp
  thus ?case by auto
next
  case (snoc b B 𝒜)
  hence disj: "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (trms_proj_lst l1 𝒜) (trms_proj_lst l2 𝒜) Sec"
    by (auto simp add: par_comp_def)

  obtain a A n where a: "𝒜 = A@[a]" "a = (ln n, b) ∨ a = (⋆, b)"
    using unlabel_snoc_inv[OF snoc.hyps(2)[symmetric]] by moura
  hence A: "𝒜 = A@[(ln n, b)] ∨ 𝒜 = A@[(⋆, b)]" by metis

  have 1: "B = unlabel A" using a snoc.hyps(2) unlabel_append[of A "[a]"] by auto
  have 2: "par_comp A Sec" using par_comp_prefix snoc.prems(1) a by metis
  have 3: "⟦{}; unlabel A⟧_d ℐ" by (metis 1 snoc.prems(2) snoc.hyps(2) strand_sem_split(3))
  have IH: "(∀l. ⟦{}; proj_unl l A⟧_d ℐ) ∨ (∃𝒜'. prefix 𝒜' A ∧ ?L 𝒜')"
    by (rule snoc.hyps(1)[OF 1 2 3])

  show ?case
  proof (cases "∀l. ⟦{}; proj_unl l A⟧_d ℐ")
    case False
    then obtain 𝒜' where 𝒜': "prefix 𝒜' A" "?L 𝒜'" by (metis IH)
    hence "prefix 𝒜' (A@[a])" using a prefix_prefix[of _ A "[a]"] by simp
    thus ?thesis using 𝒜'(2) a by auto
  next
    case True
    note IH' = True
    show ?thesis
    proof (cases b)
      case (Send t)
      hence "ik_st (unlabel A) ·_set ℐ ⊢ t · ℐ"
        using a ‹⟦{}; unlabel 𝒜⟧_d ℐ› strand_sem_split(2)[of "{}" "unlabel A" "unlabel [a]" ℐ]
              unlabel_append[of A "[a]"]
        by auto
      hence *: "(⋃l. (ik_st (proj_unl l A) ·_set ℐ)) ⊢ t · ℐ"
        using proj_ik_union_is_unlabel_ik image_UN by metis

      have "ik_st (proj_unl l 𝒜) = ik_st (proj_unl l A)" for l
        using Send A
        by (metis append_Nil2 ik_st.simps(3) proj_unl_cons(3) proj_nil(2)
                  singleton_lst_proj(1,2) proj_ik_append)
      hence **: "ik_st (proj_unl l A) ·_set ℐ ⊆ GSMP (trms_proj_lst l 𝒜)" for l
        using ik_proj_subst_GSMP_subset[OF ℐ(3,4,2), of _ 𝒜]
        by auto

      note Discl =
        declassified_proj_ik_subset[of A ℐ]
        declassified_proj_GSMP_subset[OF ℐ(3,4,2), of A]
        declassified_secrets_subset[OF disj ℐ(3,4,2)]
        declassified_append[of A "[a]" ℐ]

      have Sec: "ground Sec"
        using 𝒜 by (auto simp add: par_comp_def)

      have "∀m∈ik_st (proj_unl l 𝒜) ·_set ℐ. homogeneous_lst m 𝒜 Sec ∨ m ∈ Sec-declassified_lst A ℐ"
           "∀m∈ik_st (proj_unl l 𝒜) ·_set ℐ. m ∈ GSMP (trms_lst 𝒜)"
           "ik_st (proj_unl l 𝒜) ·_set ℐ ⊆ GSMP (trms_proj_lst l 𝒜)"
        for l
        using declassified_secrets_are_homogeneous[OF disj ℐ(3,4,2)]
```

```
        GSMP_proj_is_homogeneous[OF disj]
        ik_proj_subst_GSMP_subset[OF 𝓘(3,4,2), of _ 𝓐]
  apply (metis (no_types, lifting) Diff_iff Discl(4) UnCI a(1) subsetCE)
  using ik_proj_subst_GSMP_subset[OF 𝓘(3,4,2), of _ 𝓐]
        GSMP_Union[of 𝓐]
  by auto
moreover have "ik_st (proj_unl l [a]) = {}" for l
  using Send proj_ik_st_is_proj_rcv_set[of _ "[a]"] a(2) by auto
ultimately have M:
    "∀l. ∀m∈ik_st (proj_unl l A) ·set 𝓘. homogeneous_lst m 𝓐 Sec ∨ m ∈ Sec-declassified_lst A
𝓘"
    "∀l. ik_st (proj_unl l A) ·set 𝓘 ⊆ GSMP (trms_proj_lst l 𝓐)"
  using a(1) proj_ik_append[of _ A "[a]"] by auto

have prefix_A: "prefix A 𝓐" using A by auto

have "s · 𝓘 = s"
  when "s ∈ Sec" for s
  using that Sec by auto
hence leakage_case: "⟦{}; proj_unl l A@[Send s]⟧_d 𝓘"
  when "s ∈ Sec - declassified_lst A 𝓘" "ik_st (proj_unl l A) ·set 𝓘 ⊢ s" for l s
  using that strand_sem_append(2) IH' by auto

have proj_deduct_case_n:
    "∀m. m ≠ n ⟶ ⟦{}; proj_unl m (A@[a])⟧_d 𝓘"
    "ik_st (proj_unl n A) ·set 𝓘 ⊢ t · 𝓘 ⟹ ⟦{}; proj_unl n (A@[a])⟧_d 𝓘"
  when "a = (ln n, Send t)"
  using that IH' proj_append(2)[of _ A]
  by auto

have proj_deduct_case_star:
    "⟦{}; proj_unl l (A@[a])⟧_d 𝓘"
  when "a = (⋆, Send t)" "ik_st (proj_unl l A) ·set 𝓘 ⊢ t · 𝓘" for l
  using that IH' proj_append(2)[of _ A]
  by auto

show ?thesis
proof (cases "∃l. ∃m ∈ ik_st (proj_unl l A) ·set 𝓘. m ∈ Sec - declassified_lst A 𝓘")
  case True
  then obtain l s where ls: "s ∈ Sec - declassified_lst A 𝓘" "ik_st (proj_unl l A) ·set 𝓘 ⊢ s"
    using intruder_deduct.Axiom by metis
  thus ?thesis using leakage_case prefix_A by blast
next
  case False
  hence M': "∀l. ∀m∈ik_st (proj_unl l A) ·set 𝓘. homogeneous_lst m 𝓐 Sec" using M(1) by blast

  note deduct_proj_lemma =
      par_comp_deduct_proj[OF snoc.prems(1) M' M(2) _ *, of "declassified_lst A 𝓘" n]

  from a(2) show ?thesis
  proof
    assume "a = (ln n, b)"
    hence "a = (ln n, Send t)" "t · 𝓘 ∈ GSMP (trms_proj_lst n 𝓐)"
      using Send a(1) trms_proj_lst_append[of n A "[a]"]
            GSMP_wt_substI[OF _ 𝓘(3,4,2)]
      by (metis, force)
    hence
        "a = (ln n, Send t)"
        "∀m. m ≠ n ⟶ ⟦{}; proj_unl m (A@[a])⟧_d 𝓘"
        "ik_st (proj_unl n A) ·set 𝓘 ⊢ t · 𝓘 ⟹ ⟦{}; proj_unl n (A@[a])⟧_d 𝓘"
        "t · 𝓘 ∈ GSMP (trms_proj_lst n 𝓐)"
      using proj_deduct_case_n
      by auto
```

```
          hence "(∀l. ⟦{}; proj_unl l 𝒜⟧_d ℐ) ∨
                (∃s ∈ Sec-declassified_lst A ℐ. ∃l. ik_st (proj_unl l A) ·_set ℐ ⊢ s)"
            using deduct_proj_lemma A a Discl
            by fast
          thus ?thesis using leakage_case prefix_A by metis
        next
          assume "a = (⋆, b)"
          hence ***: "a = (⋆, Send t)" "t · ℐ ∈ GSMP (trms_proj_lst l 𝒜)" for l
            using Send a(1) GSMP_wt_substI[OF _ ℐ(3,4,2)]
            by (metis, force)
          hence "t · ℐ ∈ Sec - declassified_lst A ℐ ∨
                t · ℐ ∈ declassified_lst A ℐ ∨
                t · ℐ ∈ {m. {} ⊢_c m}"
            using snoc.prems(1) a(1) at_least_2_labels
            unfolding par_comp_def GSMP_disjoint_def
            by blast
          thus ?thesis
          proof (elim disjE)
            assume "t · ℐ ∈ Sec - declassified_lst A ℐ"
            hence "∃s ∈ Sec - declassified_lst A ℐ. ∃l. ik_st (proj_unl l A) ·_set ℐ ⊢ s"
              using deduct_proj_lemma ***(2) A a Discl
              by blast
            thus ?thesis using prefix_A leakage_case by blast
          next
            assume "t · ℐ ∈ declassified_lst A ℐ"
            hence "ik_st (proj_unl l A) ·_set ℐ ⊢ t · ℐ" for l
              using intruder_deduct.Axiom Discl(1) by blast
            thus ?thesis using proj_deduct_case_star[OF ***(1)] a(1) by fast
          next
            assume "t · ℐ ∈ {m. {} ⊢_c m}"
            hence "M ⊢ t · ℐ" for M using ideduct_mono[OF deduct_if_synth] by blast
            thus ?thesis using IH' a(1) ***(1) by fastforce
          qed
        qed
      qed
    next
      case (Receive t)
      hence "⟦{}; proj_unl l 𝒜⟧_d ℐ" for l
        using IH' a proj_append(2)[of l A "[a]"]
        unfolding unlabel_def proj_def by auto
      thus ?thesis by metis
    next
      case (Equality ac t t')
      hence *: "⟦M; [Equality ac t t']⟧_d ℐ" for M
        using a ‹⟦{}; unlabel 𝒜⟧_d ℐ› unlabel_append[of A "[a]"]
        by auto
      show ?thesis
        using a proj_append(2)[of _ A "[a]"] Equality
              strand_sem_append(2)[OF _ *] IH'
        unfolding unlabel_def proj_def by auto
    next
      case (Inequality X F)
      hence *: "⟦M; [Inequality X F]⟧_d ℐ" for M
        using a ‹⟦{}; unlabel 𝒜⟧_d ℐ› unlabel_append[of A "[a]"]
        by auto
      show ?thesis
        using a proj_append(2)[of _ A "[a]"] Inequality
              strand_sem_append(2)[OF _ *] IH'
        unfolding unlabel_def proj_def by auto
    qed
  qed
qed
thus ?thesis using ℐ(1) unfolding strand_leaks_lst_def by (simp add: constr_sem_d_def)
```

```
qed

theorem par_comp_constr:
  assumes 𝒜: "par_comp 𝒜 Sec" "typing_cond (unlabel 𝒜)"
  and ℐ: "ℐ ⊨ ⟨unlabel 𝒜⟩" "interpretation_subst ℐ"
  shows "∃ℐτ. interpretation_subst ℐτ ∧ wt_subst ℐτ ∧ wf_trms (subst_range ℐτ) ∧ (ℐτ ⊨ ⟨unlabel 𝒜⟩) ∧
            ((∀l. (ℐτ ⊨ ⟨proj_unl l 𝒜⟩)) ∨ (∃𝒜'. prefix 𝒜' 𝒜 ∧ (strand_leaks_lst 𝒜' Sec ℐτ)))"
proof -
  from 𝒜(2) have *:
      "wf_st {} (unlabel 𝒜)"
      "fv_st (unlabel 𝒜) ∩ bvars_st (unlabel 𝒜) = {}"
      "tfr_st (unlabel 𝒜)"
      "wf_trms (trms_st (unlabel 𝒜))"
      "Ana_invar_subst (ik_st (unlabel 𝒜) ∪ assignment_rhs_st (unlabel 𝒜))"
    unfolding typing_cond_def tfr_st_def by metis+

  obtain ℐτ where ℐτ: "ℐτ ⊨ ⟨unlabel 𝒜⟩" "interpretation_subst ℐτ" "wt_subst ℐτ" "wf_trms (subst_range
ℐτ)"
    using wt_attack_if_tfr_attack_d[OF * ℐ(2,1)] by metis

  show ?thesis using par_comp_constr_typed[OF 𝒜(1) ℐτ] ℐτ by auto
qed
```

5.2.10 Theorem: Parallel Compositionality for Labeled Protocols

Definitions: Labeled Protocols

We state our result on the level of protocol traces (i.e., the constraints reachable in a symbolic execution of the actual protocol). Hence, we do not need to convert protocol strands to intruder constraints in the following well-formedness definitions.

```
definition wf_lsts::"('fun,'var,'lbl) labeled_strand set ⇒ bool" where
  "wf_lsts 𝒮 ≡ (∀𝒜 ∈ 𝒮. wf_lst {} 𝒜) ∧ (∀𝒜 ∈ 𝒮. ∀𝒜' ∈ 𝒮. fv_lst 𝒜 ∩ bvars_lst 𝒜' = {})"

definition wf_lsts'::"('fun,'var,'lbl) labeled_strand set ⇒ ('fun,'var,'lbl) labeled_strand ⇒ bool"
where
  "wf_lsts' 𝒮 𝒜 ≡ (∀𝒜' ∈ 𝒮. wf_st (wfrestrictedvars_lst 𝒜) (unlabel 𝒜')) ∧
                 (∀𝒜' ∈ 𝒮. ∀𝒜'' ∈ 𝒮. fv_lst 𝒜' ∩ bvars_lst 𝒜'' = {}) ∧
                 (∀𝒜' ∈ 𝒮. fv_lst 𝒜' ∩ bvars_lst 𝒜 = {}) ∧
                 (∀𝒜' ∈ 𝒮. fv_lst 𝒜 ∩ bvars_lst 𝒜' = {})"

definition typing_cond_prot where
  "typing_cond_prot 𝒫 ≡
    wf_lsts 𝒫 ∧
    tfr_set (⋃(trms_lst ` 𝒫)) ∧
    wf_trms (⋃(trms_lst ` 𝒫)) ∧
    (∀𝒜 ∈ 𝒫. list_all tfr_stp (unlabel 𝒜)) ∧
    Ana_invar_subst (⋃(ik_st ` unlabel ` 𝒫) ∪ ⋃(assignment_rhs_st ` unlabel ` 𝒫))"

definition par_comp_prot where
  "par_comp_prot 𝒫 Sec ≡
    (∀l1 l2. l1 ≠ l2 ⟶
      GSMP_disjoint (⋃𝒜 ∈ 𝒫. trms_proj_lst l1 𝒜) (⋃𝒜 ∈ 𝒫. trms_proj_lst l2 𝒜) Sec) ∧
    ground Sec ∧ (∀s ∈ Sec. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Sec) ∧
    typing_cond_prot 𝒫"
```

Lemmata: Labeled Protocols

```
lemma wf_lsts_eqs_wf_lsts'[simp]: "wf_lsts S = wf_lsts' S []"
unfolding wf_lsts_def wf_lsts'_def unlabel_def by auto

lemma par_comp_prot_impl_par_comp:
  assumes "par_comp_prot 𝒫 Sec" "𝒜 ∈ 𝒫"
  shows "par_comp 𝒜 Sec"
```

```
proof -
  have *: "∀l1 l2. l1 ≠ l2 ⟶
           GSMP_disjoint (⋃𝒜 ∈ 𝒫. trms_proj_lst l1 𝒜) (⋃𝒜 ∈ 𝒫. trms_proj_lst l2 𝒜) Sec"
    using assms(1) unfolding par_comp_prot_def by metis
  { fix l1 l2::'lbl assume **: "l1 ≠ l2"
    hence ***: "GSMP_disjoint (⋃𝒜 ∈ 𝒫. trms_proj_lst l1 𝒜) (⋃𝒜 ∈ 𝒫. trms_proj_lst l2 𝒜) Sec"
      using * by auto
    have "GSMP_disjoint (trms_proj_lst l1 𝒜) (trms_proj_lst l2 𝒜) Sec"
      using GSMP_disjoint_subset[OF ***] assms(2) by auto
  } hence "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (trms_proj_lst l1 𝒜) (trms_proj_lst l2 𝒜) Sec" by metis
  thus ?thesis using assms unfolding par_comp_prot_def par_comp_def by metis
qed

lemma typing_cond_prot_impl_typing_cond:
  assumes "typing_cond_prot 𝒫" "𝒜 ∈ 𝒫"
  shows "typing_cond (unlabel 𝒜)"
proof -
  have 1: "wf_st {} (unlabel 𝒜)" "fv_lst 𝒜 ∩ bvars_lst 𝒜 = {}"
    using assms unfolding typing_cond_prot_def wf_lsts_def by auto

  have "tfr_set (⋃(trms_lst ` 𝒫))"
       "wf_trms (⋃(trms_lst ` 𝒫))"
       "trms_lst 𝒜 ⊆ ⋃(trms_lst ` 𝒫)"
       "SMP (trms_lst 𝒜) - Var`𝒱 ⊆ SMP (⋃(trms_lst ` 𝒫)) - Var`𝒱"
    using assms SMP_mono[of "trms_lst 𝒜" "⋃(trms_lst ` 𝒫)"]
    unfolding typing_cond_prot_def
    by (metis, metis, auto)
  hence 2: "tfr_set (trms_lst 𝒜)" and 3: "wf_trms (trms_lst 𝒜)"
    unfolding tfr_set_def by (meson subsetD)+

  have 4: "list_all tfr_stp (unlabel 𝒜)" using assms unfolding typing_cond_prot_def by auto

  have "subterms_set (ik_st (unlabel 𝒜) ∪ assignment_rhs_st (unlabel 𝒜)) ⊆
        subterms_set (⋃(ik_st ` unlabel ` 𝒫) ∪ ⋃(assignment_rhs_st ` unlabel ` 𝒫))"
    using assms(2) by auto
  hence 5: "Ana_invar_subst (ik_st (unlabel 𝒜) ∪ assignment_rhs_st (unlabel 𝒜))"
    using assms SMP_mono unfolding typing_cond_prot_def Ana_invar_subst_def by (meson subsetD)

  show ?thesis using 1 2 3 4 5 unfolding typing_cond_def tfr_st_def by blast
qed
```

Theorem: Parallel Compositionality for Labeled Protocols

```
definition component_prot where
  "component_prot n P ≡ (∀l ∈ P. ∀s ∈ set l. is_LabelN n s ∨ is_LabelS s)"

definition composed_prot where
  "composed_prot 𝒫_i ≡ {𝒜. ∀n. proj n 𝒜 ∈ 𝒫_i n}"

definition component_secure_prot where
  "component_secure_prot n P Sec attack ≡ (∀𝒜 ∈ P. suffix [(ln n, Send (Fun attack []))] 𝒜 ⟶
     (∀ℐ_τ. (interpretation_subst ℐ_τ ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ)) ⟶
           ¬(ℐ_τ ⊨ ⟨proj_unl n 𝒜⟩) ∧
           (∀𝒜'. prefix 𝒜' 𝒜 ⟶
                 (∀t ∈ Sec-declassified_lst 𝒜' ℐ_τ. ¬(ℐ_τ ⊨ ⟨proj_unl n 𝒜'@[Send t]⟩)))))"

definition component_leaks where
  "component_leaks n 𝒜 Sec ≡ (∃𝒜' ℐ_τ. interpretation_subst ℐ_τ ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ)
∧
     prefix 𝒜' 𝒜 ∧ (∃t ∈ Sec - declassified_lst 𝒜' ℐ_τ. (ℐ_τ ⊨ ⟨proj_unl n 𝒜'@[Send t]⟩)))"

definition unsat where
  "unsat 𝒜 ≡ (∀ℐ. interpretation_subst ℐ ⟶ ¬(ℐ ⊨ ⟨unlabel 𝒜⟩))"
```

```
theorem par_comp_constr_prot:
  assumes P: "P = composed_prot Pi" "par_comp_prot P Sec" "∀n. component_prot n (Pi n)"
  and left_secure: "component_secure_prot n (Pi n) Sec attack"
  shows "∀𝒜 ∈ P. suffix [(ln n, Send (Fun attack []))] 𝒜 ⟶
                unsat 𝒜 ∨ (∃m. n ≠ m ∧ component_leaks m 𝒜 Sec)"
proof -
  { fix 𝒜 𝒜' assume 𝒜: "𝒜 = 𝒜'@[(ln n, Send (Fun attack []))]" "𝒜 ∈ P"
    let ?P = "∃𝒜' ℐτ. interpretation_subst ℐτ ∧ wt_subst ℐτ ∧ wf_trms (subst_range ℐτ) ∧ prefix 𝒜' 𝒜
∧
                (∃t ∈ Sec - declassified_lst 𝒜' ℐτ. ∃m. n ≠ m ∧ (ℐτ ⊨ ⟨proj_unl m 𝒜'@[Send
t]⟩))"
    have tcp: "typing_cond_prot P" using P(2) unfolding par_comp_prot_def by simp
    have par_comp: "par_comp 𝒜 Sec" "typing_cond (unlabel 𝒜)"
      using par_comp_prot_impl_par_comp[OF P(2) 𝒜(2)]
            typing_cond_prot_impl_typing_cond[OF tcp 𝒜(2)]
      by metis+

    have "unlabel (proj n 𝒜) = proj_unl n 𝒜" "proj_unl n 𝒜 = proj_unl n (proj n 𝒜)"
        "⋀A. A ∈ Pi n ⟹ proj n A = A"
        "proj n 𝒜 = (proj n 𝒜')@[(ln n, Send (Fun attack []))]"
      using P(1,3) 𝒜 by (auto simp add: proj_def unlabel_def component_prot_def composed_prot_def)
    moreover have "proj n 𝒜 ∈ Pi n"
      using P(1) 𝒜 unfolding composed_prot_def by blast
    moreover {
      fix A assume "prefix A 𝒜"
      hence *: "prefix (proj n A) (proj n 𝒜)" unfolding proj_def prefix_def by force
      hence "proj_unl n A = proj_unl n (proj n A)"
           "∀I. declassified_lst A I = declassified_lst (proj n A) I"
        unfolding proj_def declassified_lst_def by auto
      hence "∃B. prefix B (proj n 𝒜) ∧ proj_unl n A = proj_unl n B ∧
               (∀I. declassified_lst A I = declassified_lst B I)"
        using * by metis

    }
    ultimately have *:
        "∀ℐτ. interpretation_subst ℐτ ∧ wt_subst ℐτ ∧ wf_trms (subst_range ℐτ) ⟶
              ¬(ℐτ ⊨ ⟨proj_unl n 𝒜⟩) ∧ (∀𝒜'. prefix 𝒜' 𝒜 ⟶
                    (∀t ∈ Sec - declassified_lst 𝒜' ℐτ. ¬(ℐτ ⊨ ⟨proj_unl n 𝒜'@[Send t]⟩)))"
      using left_secure unfolding component_secure_prot_def composed_prot_def suffix_def by metis
    { fix ℐ assume ℐ: "interpretation_subst ℐ" "ℐ ⊨ ⟨unlabel 𝒜⟩"
      obtain ℐτ where ℐτ:
          "interpretation_subst ℐτ" "wt_subst ℐτ" "wf_trms (subst_range ℐτ)"
          "∃𝒜'. prefix 𝒜' 𝒜 ∧ (strand_leaks_lst 𝒜' Sec ℐτ)"
        using par_comp_constr[OF par_comp ℐ(2,1)] * by moura
      hence "∃𝒜'. prefix 𝒜' 𝒜 ∧ (∃t ∈ Sec - declassified_lst 𝒜' ℐτ. ∃m.
               n ≠ m ∧ (ℐτ ⊨ ⟨proj_unl m 𝒜'@[Send t]⟩))"
        using ℐτ(4) * unfolding strand_leaks_lst_def by metis
      hence ?P using ℐτ(1,2,3) by auto
    } hence "unsat 𝒜 ∨ (∃m. n ≠ m ∧ component_leaks m 𝒜 Sec)"
      by (metis unsat_def component_leaks_def)
  } thus ?thesis unfolding suffix_def by metis
qed

end
```

5.2.11 Automated GSMP Disjointness

```
locale labeled_typed_model' = typed_model' arity public Ana Γ +
  labeled_typed_model arity public Ana Γ label_witness1 label_witness2
  for arity::"'fun ⇒ nat"
    and public::"'fun ⇒ bool"
    and Ana::"('fun,(('fun,'atom::finite) term_type × nat)) term
```

```
          ⇒ (('fun,(('fun,'atom) term_type × nat)) term list
             × ('fun,(('fun,'atom) term_type × nat)) term list)"
    and Γ::"('fun,(('fun,'atom) term_type × nat)) term ⇒ ('fun,'atom) term_type"
    and label_witness1 label_witness2::'lbl
begin

lemma GSMP_disjointI:
  fixes A' A B B'::"('fun, ('fun, 'atom) term × nat) term list"
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "δ ≡ var_rename (max_var_set (fv_set (set A)))"
  assumes A'_wf: "list_all (wf_trm' arity) A'"
    and B'_wf: "list_all (wf_trm' arity) B'"
    and A_inst: "has_all_wt_instances_of Γ (set A') (set A)"
    and B_inst: "has_all_wt_instances_of Γ (set B') (set (B ·_list δ))"
    and A_SMP_repr: "finite_SMP_representation arity Ana Γ A"
    and B_SMP_repr: "finite_SMP_representation arity Ana Γ (B ·_list δ)"
    and AB_trms_disj:
      "∀t ∈ set A. ∀s ∈ set (B ·_list δ). Γ t = Γ s ∧ mgu t s ≠ None ⟶
        (intruder_synth' public arity {} t ∧ intruder_synth' public arity {} s) ∨
        ((∃u ∈ Sec. is_wt_instance_of_cond Γ t u) ∧ (∃u ∈ Sec. is_wt_instance_of_cond Γ s u))"
    and Sec_wf: "wf_trms Sec"
  shows "GSMP_disjoint (set A') (set B') ((f Sec) - {m. {} ⊢_c m})"
proof -
  have A_wf: "wf_trms (set A)" and B_wf: "wf_trms (set (B ·_list δ))"
    and A'_wf': "wf_trms (set A')" and B'_wf': "wf_trms (set B')"
    using finite_SMP_representationD[OF A_SMP_repr]
          finite_SMP_representationD[OF B_SMP_repr]
          A'_wf B'_wf
    unfolding wf_trms_code[symmetric] wf_trm_code[symmetric] list_all_iff by blast+

  have AB_fv_disj: "fv_set (set A) ∩ fv_set (set (B ·_list δ)) = {}"
    using var_rename_fv_set_disjoint'[of "set A" "set B", unfolded δ_def[symmetric]] by simp

  have "GSMP_disjoint (set A) (set (B ·_list δ)) ((f Sec) - {m. {} ⊢_c m})"
    using ground_SMP_disjointI[OF AB_fv_disj A_SMP_repr B_SMP_repr Sec_wf AB_trms_disj]
    unfolding GSMP_def GSMP_disjoint_def f_def by blast
  moreover have "SMP (set A') ⊆ SMP (set A)" "SMP (set B') ⊆ SMP (set (B ·_list δ))"
    using SMP_I'[OF A'_wf' A_wf A_inst] SMP_SMP_subset[of "set A'" "set A"]
          SMP_I'[OF B'_wf' B_wf B_inst] SMP_SMP_subset[of "set B'" "set (B ·_list δ)"]
    by blast+
  ultimately show ?thesis unfolding GSMP_def GSMP_disjoint_def by auto
qed

end

end
```

6 The Stateful Protocol Composition Result

In this chapter, we extend the compositionality result to stateful security protocols. This work is an extension of the work described in [4] and [1, chapter 5].

6.1 Labeled Stateful Strands (Labeled_Stateful_Strands)

```
theory Labeled_Stateful_Strands
imports Stateful_Strands Labeled_Strands
begin
```

6.1.1 Definitions

Syntax for stateful strand labels

```
abbreviation Star_step ("⟨⋆, _⟩") where
  "⟨⋆, (s::('a,'b) stateful_strand_step)⟩ ≡ (⋆, s)"

abbreviation LabelN_step ("⟨_, _⟩") where
  "⟨(l::'a), (s::('b,'c) stateful_strand_step)⟩ ≡ (ln l, s)"
```

Database projection

```
abbreviation dbproj where "dbproj l D ≡ filter (λd. fst d = l) D"
```

The type of labeled stateful strands

```
type_synonym ('a,'b,'c) labeled_stateful_strand_step = "'c strand_label × ('a,'b)
stateful_strand_step"
type_synonym ('a,'b,'c) labeled_stateful_strand = "('a,'b,'c) labeled_stateful_strand_step list"
```

Dual strands

```
fun dual_lsstp::"('a,'b,'c) labeled_stateful_strand_step ⇒ ('a,'b,'c) labeled_stateful_strand_step"
where
  "dual_lsstp (l,send⟨t⟩) = (l,receive⟨t⟩)"
| "dual_lsstp (l,receive⟨t⟩) = (l,send⟨t⟩)"
| "dual_lsstp x = x"

definition dual_lsst::"('a,'b,'c) labeled_stateful_strand ⇒ ('a,'b,'c) labeled_stateful_strand"
where
  "dual_lsst ≡ map dual_lsstp"
```

Substitution application

```
fun subst_apply_labeled_stateful_strand_step::
  "('a,'b,'c) labeled_stateful_strand_step ⇒ ('a,'b) subst ⇒
   ('a,'b,'c) labeled_stateful_strand_step"
  (infix "·_lsstp" 51) where
  "(l,s) ·_lsstp ϑ  = (l,s ·_sstp ϑ)"

definition subst_apply_labeled_stateful_strand::
  "('a,'b,'c) labeled_stateful_strand ⇒ ('a,'b) subst ⇒ ('a,'b,'c) labeled_stateful_strand"
  (infix "·_lsst" 51) where
  "S ·_lsst ϑ ≡ map (λx. x ·_lsstp ϑ) S"
```

Definitions lifted from stateful strands

```
abbreviation wfrestrictedvars_lsst where "wfrestrictedvars_lsst S ≡ wfrestrictedvars_sst (unlabel S)"

abbreviation ik_lsst where "ik_lsst S ≡ ik_sst (unlabel S)"
```

```
abbreviation dblsst where "dblsst S ≡ dbsst (unlabel S)"
abbreviation db'lsst where "db'lsst S ≡ db'sst (unlabel S)"

abbreviation trmslsst where "trmslsst S ≡ trmssst (unlabel S)"
abbreviation trms_projlsst where "trms_projlsst n S ≡ trmssst (proj_unl n S)"

abbreviation varslsst where "varslsst S ≡ varssst (unlabel S)"
abbreviation vars_projlsst where "vars_projlsst n S ≡ varssst (proj_unl n S)"

abbreviation bvarslsst where "bvarslsst S ≡ bvarssst (unlabel S)"
abbreviation fvlsst where "fvlsst S ≡ fvsst (unlabel S)"
```

Labeled set-operations

```
fun setopslsstp where
  "setopslsstp (i,insert⟨t,s⟩) = {(i,t,s)}"
| "setopslsstp (i,delete⟨t,s⟩) = {(i,t,s)}"
| "setopslsstp (i,⟨_: t ∈ s⟩) = {(i,t,s)}"
| "setopslsstp (i,∀_⟨∨≠: _ ∨∉: F'⟩) = ((λ(t,s). (i,t,s)) ` set F')"
| "setopslsstp _ = {}"

definition setopslsst where
  "setopslsst S ≡ ⋃(setopslsstp ` set S)"
```

6.1.2 Minor Lemmata

```
lemma subst_lsst_nil[simp]: "[] ·lsst δ = []"
by (simp add: subst_apply_labeled_stateful_strand_def)

lemma subst_lsst_cons: "a#A ·lsst δ = (a ·lsstp δ)#(A ·lsst δ)"
by (simp add: subst_apply_labeled_stateful_strand_def)

lemma subst_lsst_singleton: "[(l,s)] ·lsst δ = [(l,s ·sstp δ)]"
by (simp add: subst_apply_labeled_stateful_strand_def)

lemma subst_lsst_append: "A@B ·lsst δ = (A ·lsst δ)@(B ·lsst δ)"
by (simp add: subst_apply_labeled_stateful_strand_def)

lemma subst_lsst_append_inv:
  assumes "A ·lsst δ = B1@B2"
  shows "∃A1 A2. A = A1@A2 ∧ A1 ·lsst δ = B1 ∧ A2 ·lsst δ = B2"
using assms
proof (induction A arbitrary: B1 B2)
  case (Cons a A)
  note prems = Cons.prems
  note IH = Cons.IH
  show ?case
  proof (cases B1)
    case Nil
    then obtain b B3 where "B2 = b#B3" "a ·lsstp δ = b" "A ·lsst δ = B3"
      using prems subst_lsst_cons by fastforce
    thus ?thesis by (simp add: Nil subst_apply_labeled_stateful_strand_def)
  next
    case (Cons b B3)
    hence "a ·lsstp δ = b" "A ·lsst δ = B3@B2"
      using prems by (simp_all add: subst_lsst_cons)
    thus ?thesis by (metis Cons_eq_appendI Cons IH subst_lsst_cons)
  qed
qed (metis append_is_Nil_conv subst_lsst_nil)

lemma subst_lsst_member[intro]: "x ∈ set A ⟹ x ·lsstp δ ∈ set (A ·lsst δ)"
by (metis image_eqI set_map subst_apply_labeled_stateful_strand_def)
```

```
lemma subst_lsst_unlabel_cons: "unlabel ((l,b)#A ·lsst ϑ) = (b ·sstp ϑ)#(unlabel (A ·lsst ϑ))"
by (simp add: subst_apply_labeled_stateful_strand_def)

lemma subst_lsst_unlabel: "unlabel (A ·lsst δ) = unlabel A ·sst δ"
proof (induction A)
  case (Cons a A)
  then obtain l b where "a = (l,b)" by (metis surj_pair)
  thus ?case
    using Cons
    by (simp add: subst_apply_labeled_stateful_strand_def subst_apply_stateful_strand_def)
qed simp

lemma subst_lsst_unlabel_member[intro]:
  assumes "x ∈ set (unlabel A)"
  shows "x ·sstp δ ∈ set (unlabel (A ·lsst δ))"
proof -
  obtain l where x: "(l,x) ∈ set A" using assms unfolding unlabel_def by moura
  thus ?thesis
    using subst_lsst_member
    by (metis unlabel_def in_set_zipE subst_apply_labeled_stateful_strand_step.simps zip_map_fst_snd)
qed

lemma subst_lsst_prefix:
  assumes "prefix B (A ·lsst ϑ)"
  shows "∃C. C ·lsst ϑ = B ∧ prefix C A"
using assms
proof (induction A rule: List.rev_induct)
  case (snoc a A) thus ?case
  proof (cases "B = A@[a] ·lsst ϑ")
    case False thus ?thesis
      using snoc by (auto simp add: subst_lsst_append[of A] subst_lsst_cons)
  qed auto
qed simp

lemma duallsst_nil[simp]: "duallsst [] = []"
by (simp add: duallsst_def)

lemma duallsst_Cons[simp]:
  "duallsst ((l,send⟨t⟩)#A) = (l,receive⟨t⟩)#(duallsst A)"
  "duallsst ((l,receive⟨t⟩)#A) = (l,send⟨t⟩)#(duallsst A)"
  "duallsst ((l,⟨a: t ≐ s⟩)#A) = (l,⟨a: t ≐ s⟩)#(duallsst A)"
  "duallsst ((l,insert⟨t,s⟩)#A) = (l,insert⟨t,s⟩)#(duallsst A)"
  "duallsst ((l,delete⟨t,s⟩)#A) = (l,delete⟨t,s⟩)#(duallsst A)"
  "duallsst ((l,⟨a: t ∈ s⟩)#A) = (l,⟨a: t ∈ s⟩)#(duallsst A)"
  "duallsst ((l,∀X⟨∨≠: F ∨∉: G⟩)#A) = (l,∀X⟨∨≠: F ∨∉: G⟩)#(duallsst A)"
by (simp_all add: duallsst_def)

lemma duallsst_append[simp]: "duallsst (A@B) = duallsst A@duallsst B"
by (simp add: duallsst_def)

lemma duallsstp_subst: "duallsstp (s ·lsstp δ) = (duallsstp s) ·lsstp δ"
proof -
  obtain l x  where s: "s = (l,x)" by moura
  thus ?thesis by (cases x) (auto simp add: subst_apply_labeled_stateful_strand_def)
qed

lemma duallsst_subst: "duallsst (S ·lsst δ) = (duallsst S) ·lsst δ"
proof (induction S)
  case (Cons s S) thus ?case
    using Cons duallsstp_subst[of s δ]
    by (simp add: duallsst_def subst_apply_labeled_stateful_strand_def)
qed (simp add: duallsst_def subst_apply_labeled_stateful_strand_def)
```

```
lemma dual_lsst_subst_unlabel: "unlabel (dual_lsst (S ·lsst δ)) = unlabel (dual_lsst S) ·sst δ"
by (metis dual_lsst_subst subst_lsst_unlabel)

lemma dual_lsst_subst_cons: "dual_lsst (a#A ·lsst σ) = (dual_lsstp a ·lsstp σ)#(dual_lsst (A ·lsst σ))"
by (metis dual_lsst_subst list.simps(9) dual_lsst_def subst_apply_labeled_stateful_strand_def)

lemma dual_lsst_subst_append: "dual_lsst (A@B ·lsst σ) = (dual_lsst A@dual_lsst B) ·lsst σ"
by (metis (no_types) dual_lsst_subst dual_lsst_append)

lemma dual_lsst_subst_snoc: "dual_lsst (A@[a] ·lsst σ) = (dual_lsst A ·lsst σ)@[dual_lsstp a ·lsstp σ]"
by (metis dual_lsst_def dual_lsst_subst dual_lsst_subst_cons list.map(1) map_append
         subst_apply_labeled_stateful_strand_def)

lemma dual_lsst_memberD:
  assumes "(l,a) ∈ set (dual_lsst A)"
  shows "∃b. (l,b) ∈ set A ∧ dual_lsstp (l,b) = (l,a)"
  using assms
proof (induction A)
  case (Cons c A)
  hence "(l,a) ∈ set (dual_lsst A) ∨ dual_lsstp c = (l,a)" unfolding dual_lsst_def by force
  thus ?case
  proof
    assume "(l,a) ∈ set (dual_lsst A)" thus ?case using Cons.IH by auto
  next
    assume a: "dual_lsstp c = (l,a)"
    obtain i b where b: "c = (i,b)" by (metis surj_pair)
    thus ?case using a by (cases b) auto
  qed
qed simp

lemma dual_lsstp_inv:
  assumes "dual_lsstp (l, a) = (k, b)"
  shows "l = k"
    and "a = receive⟨t⟩ ⟹ b = send⟨t⟩"
    and "a = send⟨t⟩ ⟹ b = receive⟨t⟩"
    and "(∄t. a = receive⟨t⟩ ∨ a = send⟨t⟩) ⟹ b = a"
proof -
  show "l = k" using assms by (cases a) auto
  show "a = receive⟨t⟩ ⟹ b = send⟨t⟩" using assms by (cases a) auto
  show "a = send⟨t⟩ ⟹ b = receive⟨t⟩" using assms by (cases a) auto
  show "(∄t. a = receive⟨t⟩ ∨ a = send⟨t⟩) ⟹ b = a" using assms by (cases a) auto
qed

lemma dual_lsst_self_inverse: "dual_lsst (dual_lsst A) = A"
proof (induction A)
  case (Cons a A)
  obtain l b where "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons by (cases b) auto
qed simp

lemma vars_sst_unlabel_dual_lsst_eq: "vars_lsst (dual_lsst A) = vars_lsst A"
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons.IH by (cases b) auto
qed simp

lemma fv_sst_unlabel_dual_lsst_eq: "fv_lsst (dual_lsst A) = fv_lsst A"
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons.IH by (cases b) auto
qed simp
```

```
lemma bvars_sst_unlabel_dual_lsst_eq: "bvars_lsst (dual_lsst A) = bvars_lsst A"
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons.IH by (cases b) simp+
qed simp

lemma vars_sst_unlabel_Cons: "vars_lsst ((l,b)#A) = vars_sstp b ∪ vars_lsst A"
by (metis unlabel_Cons(1) vars_sst_Cons)

lemma fv_sst_unlabel_Cons: "fv_lsst ((l,b)#A) = fv_sstp b ∪ fv_lsst A"
by (metis unlabel_Cons(1) fv_sst_Cons)

lemma bvars_sst_unlabel_Cons: "bvars_lsst ((l,b)#A) = set (bvars_sstp b) ∪ bvars_lsst A"
by (metis unlabel_Cons(1) bvars_sst_Cons)

lemma bvars_lsst_subst: "bvars_lsst (A ·_lsst δ) = bvars_lsst A"
by (metis subst_lsst_unlabel bvars_sst_subst)

lemma dual_lsst_member:
  assumes "(l,x) ∈ set A"
    and "¬is_Receive x" "¬is_Send x"
  shows "(l,x) ∈ set (dual_lsst A)"
using assms
proof (induction A)
  case (Cons a A) thus ?case using assms(2,3) by (cases x) (auto simp add: dual_lsst_def)
qed simp

lemma dual_lsst_unlabel_member:
  assumes "x ∈ set (unlabel A)"
    and "¬is_Receive x" "¬is_Send x"
  shows "x ∈ set (unlabel (dual_lsst A))"
using assms dual_lsst_member[of _ _ A]
  by (meson unlabel_in unlabel_mem_has_label)

lemma dual_lsst_steps_iff:
  "(l,send⟨t⟩) ∈ set A ⟷ (l,receive⟨t⟩) ∈ set (dual_lsst A)"
  "(l,receive⟨t⟩) ∈ set A ⟷ (l,send⟨t⟩) ∈ set (dual_lsst A)"
  "(l,⟨c: t ≐ s⟩) ∈ set A ⟷ (l,⟨c: t ≐ s⟩) ∈ set (dual_lsst A)"
  "(l,insert⟨t,s⟩) ∈ set A ⟷ (l,insert⟨t,s⟩) ∈ set (dual_lsst A)"
  "(l,delete⟨t,s⟩) ∈ set A ⟷ (l,delete⟨t,s⟩) ∈ set (dual_lsst A)"
  "(l,⟨c: t ∈ s⟩) ∈ set A ⟷ (l,⟨c: t ∈ s⟩) ∈ set (dual_lsst A)"
  "(l,∀X⟨∨≠: F ∨∉: G⟩) ∈ set A ⟷ (l,∀X⟨∨≠: F ∨∉: G⟩) ∈ set (dual_lsst A)"
proof (induction A)
  case (Cons a A)
  obtain j b where a: "a = (j,b)" by (metis surj_pair)
  { case 1 thus ?case by (cases b) (simp_all add: Cons.IH(1) a dual_lsst_def) }
  { case 2 thus ?case by (cases b) (simp_all add: Cons.IH(2) a dual_lsst_def) }
  { case 3 thus ?case by (cases b) (simp_all add: Cons.IH(3) a dual_lsst_def) }
  { case 4 thus ?case by (cases b) (simp_all add: Cons.IH(4) a dual_lsst_def) }
  { case 5 thus ?case by (cases b) (simp_all add: Cons.IH(5) a dual_lsst_def) }
  { case 6 thus ?case by (cases b) (simp_all add: Cons.IH(6) a dual_lsst_def) }
  { case 7 thus ?case by (cases b) (simp_all add: Cons.IH(7) a dual_lsst_def) }
qed (simp_all add: dual_lsst_def)

lemma dual_lsst_unlabel_steps_iff:
  "send⟨t⟩ ∈ set (unlabel A) ⟷ receive⟨t⟩ ∈ set (unlabel (dual_lsst A))"
  "receive⟨t⟩ ∈ set (unlabel A) ⟷ send⟨t⟩ ∈ set (unlabel (dual_lsst A))"
  "⟨c: t ≐ s⟩ ∈ set (unlabel A) ⟷ ⟨c: t ≐ s⟩ ∈ set (unlabel (dual_lsst A))"
  "insert⟨t,s⟩ ∈ set (unlabel A) ⟷ insert⟨t,s⟩ ∈ set (unlabel (dual_lsst A))"
  "delete⟨t,s⟩ ∈ set (unlabel A) ⟷ delete⟨t,s⟩ ∈ set (unlabel (dual_lsst A))"
  "⟨c: t ∈ s⟩ ∈ set (unlabel A) ⟷ ⟨c: t ∈ s⟩ ∈ set (unlabel (dual_lsst A))"
```

```
  "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel A) ⟷ ∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (dual_lsst A))"
using dual_lsst_steps_iff(1,2)[of _ t A]
      dual_lsst_steps_iff(3,6)[of _ c t s A]
      dual_lsst_steps_iff(4,5)[of _ t s A]
      dual_lsst_steps_iff(7)[of _ X F G A]
by (meson unlabel_in unlabel_mem_has_label)+

lemma dual_lsst_list_all:
  "list_all is_Receive (unlabel A) ⟹ list_all is_Send (unlabel (dual_lsst A))"
  "list_all is_Send (unlabel A) ⟹ list_all is_Receive (unlabel (dual_lsst A))"
  "list_all is_Equality (unlabel A) ⟹ list_all is_Equality (unlabel (dual_lsst A))"
  "list_all is_Insert (unlabel A) ⟹ list_all is_Insert (unlabel (dual_lsst A))"
  "list_all is_Delete (unlabel A) ⟹ list_all is_Delete (unlabel (dual_lsst A))"
  "list_all is_InSet (unlabel A) ⟹ list_all is_InSet (unlabel (dual_lsst A))"
  "list_all is_NegChecks (unlabel A) ⟹ list_all is_NegChecks (unlabel (dual_lsst A))"
  "list_all is_Assignment (unlabel A) ⟹ list_all is_Assignment (unlabel (dual_lsst A))"
  "list_all is_Check (unlabel A) ⟹ list_all is_Check (unlabel (dual_lsst A))"
  "list_all is_Update (unlabel A) ⟹ list_all is_Update (unlabel (dual_lsst A))"
proof (induct A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  { case 1 thus ?case using Cons.hyps(1) a by (cases b) auto }
  { case 2 thus ?case using Cons.hyps(2) a by (cases b) auto }
  { case 3 thus ?case using Cons.hyps(3) a by (cases b) auto }
  { case 4 thus ?case using Cons.hyps(4) a by (cases b) auto }
  { case 5 thus ?case using Cons.hyps(5) a by (cases b) auto }
  { case 6 thus ?case using Cons.hyps(6) a by (cases b) auto }
  { case 7 thus ?case using Cons.hyps(7) a by (cases b) auto }
  { case 8 thus ?case using Cons.hyps(8) a by (cases b) auto }
  { case 9 thus ?case using Cons.hyps(9) a by (cases b) auto }
  { case 10 thus ?case using Cons.hyps(10) a by (cases b) auto }
qed simp_all

lemma dual_lsst_in_set_prefix_obtain:
  assumes "s ∈ set (unlabel (dual_lsst A))"
  shows "∃l B s'. (l,s) = dual_lsstp (l,s') ∧ prefix (B@[(l,s')]) A"
  using assms
proof (induction A rule: List.rev_induct)
  case (snoc a A)
  obtain i b where a: "a = (i,b)" by (metis surj_pair)
  show ?case using snoc
  proof (cases "s ∈ set (unlabel (dual_lsst A))")
    case False thus ?thesis
      using a snoc.prems unlabel_append[of "dual_lsst A" "dual_lsst [a]"] dual_lsst_append[of A "[a]"]
      by (cases b) (force simp add: unlabel_def dual_lsst_def)+
  qed auto
qed simp

lemma dual_lsst_in_set_prefix_obtain_subst:
  assumes "s ∈ set (unlabel (dual_lsst (A ·lsst ϑ)))"
  shows "∃l B s'. (l,s) = dual_lsstp ((l,s') ·lsstp ϑ) ∧ prefix ((B ·lsst ϑ)@[(l,s') ·lsstp ϑ]) (A ·lsst
ϑ)"
proof -
  obtain B l s' where B: "(l,s) = dual_lsstp (l,s')" "prefix (B@[(l,s')]) (A ·lsst ϑ)"
    using dual_lsst_in_set_prefix_obtain[OF assms] by moura

  obtain C where C: "C ·lsst ϑ = B@[(l,s')]"
    using subst_lsst_prefix[OF B(2)] by moura

  obtain D u where D: "C = D@[(l,u)]" "D ·lsst ϑ = B" "[(l,u)] ·lsst ϑ = [(l, s')]"
    using subst_lsst_prefix[OF B(2)] subst_lsst_append_inv[OF C(1)]
    by (auto simp add: subst_apply_labeled_stateful_strand_def)
```

```
  show ?thesis
    using B D subst_lsst_cons subst_lsst_singleton
    by (metis (no_types, lifting) nth_append_length)
qed

lemma trms_sst_unlabel_dual_lsst_eq: "trms_lsst (dual_lsst A) = trms_lsst A"
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons.IH by (cases b) auto
qed simp

lemma trms_sst_unlabel_subst_cons:
  "trms_lsst ((l,b)#A ·_lsst δ) = trms_sstp (b ·_sstp δ) ∪ trms_lsst (A ·_lsst δ)"
by (metis subst_lsst_unlabel trms_sst_subst_cons unlabel_Cons(1))

lemma trms_sst_unlabel_subst:
  assumes "bvars_lsst S ∩ subst_domain ϑ = {}"
  shows "trms_lsst (S ·_lsst ϑ) = trms_lsst S ·_set ϑ"
by (metis trms_sst_subst[OF assms] subst_lsst_unlabel)

lemma trms_sst_unlabel_subst':
  fixes t::"('a,'b) term" and δ::"('a,'b) subst"
  assumes "t ∈ trms_lsst (S ·_lsst δ)"
  shows "∃s ∈ trms_lsst S. ∃X. set X ⊆ bvars_lsst S ∧ t = s · rm_vars (set X) δ"
using assms
proof (induction S)
  case (Cons a S)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  hence "t ∈ trms_lsst (S ·_lsst δ) ∨ t ∈ trms_sstp (b ·_sstp δ)"
    using Cons.prems trms_sst_unlabel_subst_cons by fast
  thus ?case
  proof
    assume *: "t ∈ trms_sstp (b ·_sstp δ)"
    show ?thesis using trms_sstp_subst''[OF *] a by auto
  next
    assume *: "t ∈ trms_lsst (S ·_lsst δ)"
    show ?thesis using Cons.IH[OF *] a by auto
  qed
qed simp

lemma trms_sst_unlabel_subst'':
  fixes t::"('a,'b) term" and δ ϑ::"('a,'b) subst"
  assumes "t ∈ trms_lsst (S ·_lsst δ) ·_set ϑ"
  shows "∃s ∈ trms_lsst S. ∃X. set X ⊆ bvars_lsst S ∧ t = s · rm_vars (set X) δ ∘_s ϑ"
proof -
  obtain s where s: "s ∈ trms_lsst (S ·_lsst δ)" "t = s · ϑ" using assms by moura
  show ?thesis using trms_sst_unlabel_subst'[OF s(1)] s(2) by auto
qed

lemma trms_sst_unlabel_dual_subst_cons:
  "trms_lsst (dual_lsst (a#A ·_lsst σ)) = (trms_sstp (snd a ·_sstp σ)) ∪ (trms_lsst (dual_lsst (A ·_lsst σ)))"
proof -
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?thesis using a dual_lsst_subst_cons[of a A σ] by (cases b) auto
qed

lemma dual_lsst_funs_term:
  "⋃(funs_term ` (trms_sst (unlabel (dual_lsst S)))) = ⋃(funs_term ` (trms_sst (unlabel S)))"
using trms_sst_unlabel_dual_lsst_eq by fast

lemma dual_lsst_db_lsst:
  "db'_lsst (dual_lsst A) = db'_lsst A"
```

```
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons by (cases b) auto
qed simp

lemma db_sst_unlabel_append:
  "db'_lsst (A@B) I D = db'_lsst B I (db'_lsst A I D)"
by (metis db_sst_append unlabel_append)

lemma db_sst_dual_lsst:
  "db'_sst (unlabel (dual_lsst (T ·_lsst δ))) ℐ D = db'_sst (unlabel (T ·_lsst δ)) ℐ D"
proof (induction T arbitrary: D)
  case (Cons x T)
  obtain l s where "x = (l,s)" by moura
  thus ?case
    using Cons
    by (cases s) (simp_all add: unlabel_def dual_lsst_def subst_apply_labeled_stateful_strand_def)
qed (simp add: unlabel_def dual_lsst_def subst_apply_labeled_stateful_strand_def)

lemma labeled_list_insert_eq_cases:
  "d ∉ set (unlabel D) ⟹ List.insert d (unlabel D) = unlabel (List.insert (i,d) D)"
  "(i,d) ∈ set D ⟹ List.insert d (unlabel D) = unlabel (List.insert (i,d) D)"
unfolding unlabel_def
by (metis (no_types, hide_lams) List.insert_def image_eqI list.simps(9) set_map snd_conv,
    metis in_set_insert set_zip_rightD zip_map_fst_snd)

lemma labeled_list_insert_eq_ex_cases:
  "List.insert d (unlabel D) = unlabel (List.insert (i,d) D) ∨
  (∃j. (j,d) ∈ set D ∧ List.insert d (unlabel D) = unlabel (List.insert (j,d) D))"
using labeled_list_insert_eq_cases unfolding unlabel_def
by (metis in_set_impl_in_set_zip2 length_map zip_map_fst_snd)

lemma proj_subst: "proj l (A ·_lsst δ) = proj l A ·_lsst δ"
proof (induction A)
  case (Cons a A)
  obtain l b where "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons unfolding proj_def subst_apply_labeled_stateful_strand_def by force
qed simp

lemma proj_set_subset[simp]:
  "set (proj n A) ⊆ set A"
unfolding proj_def by auto

lemma proj_proj_set_subset[simp]:
  "set (proj n (proj m A)) ⊆ set (proj n A)"
  "set (proj n (proj m A)) ⊆ set (proj m A)"
  "set (proj_unl n (proj m A)) ⊆ set (proj_unl n A)"
  "set (proj_unl n (proj m A)) ⊆ set (proj_unl m A)"
unfolding unlabel_def proj_def by auto

lemma proj_in_set_iff:
  "(ln i, d) ∈ set (proj i D) ⟷ (ln i, d) ∈ set D"
  "(⋆, d) ∈ set (proj i D) ⟷ (⋆, d) ∈ set D"
unfolding proj_def by auto

lemma proj_list_insert:
  "proj i (List.insert (ln i,d) D) = List.insert (ln i,d) (proj i D)"
  "proj i (List.insert (⋆,d) D) = List.insert (⋆,d) (proj i D)"
  "i ≠ j ⟹ proj i (List.insert (ln j,d) D) = proj i D"
unfolding List.insert_def proj_def by auto

lemma proj_filter: "proj i [d←D. d ∉ set Di] = [d←proj i D. d ∉ set Di]"
```

```
by (simp_all add: proj_def conj_commute)

lemma proj_list_Cons:
  "proj i ((ln i,d)#D) = (ln i,d)#proj i D"
  "proj i ((⋆,d)#D) = (⋆,d)#proj i D"
  "i ≠ j ⟹ proj i ((ln j,d)#D) = proj i D"
unfolding List.insert_def proj_def by auto

lemma proj_dual_lsst:
  "proj l (dual_lsst A) = dual_lsst (proj l A)"
proof (induction A)
  case (Cons a A)
  obtain k b where "a = (k,b)" by (metis surj_pair)
  thus ?case using Cons unfolding dual_lsst_def proj_def by (cases b) auto
qed simp

lemma proj_instance_ex:
  assumes B: "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·lsstp δ ∧ P δ"
    and b: "b ∈ set (proj l B)"
  shows "∃a ∈ set (proj l A). ∃δ. b = a ·lsstp δ ∧ P δ"
proof -
  obtain a δ where a: "a ∈ set A" "b = a ·lsstp δ" "P δ" using B b proj_set_subset by fast
  obtain k b' where b': "b = (k, b')" "k = (ln l) ∨ k = ⋆" using b proj_in_setD by metis
  obtain a' where a': "a = (k, a')" using b'(1) a(2) by (cases a) simp_all
  show ?thesis using a a' b'(2) unfolding proj_def by auto
qed

lemma proj_dbproj:
  "dbproj (ln i) (proj i D) = dbproj (ln i) D"
  "dbproj ⋆ (proj i D) = dbproj ⋆ D"
  "i ≠ j ⟹ dbproj (ln j) (proj i D) = []"
unfolding proj_def by (induct D) auto

lemma dbproj_Cons:
  "dbproj i ((i,d)#D) = (i,d)#dbproj i D"
  "i ≠ j ⟹ dbproj j ((i,d)#D) = dbproj j D"
by auto

lemma dbproj_subset[simp]:
  "set (unlabel (dbproj i D)) ⊆ set (unlabel D)"
unfolding unlabel_def by auto

lemma dbproj_subseq:
  assumes "Di ∈ set (subseqs (dbproj k D))"
  shows "dbproj k Di = Di" (is ?A)
  and "i ≠ k ⟹ dbproj i Di = []" (is "i ≠ k ⟹ ?B")
proof -
  have *: "set Di ⊆ set (dbproj k D)" using subseqs_powset[of "dbproj k D"] assms by auto
  thus ?A by (metis filter_True filter_set member_filter subsetCE)

  have "⋀j d. (j,d) ∈ set Di ⟹ j = k" using * by auto
  moreover have "⋀j d. (j,d) ∈ set (dbproj i Di) ⟹ j = i" by auto
  moreover have "⋀j d. (j,d) ∈ set (dbproj i Di) ⟹ (j,d) ∈ set Di" by auto
  ultimately show "i ≠ k ⟹ ?B" by (metis set_empty subrelI subset_empty)
qed

lemma dbproj_subseq_subset:
  assumes "Di ∈ set (subseqs (dbproj i D))"
  shows "set Di ⊆ set D"
by (metis Pow_iff assms filter_set image_eqI member_filter subseqs_powset subsetCE subsetI)

lemma dbproj_subseq_in_subseqs:
  assumes "Di ∈ set (subseqs (dbproj i D))"
```

```
  shows "Di ∈ set (subseqs D)"
using assms in_set_subseqs subseq_filter_left subseq_order.dual_order.trans by blast

lemma proj_subseq:
  assumes "Di ∈ set (subseqs (dbproj (ln j) D))" "j ≠ i"
  shows "[d←proj i D. d ∉ set Di] = proj i D"
proof -
  have "set Di ⊆ set (dbproj (ln j) D)" using subseqs_powset[of "dbproj (ln j) D"] assms by auto
  hence "⋀k d. (k,d) ∈ set Di ⟹ k = ln j" by auto
  moreover have "⋀k d. (k,d) ∈ set (proj i D) ⟹ k ≠ ln j"
    using assms(2) unfolding proj_def by auto
  ultimately have "⋀d. d ∈ set (proj i D) ⟹ d ∉ set Di" by auto
  thus ?thesis by simp
qed

lemma unlabel_subseqsD:
  assumes "A ∈ set (subseqs (unlabel B))"
  shows "∃C ∈ set (subseqs B). unlabel C = A"
using assms map_subseqs unfolding unlabel_def by (metis imageE set_map)

lemma unlabel_filter_eq:
  assumes "∀(j, p) ∈ set A ∪ B. ∀(k, q) ∈ set A ∪ B. p = q ⟶ j = k" (is "?P (set A)")
  shows "[d←unlabel A. d ∉ snd ‘ B] = unlabel [d←A. d ∉ B]"
using assms unfolding unlabel_def
proof (induction A)
  case (Cons a A)
  have "set A ⊆ set (a#A)" "{a} ⊆ set (a#A)" by auto
  hence *: "?P (set A)" "?P {a}" using Cons.prems by fast+
  hence IH: "[d←map snd A . d ∉ snd ‘ B] = map snd [d←A . d ∉ B]" using Cons.IH by auto

  { assume "snd a ∈ snd ‘ B"
    then obtain b where b: "b ∈ B" "snd a = snd b" by moura
    hence "fst a = fst b" using *(2) by auto
    hence "a ∈ B" using b by (metis surjective_pairing)
  } hence **: "a ∉ B ⟹ snd a ∉ snd ‘ B" by metis

  show ?case by (cases "a ∈ B") (simp add: ** IH)+
qed simp

lemma subseqs_mem_dbproj:
  assumes "Di ∈ set (subseqs D)" "list_all (λd. fst d = i) Di"
  shows "Di ∈ set (subseqs (dbproj i D))"
using assms
proof (induction D arbitrary: Di)
  case (Cons di D)
  obtain d j where di: "di = (j,d)" by (metis surj_pair)
  show ?case
  proof (cases "Di ∈ set (subseqs D)")
    case True
    hence "Di ∈ set (subseqs (dbproj i D))" using Cons.IH Cons.prems by auto
    thus ?thesis using subseqs_Cons by auto
  next
    case False
    then obtain Di' where Di': "Di = di#Di'" using Cons.prems(1)
      by (metis (mono_tags, lifting) Un_iff imageE set_append set_map subseqs.simps(2))
    hence "Di' ∈ set (subseqs D)" using Cons.prems(1) False
      by (metis (no_types, lifting) UnE imageE list.inject set_append set_map subseqs.simps(2))
    hence "Di' ∈ set (subseqs (dbproj i D))" using Cons.IH Cons.prems Di' by auto
    moreover have "i = j" using Di' di Cons.prems(2) by auto
    hence "dbproj i (di#D) = di#dbproj i D" by (simp add: di)
    ultimately show ?thesis using Di'
      by (metis (no_types, lifting) UnCI image_eqI set_append set_map subseqs.simps(2))
  qed
```

```
qed simp

lemma unlabel_subst: "unlabel S ·sst δ = unlabel (S ·lsst δ)"
unfolding unlabel_def subst_apply_stateful_strand_def subst_apply_labeled_stateful_strand_def
by auto

lemma subterms_subst_lsst:
  assumes "∀x ∈ fvset (trmslsst S). (∃f. σ x = Fun f []) ∨ (∃y. σ x = Var y)"
    and "bvarslsst S ∩ subst_domain σ = {}"
  shows "subtermsset (trmslsst (S ·lsst σ)) = subtermsset (trmslsst S) ·set σ"
using subterms_subst''[OF assms(1)] trmssst_subst[OF assms(2)] unlabel_subst[of S σ]
by simp

lemma subterms_subst_lsst_ik:
  assumes "∀x ∈ fvset (iklsst S). (∃f. σ x = Fun f []) ∨ (∃y. σ x = Var y)"
  shows "subtermsset (iklsst (S ·lsst σ)) = subtermsset (iklsst S) ·set σ"
using subterms_subst''[OF assms(1)] iksst_subst[of "unlabel S" σ] unlabel_subst[of S σ]
by simp

lemma labeled_stateful_strand_subst_comp:
  assumes "range_vars δ ∩ bvarslsst S = {}"
  shows "S ·lsst δ ∘s ϑ = (S ·lsst δ) ·lsst ϑ"
using assms
proof (induction S)
  case (Cons s S)
  obtain l x where s: "s = (l,x)" by (metis surj_pair)
  hence IH: "S ·lsst δ ∘s ϑ = (S ·lsst δ) ·lsst ϑ" using Cons by auto

  have "x ·sstp δ ∘s ϑ = (x ·sstp δ) ·sstp ϑ"
    using s Cons.prems stateful_strand_step_subst_comp[of δ x ϑ] by auto
  thus ?case using s IH by (simp add: subst_apply_labeled_stateful_strand_def)
qed simp

lemma sst_vars_proj_subset[simp]:
  "fvsst (proj_unl n A) ⊆ fvsst (unlabel A)"
  "bvarssst (proj_unl n A) ⊆ bvarssst (unlabel A)"
  "varssst (proj_unl n A) ⊆ varssst (unlabel A)"
using varssst_is_fvsst_bvarssst[of "unlabel A"]
      varssst_is_fvsst_bvarssst[of "proj_unl n A"]
unfolding unlabel_def proj_def by auto

lemma trmssst_proj_subset[simp]:
  "trmssst (proj_unl n A) ⊆ trmssst (unlabel A)" (is ?A)
  "trmssst (proj_unl m (proj n A)) ⊆ trmssst (proj_unl n A)" (is ?B)
  "trmssst (proj_unl m (proj n A)) ⊆ trmssst (proj_unl m A)" (is ?C)
proof -
  show ?A unfolding unlabel_def proj_def by auto
  show ?B using trmssst_mono[OF proj_proj_set_subset(4)] by metis
  show ?C using trmssst_mono[OF proj_proj_set_subset(3)] by metis
qed

lemma trmssst_unlabel_prefix_subset:
  "trmssst (unlabel A) ⊆ trmssst (unlabel (A@B))" (is ?A)
  "trmssst (proj_unl n A) ⊆ trmssst (proj_unl n (A@B))" (is ?B)
using trmssst_mono[of "proj_unl n A" "proj_unl n (A@B)"]
unfolding unlabel_def proj_def by auto

lemma trmssst_unlabel_suffix_subset:
  "trmssst (unlabel B) ⊆ trmssst (unlabel (A@B))"
  "trmssst (proj_unl n B) ⊆ trmssst (proj_unl n (A@B))"
using trmssst_mono[of "proj_unl n B" "proj_unl n (A@B)"]
unfolding unlabel_def proj_def by auto
```

```
lemma setops_lsstp D:
  assumes p: "p ∈ setops_lsstp a"
  shows "fst p = fst a" (is ?P)
    and "is_Update (snd a) ∨ is_InSet (snd a) ∨ is_NegChecks (snd a)" (is ?Q)
proof -
  obtain l k p' a' where a: "p = (l,p')" "a = (k,a')" by (metis surj_pair)
  show ?P using p a by (cases a') auto
  show ?Q using p a by (cases a') auto
qed

lemma setops_lsst_nil[simp]:
  "setops_lsst [] = {}"
by (simp add: setops_lsst_def)

lemma setops_lsst_cons[simp]:
  "setops_lsst (x#S) = setops_lsstp x ∪ setops_lsst S"
by (simp add: setops_lsst_def)

lemma setops_sst_proj_subset:
  "setops_sst (proj_unl n A) ⊆ setops_sst (unlabel A)"
  "setops_sst (proj_unl m (proj n A)) ⊆ setops_sst (proj_unl n A)"
  "setops_sst (proj_unl m (proj n A)) ⊆ setops_sst (proj_unl m A)"
unfolding unlabel_def proj_def
proof (induction A)
  case (Cons a A)
  obtain l b where lb: "a = (l,b)" by moura
  { case 1 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
  { case 2 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
  { case 3 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
qed simp_all

lemma setops_sst_unlabel_prefix_subset:
  "setops_sst (unlabel A) ⊆ setops_sst (unlabel (A@B))"
  "setops_sst (proj_unl n A) ⊆ setops_sst (proj_unl n (A@B))"
unfolding unlabel_def proj_def
proof (induction A)
  case (Cons a A)
  obtain l b where lb: "a = (l,b)" by moura
  { case 1 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
  { case 2 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
qed (simp_all add: setops_sst_def)

lemma setops_sst_unlabel_suffix_subset:
  "setops_sst (unlabel B) ⊆ setops_sst (unlabel (A@B))"
  "setops_sst (proj_unl n B) ⊆ setops_sst (proj_unl n (A@B))"
unfolding unlabel_def proj_def
proof (induction A)
  case (Cons a A)
  obtain l b where lb: "a = (l,b)" by moura
  { case 1 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
  { case 2 thus ?case using Cons.IH lb by (cases b) (auto simp add: setops_sst_def) }
qed simp_all

lemma setops_lsst_proj_subset:
  "setops_lsst (proj n A) ⊆ setops_lsst A"
  "setops_lsst (proj m (proj n A)) ⊆ setops_lsst (proj n A)"
unfolding proj_def setops_lsst_def by auto

lemma setops_lsst_prefix_subset:
  "setops_lsst A ⊆ setops_lsst (A@B)"
  "setops_lsst (proj n A) ⊆ setops_lsst (proj n (A@B))"
unfolding proj_def setops_lsst_def by auto
```

```
lemma setops_lsst_suffix_subset:
  "setops_lsst B ⊆ setops_lsst (A@B)"
  "setops_lsst (proj n B) ⊆ setops_lsst (proj n (A@B))"
unfolding proj_def setops_lsst_def by auto

lemma setops_lsst_mono:
  "set M ⊆ set N ⟹ setops_lsst M ⊆ setops_lsst N"
by (auto simp add: setops_lsst_def)

lemma trms_sst_unlabel_subset_if_no_label:
  "¬list_ex (is_LabelN l) A ⟹ trms_lsst (proj l A) ⊆ trms_lsst (proj l' A)"
by (rule trms_sst_mono[OF proj_subset_if_no_label(2)[of l A l']])

lemma setops_sst_unlabel_subset_if_no_label:
  "¬list_ex (is_LabelN l) A ⟹ setops_sst (proj_unl l A) ⊆ setops_sst (proj_unl l' A)"
by (rule setops_sst_mono[OF proj_subset_if_no_label(2)[of l A l']])

lemma setops_lsst_proj_subset_if_no_label:
  "¬list_ex (is_LabelN l) A ⟹ setops_lsst (proj l A) ⊆ setops_lsst (proj l' A)"
by (rule setops_lsst_mono[OF proj_subset_if_no_label(1)[of l A l']])

lemma setops_lsstp_subst_cases[simp]:
  "setops_lsstp ((l,send⟨t⟩) ·_lsstp δ) = {}"
  "setops_lsstp ((l,receive⟨t⟩) ·_lsstp δ) = {}"
  "setops_lsstp ((l,⟨ac: s ≐ t⟩) ·_lsstp δ) = {}"
  "setops_lsstp ((l,insert⟨t,s⟩) ·_lsstp δ) = {(l,t · δ,s · δ)}"
  "setops_lsstp ((l,delete⟨t,s⟩) ·_lsstp δ) = {(l,t · δ,s · δ)}"
  "setops_lsstp ((l,⟨ac: t ∈ s⟩) ·_lsstp δ) = {(l,t · δ,s · δ)}"
  "setops_lsstp ((l,∀X⟨∨≠: F ∨∉: F'⟩) ·_lsstp δ) =
    ((λ(t,s). (l,t · rm_vars (set X) δ,s · rm_vars (set X) δ)) ` set F')" (is "?A = ?B")
proof -
  have "?A = (λ(t,s). (l,t,s)) ` set (F' ·_pairs rm_vars (set X) δ)" by auto
  thus "?A = ?B" unfolding subst_apply_pairs_def by auto
qed simp_all

lemma setops_lsstp_subst:
  assumes "set (bvars_sstp (snd a)) ∩ subst_domain ϑ = {}"
  shows "setops_lsstp (a ·_lsstp ϑ) = (λp. (fst a,snd p ·_p ϑ)) ` setops_lsstp a"
proof -
  obtain l a' where a: "a = (l,a')" by (metis surj_pair)
  show ?thesis
  proof (cases a')
    case (NegChecks X F G)
    hence *: "rm_vars (set X) ϑ = ϑ" using a assms rm_vars_apply'[of ϑ "set X"] by auto
    have "setops_lsstp (a ·_lsstp ϑ) = (λp. (fst a, p)) ` set (G ·_pairs ϑ)"
      using * NegChecks a  by auto
    moreover have "setops_lsstp a = (λp. (fst a, p)) ` set G" using NegChecks a by simp
    hence "(λp. (fst a,snd p ·_p ϑ)) ` setops_lsstp a = (λp. (fst a, p ·_p ϑ)) ` set G"
      by (metis (mono_tags, lifting) image_cong image_image snd_conv)
    hence "(λp. (fst a,snd p ·_p ϑ)) ` setops_lsstp a = (λp. (fst a, p)) ` (set G ·_pset ϑ)"
      unfolding case_prod_unfold by auto
    ultimately show ?thesis by (simp add: subst_apply_pairs_def)
  qed (use a in simp_all)
qed

lemma setops_lsstp_subst':
  assumes "set (bvars_sstp (snd a)) ∩ subst_domain ϑ = {}"
  shows "setops_lsstp (a ·_lsstp ϑ) = (λ(i,p). (i,p ·_p ϑ)) ` setops_lsstp a"
using setops_lsstp_subst[OF assms] setops_lsstpD(1) unfolding case_prod_unfold
by (metis (mono_tags, lifting) image_cong)

lemma setops_lsst_subst:
  assumes "bvars_lsst S ∩ subst_domain ϑ = {}"
```

```
  shows "setops_lsst (S ·_lsst ϑ) = (λp. (fst p,snd p ·_p ϑ)) ‘ setops_lsst S"
using assms
proof (induction S)
  case (Cons a S)
  have "bvars_lsst S ∩ subst_domain ϑ = {}" and *: "set (bvars_sstp (snd a)) ∩ subst_domain ϑ = {}"
    using Cons.prems by auto
  hence IH: "setops_lsst (S ·_lsst ϑ) = (λp. (fst p,snd p ·_p ϑ)) ‘ setops_lsst S"
    using Cons.IH by auto
  show ?case
    using setops_lsstp_subst'[OF *] IH
    unfolding setops_lsst_def case_prod_unfold subst_lsst_cons
    by auto
qed (simp add: setops_sst_def)

lemma setops_lsstp_in_subst:
  assumes p: "p ∈ setops_lsstp (a ·_lsstp δ)"
  shows "∃q ∈ setops_lsstp a. fst p = fst q ∧ snd p = snd q ·_p rm_vars (set (bvars_sstp (snd a))) δ"
    (is "∃q ∈ setops_lsstp a. ?P q")
proof -
  obtain l b where a: "a = (l,b)" by (metis surj_pair)

  show ?thesis
  proof (cases b)
    case (NegChecks X F F')
    hence "p ∈ (λ(t, s). (l, t · rm_vars (set X) δ, s · rm_vars (set X) δ)) ‘ set F'"
      using p a setops_lsstp_subst_cases(7)[of l X F F' δ] by blast
    then obtain s t where st:
        "(t,s) ∈ set F'" "p = (l, t · rm_vars (set X) δ, s · rm_vars (set X) δ)"
      by auto
    hence "(l,t,s) ∈ setops_lsstp a" "fst p = fst (l,t,s)"
          "snd p = snd (l,t,s) ·_p rm_vars (set X) δ"
      using a NegChecks by fastforce+
    moreover have "bvars_sstp (snd a) = X" using NegChecks a by auto
    ultimately show ?thesis by blast
  qed (use p a in auto)
qed

lemma setops_lsst_in_subst:
  assumes "p ∈ setops_lsst (A ·_lsst δ)"
  shows "∃q ∈ setops_lsst A. fst p = fst q ∧ (∃X ⊆ bvars_lsst A. snd p = snd q ·_p rm_vars X δ)"
    (is "∃q ∈ setops_lsst A. ?P A q")
  using assms
proof (induction A)
  case (Cons a A)
  note 0 = unlabel_Cons(2)[of a A] bvars_sst_Cons[of "snd a" "unlabel A"]
  show ?case
  proof (cases "p ∈ setops_lsst (A ·_lsst δ)")
    case False
    hence "p ∈ setops_lsstp (a ·_lsstp δ)"
      using Cons.prems setops_lsst_cons[of "a ·_lsstp δ" "A ·_lsst δ"] subst_lsst_cons[of a A δ] by auto
    moreover have "(set (bvars_sstp (snd a))) ⊆ bvars_lsst (a#A)" using 0 by simp
    ultimately have "∃q ∈ setops_lsstp a. ?P (a#A) q" using setops_lsstp_in_subst[of p a δ] by blast
    thus ?thesis by auto
  qed (use Cons.IH 0 in auto)
qed simp

lemma setops_lsst_dual_lsst_eq:
  "setops_lsst (dual_lsst A) = setops_lsst A"
proof (induction A)
  case (Cons a A)
  obtain l b where "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons unfolding setops_lsst_def dual_lsst_def by (cases b) auto
qed simp
```

end

6.2 Stateful Protocol Compositionality (Stateful_Compositionality)

```
theory Stateful_Compositionality
imports Stateful_Typing Parallel_Compositionality Labeled_Stateful_Strands
begin
```

6.2.1 Small Lemmata

```
lemma (in typed_model) wt_subst_sstp_vars_type_subset:
  fixes a::"('fun,'var) stateful_strand_step"
  assumes "wt_subst δ"
    and "∀t ∈ subst_range δ. fv t = {} ∨ (∃x. t = Var x)"
  shows "Γ ` Var ` fv_sstp (a ·sstp δ) ⊆ Γ ` Var ` fv_sstp a" (is ?A)
    and "Γ ` Var ` set (bvars_sstp (a ·sstp δ)) = Γ ` Var ` set (bvars_sstp a)" (is ?B)
    and "Γ ` Var ` vars_sstp (a ·sstp δ) ⊆ Γ ` Var ` vars_sstp a" (is ?C)
proof -
  show ?A
  proof
    fix τ assume τ: "τ ∈ Γ ` Var ` fv_sstp (a ·sstp δ)"
    then obtain x where x: "x ∈ fv_sstp (a ·sstp δ)" "Γ (Var x) = τ" by moura

    show "τ ∈ Γ ` Var ` fv_sstp a"
    proof (cases "x ∈ fv_sstp a")
      case False
      hence "∃y ∈ fv_sstp a. δ y = Var x"
      proof (cases a)
        case (NegChecks X F G)
        hence *: "x ∈ fv_pairs (F ·pairs rm_vars (set X) δ) ∪ fv_pairs (G ·pairs rm_vars (set X) δ)"
                 "x ∉ set X"
          using fv_sstp_NegCheck(1)[of X "F ·pairs rm_vars (set X) δ" "G ·pairs rm_vars (set X) δ"]
                fv_sstp_NegCheck(1)[of X F G] False x(1)
          by fastforce+

        obtain y where y: "y ∈ fv_pairs F ∪ fv_pairs G" "x ∈ fv (rm_vars (set X) δ y)"
          using fv_pairs_subst_obtain_var[of _ _ "rm_vars (set X) δ"]
                fv_pairs_subst_obtain_var[of _ _ "rm_vars (set X) δ"]
                *(1)
          by blast

        have "fv (rm_vars (set X) δ z) = {} ∨ (∃u. rm_vars (set X) δ z = Var u)" for z
          using assms(2) rm_vars_img_subset[of "set X" δ] by blast
        hence "rm_vars (set X) δ y = Var x" using y(2) by fastforce
        hence "∃y ∈ fv_sstp a. rm_vars (set X) δ y = Var x"
          using y fv_sstp_NegCheck(1)[of X F G] NegChecks *(2) by fastforce
        thus ?thesis by (metis (full_types) *(2) term.inject(1))
      qed (use assms(2) x(1) subst_apply_img_var'[of x _ δ] in fastforce)+
      then obtain y where y: "y ∈ fv_sstp a" "δ y = Var x" by moura
      hence "Γ (Var y) = τ" using x(2) assms(1) by (simp add: wt_subst_def)
      thus ?thesis using y(1) by auto
    qed (use x in auto)
  qed

  show ?B by (metis bvars_sstp_subst)

  show ?C
  proof
    fix τ assume τ: "τ ∈ Γ ` Var ` vars_sstp (a ·sstp δ)"
    then obtain x where x: "x ∈ vars_sstp (a ·sstp δ)" "Γ (Var x) = τ" by moura
```

```
    show "τ ∈ Γ ` Var ` vars_sstp a"
    proof (cases "x ∈ vars_sstp a")
      case False
      hence "∃y ∈ vars_sstp a. δ y = Var x"
      proof (cases a)
        case (NegChecks X F G)
        hence *: "x ∈ fv_pairs (F ·pairs rm_vars (set X) δ) ∪ fv_pairs (G ·pairs rm_vars (set X) δ)"
                 "x ∉ set X"
          using vars_sstp_NegCheck[of X "F ·pairs rm_vars (set X) δ" "G ·pairs rm_vars (set X) δ"]
                vars_sstp_NegCheck[of X F G] False x(1)
          by (fastforce, blast)

        obtain y where y: "y ∈ fv_pairs F ∪ fv_pairs G" "x ∈ fv (rm_vars (set X) δ y)"
          using fv_pairs_subst_obtain_var[of _ _ "rm_vars (set X) δ"]
                fv_pairs_subst_obtain_var[of _ _ "rm_vars (set X) δ"]
                *(1)
          by blast

        have "fv (rm_vars (set X) δ z) = {} ∨ (∃u. rm_vars (set X) δ z = Var u)" for z
          using assms(2) rm_vars_img_subset[of "set X" δ] by blast
        hence "rm_vars (set X) δ y = Var x" using y(2) by fastforce
        hence "∃y ∈ vars_sstp a. rm_vars (set X) δ y = Var x"
          using y vars_sstp_NegCheck[of X F G] NegChecks by blast
        thus ?thesis by (metis (full_types) *(2) term.inject(1))
      qed (use assms(2) x(1) subst_apply_img_var'[of x _ δ] in fastforce)+
      then obtain y where y: "y ∈ vars_sstp a" "δ y = Var x" by moura
      hence "Γ (Var y) = τ" using x(2) assms(1) by (simp add: wt_subst_def)
      thus ?thesis using y(1) by auto
    qed (use x in auto)
  qed
qed

lemma (in typed_model) wt_subst_lsst_vars_type_subset:
  fixes A::"('fun,'var,'a) labeled_stateful_strand"
  assumes "wt_subst δ"
    and "∀t ∈ subst_range δ. fv t = {} ∨ (∃x. t = Var x)"
  shows "Γ ` Var ` fv_lsst (A ·lsst δ) ⊆ Γ ` Var ` fv_lsst A" (is ?A)
    and "Γ ` Var ` bvars_lsst (A ·lsst δ) = Γ ` Var ` bvars_lsst A" (is ?B)
    and "Γ ` Var ` vars_lsst (A ·lsst δ) ⊆ Γ ` Var ` vars_lsst A" (is ?C)
proof -
  have "vars_lsst (a#A ·lsst δ) = vars_sstp (b ·sstp δ) ∪ vars_lsst (A ·lsst δ)"
       "vars_lsst (a#A) = vars_sstp b ∪ vars_lsst A"
       "fv_lsst (a#A ·lsst δ) = fv_sstp (b ·sstp δ) ∪ fv_lsst (A ·lsst δ)"
       "fv_lsst (a#A) = fv_sstp b ∪ fv_lsst A"
       "bvars_lsst (a#A ·lsst δ) = set (bvars_sstp (b ·sstp δ)) ∪ bvars_lsst (A ·lsst δ)"
       "bvars_lsst (a#A) = set (bvars_sstp b) ∪ bvars_lsst A"
    when "a = (l,b)" for a l b and A::"('fun,'var,'a) labeled_stateful_strand"
    using that unlabel_Cons(1)[of l b A] unlabel_subst[of "a#A" δ]
          subst_lsst_cons[of a A δ] subst_sst_cons[of b "unlabel A" δ]
          subst_apply_labeled_stateful_strand_step.simps(1)[of l b δ]
          vars_sst_unlabel_Cons[of l b A] vars_sst_unlabel_Cons[of l "b ·sstp δ" "A ·lsst δ"]
          fv_sst_unlabel_Cons[of l b A] fv_sst_unlabel_Cons[of l "b ·sstp δ" "A ·lsst δ"]
          bvars_sst_unlabel_Cons[of l b A] bvars_sst_unlabel_Cons[of l "b ·sstp δ" "A ·lsst δ"]
    by simp_all
  hence *: "Γ ` Var ` vars_lsst (a#A ·lsst δ) =
             Γ ` Var ` vars_sstp (b ·sstp δ) ∪ Γ ` Var ` vars_lsst (A ·lsst δ)"
            "Γ ` Var ` vars_lsst (a#A) = Γ ` Var ` vars_sstp b ∪ Γ ` Var ` vars_lsst A"
            "Γ ` Var ` fv_lsst (a#A ·lsst δ) =
             Γ ` Var ` fv_sstp (b ·sstp δ) ∪ Γ ` Var ` fv_lsst (A ·lsst δ)"
            "Γ ` Var ` fv_lsst (a#A) = Γ ` Var ` fv_sstp b ∪ Γ ` Var ` fv_lsst A"
            "Γ ` Var ` bvars_lsst (a#A ·lsst δ) =
             Γ ` Var ` set (bvars_sstp (b ·sstp δ)) ∪ Γ ` Var ` bvars_lsst (A ·lsst δ)"
            "Γ ` Var ` bvars_lsst (a#A) = Γ ` Var ` set (bvars_sstp b) ∪ Γ ` Var ` bvars_lsst A"
```

```
    when "a = (l,b)" for a l b and A::"('fun,'var,'a) labeled_stateful_strand"
    using that by fast+

  have "?A ∧ ?B ∧ ?C"
  proof (induction A)
    case (Cons a A)
    obtain l b where a: "a = (l,b)" by (metis surj_pair)

    show ?case
      using Cons.IH wt_subst_sstp_vars_type_subset[OF assms, of b] *[OF a, of A]
      by (metis Un_mono)
  qed simp
  thus ?A ?B ?C by metis+
qed

lemma (in stateful_typed_model) fv_pair_fv_pairs_subset:
  assumes "d ∈ set D"
  shows "fv (pair (snd d)) ⊆ fv_pairs (unlabel D)"
using assms unfolding pair_def by (induct D) (auto simp add: unlabel_def)

lemma (in stateful_typed_model) labeled_sat_ineq_lift:
  assumes "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←dbproj i D. d ∉ set Di]⟧_d I"
    (is "?R1 D")
  and "∀(j,p) ∈ {(i,t,s)} ∪ set D ∪ set Di. ∀(k,q) ∈ {(i,t,s)} ∪ set D ∪ set Di.
          (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k" (is "?R2 D")
  shows "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←D. d ∉ set Di]⟧_d I"
using assms
proof (induction D)
  case (Cons dl D)
  obtain d l where dl: "dl = (l,d)" by (metis surj_pair)

  have 1: "?R1 D"
  proof (cases "i = l")
    case True thus ?thesis using Cons.prems(1) dl by (cases "dl ∈ set Di") auto
  next
    case False thus ?thesis using Cons.prems(1) dl by auto
  qed

  have "set D ⊆ set (dl#D)" by auto
  hence 2: "?R2 D" using Cons.prems(2) by blast

  have "i ≠ l ∨ dl ∈ set Di ∨ ⟦M; [∀X⟨∨≠: [(pair (t,s), pair (snd dl))]⟩_st]⟧_d I"
    using Cons.prems(1) dl by (auto simp add: ineq_model_def)
  moreover have "∃δ. Unifier δ (pair (t,s)) (pair d) ⟹ i = l"
    using Cons.prems(2) dl by force
  ultimately have 3: "dl ∈ set Di ∨ ⟦M; [∀X⟨∨≠: [(pair (t,s), pair (snd dl))]⟩_st]⟧_d I"
    using strand_sem_not_unif_is_sat_ineq[of "pair (t,s)" "pair d"] dl by fastforce

  show ?case using Cons.IH[OF 1 2] 3 dl by auto
qed simp

lemma (in stateful_typed_model) labeled_sat_ineq_dbproj:
  assumes "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←D. d ∉ set Di]⟧_d I"
    (is "?P D")
  shows "⟦M; map (λd. ∀X⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←dbproj i D. d ∉ set Di]⟧_d I"
    (is "?Q D")
using assms
proof (induction D)
  case (Cons di D)
  obtain d j where di: "di = (j,d)" by (metis surj_pair)

  have "?P D" using Cons.prems by (cases "di ∈ set Di") auto
  hence IH: "?Q D" by (metis Cons.IH)
```

```
  show ?case using di IH
  proof (cases "i = j ∧ di ∉ set Di")
    case True
    have 1: "⟦M; [∀X⟨∨≠: [(pair (t,s), pair (snd di))]⟩st]⟧d ℐ"
      using Cons.prems True by auto
    have 2: "dbproj i (di#D) = di#dbproj i D" using True dbproj_Cons(1) di by auto
    show ?thesis using 1 2 IH by auto
  qed auto
qed simp

lemma (in stateful_typed_model) labeled_sat_ineq_dbproj_sem_equiv:
  assumes "∀(j,p) ∈ ((λ(t, s). (i, t, s)) ` set F') ∪ set D.
           ∀(k,q) ∈ ((λ(t, s). (i, t, s)) ` set F') ∪ set D.
            (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k"
  and "fvpairs (map snd D) ∩ set X = {}"
  shows "⟦M; map (λG. ∀X⟨∨≠: (F@G)⟩st) (trpairs F' (map snd D))⟧d ℐ ⟷
         ⟦M; map (λG. ∀X⟨∨≠: (F@G)⟩st) (trpairs F' (map snd (dbproj i D)))⟧d ℐ"
proof -
  let ?A = "set (map snd D) ·pset ℐ"
  let ?B = "set (map snd (dbproj i D)) ·pset ℐ"
  let ?C = "set (map snd D) - set (map snd (dbproj i D))"
  let ?F = "(λ(t, s). (i, t, s)) ` set F'"
  let ?P = "λδ. subst_domain δ = set X ∧ ground (subst_range δ)"

  have 1: "∀(t, t') ∈ set (map snd D). (fv t ∪ fv t') ∩ set X = {}"
          "∀(t, t') ∈ set (map snd (dbproj i D)). (fv t ∪ fv t') ∩ set X = {}"
    using assms(2) dbproj_subset[of i D] unfolding unlabel_def by force+

  have 2: "?B ⊆ ?A" by auto

  have 3: "¬Unifier δ (pair f) (pair d)"
    when f: "f ∈ set F'" and d: "d ∈ set (map snd D) - set (map snd (dbproj i D))"
    for f d and δ::"('fun,'var) subst"
  proof -
    obtain k where k: "(k,d) ∈ set D - set (dbproj i D)"
      using d by force

    have "(i,f) ∈ ((λ(t, s). (i, t, s)) ` set F') ∪ set D"
         "(k,d) ∈ ((λ(t, s). (i, t, s)) ` set F') ∪ set D"
      using f k by auto
    hence "i = k" when "Unifier δ (pair f) (pair d)" for δ
      using assms(1) that by blast
    moreover have "k ≠ i" using k d by simp
    ultimately show ?thesis by metis
  qed

  have "f ·p δ ≠ d ·p δ"
    when "f ∈ set F'" "d ∈ ?C" for f d and δ::"('fun,'var) subst"
    by (metis fun_pair_eq_subst 3[OF that])
  hence "f ·p (δ ∘s ℐ) ∉ ?C ·pset (δ ∘s ℐ)"
    when "f ∈ set F'" for f and δ::"('fun,'var) subst"
    using that by blast
  moreover have "?C ·pset δ ·pset ℐ = ?C ·pset ℐ"
    when "?P δ" for δ
    using assms(2) that pairs_substI[of δ "(set (map snd D) - set (map snd (dbproj i D)))"]
    by blast
  ultimately have 4: "f ·p (δ ∘s ℐ) ∉ ?C ·pset ℐ"
    when "f ∈ set F'" "?P δ" for f and δ::"('fun,'var) subst"
    by (metis that subst_pairs_compose)

  { fix f and δ::"('fun,'var) subst"
    assume "f ∈ set F'" "?P δ"
```

```
    hence "f ·p (δ ∘s I) ∉ ?C ·pset I" by (metis 4)
    hence "f ·p (δ ∘s I) ∉ ?A - ?B" by force
  } hence 5: "∀f∈set F'. ∀δ. ?P δ ⟶ f ·p (δ ∘s I) ∉ ?A - ?B" by metis

  show ?thesis
    using negchecks_model_db_subset[OF 2]
          negchecks_model_db_supset[OF 2 5]
          tr_pairs_sem_equiv[OF 1(1)]
          tr_pairs_sem_equiv[OF 1(2)]
          tr_NegChecks_constr_iff(1)
          strand_sem_eq_defs(2)
    by (metis (no_types, lifting))
qed

lemma (in stateful_typed_model) labeled_sat_eqs_list_all:
  assumes "∀(j, p) ∈ {(i,t,s)} ∪ set D. ∀(k,q) ∈ {(i,t,s)} ∪ set D.
            (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k" (is "?P D")
  and "⟦M; map (λd. ⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st) D⟧d I" (is "?Q D")
  shows "list_all (λd. fst d = i) D"
using assms
proof (induction D rule: List.rev_induct)
  case (snoc di D)
  obtain d j where di: "di = (j,d)" by (metis surj_pair)
  have "pair (t,s) · I = pair d · I" using di snoc.prems(2) by auto
  hence "∃δ. Unifier δ (pair (t,s)) (pair d)" by auto
  hence 1: "i = j" using snoc.prems(1) di by fastforce

  have "set D ⊆ set (D@[di])" by auto
  hence 2: "?P D" using snoc.prems(1) by blast

  have 3: "?Q D" using snoc.prems(2) by auto

  show ?case using di 1 snoc.IH[OF 2 3] by simp
qed simp

lemma (in stateful_typed_model) labeled_sat_eqs_subseqs:
  assumes "Di ∈ set (subseqs D)"
  and "∀(j, p) ∈ {(i,t,s)} ∪ set D. ∀(k, q) ∈ {(i,t,s)} ∪ set D.
          (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k" (is "?P D")
  and "⟦M; map (λd. ⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st) Di⟧d I"
  shows "Di ∈ set (subseqs (dbproj i D))"
proof -
  have "set Di ⊆ set D" by (rule subseqs_subset[OF assms(1)])
  hence "?P Di" using assms(2) by blast
  thus ?thesis using labeled_sat_eqs_list_all[OF _ assms(3)] subseqs_mem_dbproj[OF assms(1)] by simp
qed

lemma (in stateful_typed_model) dual_lsst_tfr_sstp:
  assumes "list_all tfr_sstp (unlabel S)"
  shows "list_all tfr_sstp (unlabel (dual_lsst S))"
using assms
proof (induction S)
  case (Cons a S)
  have prems: "tfr_sstp (snd a)" "list_all tfr_sstp (unlabel S)"
    using Cons.prems unlabel_Cons(2)[of a S] by simp_all
  hence IH: "list_all tfr_sstp (unlabel (dual_lsst S))" by (metis Cons.IH)

  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  with Cons show ?case
  proof (cases b)
    case (Equality c t t')
    hence "dual_lsst (a#S) = a#dual_lsst S" by (metis dual_lsst_Cons(3) a)
    thus ?thesis using a IH prems by fastforce
```

```
  next
    case (NegChecks X F G)
    hence "dual_lsst (a#S) = a#dual_lsst S" by (metis dual_lsst_Cons(7) a)
    thus ?thesis using a IH prems by fastforce
  qed auto
qed simp

lemma (in stateful_typed_model) setops_sst_unlabel_dual_lsst_eq:
  "setops_sst (unlabel (dual_lsst A)) = setops_sst (unlabel A)"
proof (induction A)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)
  thus ?case using Cons.IH by (cases b) (simp_all add: setops_sst_def)
qed simp
```

6.2.2 Locale Setup and Definitions

```
locale labeled_stateful_typed_model =
  stateful_typed_model arity public Ana Γ Pair
+ labeled_typed_model arity public Ana Γ label_witness1 label_witness2
  for arity::"'fun ⇒ nat"
  and public::"'fun ⇒ bool"
  and Ana::"('fun,'var) term ⇒ (('fun,'var) term list × ('fun,'var) term list)"
  and Γ::"('fun,'var) term ⇒ ('fun,'atom::finite) term_type"
  and Pair::"'fun"
  and label_witness1::"'lbl"
  and label_witness2::"'lbl"
begin

definition lpair where
  "lpair lp ≡ case lp of (i,p) ⇒ (i,pair p)"

lemma setops_lsstp_pair_image[simp]:
  "lpair ` (setops_lsstp (i,send⟨t⟩)) = {}"
  "lpair ` (setops_lsstp (i,receive⟨t⟩)) = {}"
  "lpair ` (setops_lsstp (i,⟨ac: t ≐ t'⟩)) = {}"
  "lpair ` (setops_lsstp (i,insert⟨t,s⟩)) = {(i, pair (t,s))}"
  "lpair ` (setops_lsstp (i,delete⟨t,s⟩)) = {(i, pair (t,s))}"
  "lpair ` (setops_lsstp (i,⟨ac: t ∈ s⟩)) = {(i, pair (t,s))}"
  "lpair ` (setops_lsstp (i,∀X⟨∨≠: F ∨∉: F'⟩)) = ((λ(t,s). (i, pair (t,s))) ` set F')"
unfolding lpair_def by force+

definition par_comp_lsst where
  "par_comp_lsst (𝒜::('fun,'var,'lbl) labeled_stateful_strand) (Secrets::('fun,'var) terms) ≡
    (∀l1 l2. l1 ≠ l2 ⟶
          GSMP_disjoint (trms_sst (proj_unl l1 𝒜) ∪ pair ` setops_sst (proj_unl l1 𝒜))
                        (trms_sst (proj_unl l2 𝒜) ∪ pair ` setops_sst (proj_unl l2 𝒜)) Secrets) ∧
    ground Secrets ∧ (∀s ∈ Secrets. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Secrets) ∧
    (∀(i,p) ∈ setops_lsst 𝒜. ∀(j,q) ∈ setops_lsst 𝒜.
      (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j)"

definition declassified_lsst where
  "declassified_lsst 𝒜 ℐ ≡ {t. ⟨⋆, receive⟨t⟩⟩ ∈ set 𝒜} ·set ℐ"

definition strand_leaks_lsst ("_ leaks _ under _") where
  "(𝒜::('fun,'var,'lbl) labeled_stateful_strand) leaks Secrets under ℐ ≡
    (∃t ∈ Secrets - declassified_lsst 𝒜 ℐ. ∃n. ℐ ⊨s (proj_unl n 𝒜@[send⟨t⟩]))"

definition typing_cond_sst where
  "typing_cond_sst 𝒜 ≡ wf_sst 𝒜 ∧ wf_trms (trms_sst 𝒜) ∧ tfr_sst 𝒜"

type_synonym ('a,'b,'c) labeleddbstate = "('c strand_label × (('a,'b) term × ('a,'b) term)) set"
type_synonym ('a,'b,'c) labeleddbstatelist = "('c strand_label × (('a,'b) term × ('a,'b) term))
```

list"

For proving the compositionality theorem for stateful constraints the idea is to first define a variant of the reduction technique that was used to establish the stateful typing result. This variant performs database-state projections, and it allows us to reduce the compositionality problem for stateful constraints to ordinary constraints.

```
fun trpc::
  "('fun,'var,'lbl) labeled_stateful_strand ⇒ ('fun,'var,'lbl) labeleddbstatelist
   ⇒ ('fun,'var,'lbl) labeled_strand list"
where
  "trpc [] D = [[]]"
| "trpc ((i,send⟨t⟩)#A) D = map ((#) (i,send⟨t⟩st)) (trpc A D)"
| "trpc ((i,receive⟨t⟩)#A) D = map ((#) (i,receive⟨t⟩st)) (trpc A D)"
| "trpc ((i,⟨ac: t ≐ t'⟩)#A) D = map ((#) (i,⟨ac: t ≐ t'⟩st)) (trpc A D)"
| "trpc ((i,insert⟨t,s⟩)#A) D = trpc A (List.insert (i,(t,s)) D)"
| "trpc ((i,delete⟨t,s⟩)#A) D = (
    concat (map (λDi. map (λB. (map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩st)) Di)@
                                (map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩st))
                                     [d←dbproj i D. d ∉ set Di])@B)
                          (trpc A [d←D. d ∉ set Di]))
                (subseqs (dbproj i D))))"
| "trpc ((i,⟨ac: t ∈ s⟩)#A) D =
    concat (map (λB. map (λd. (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st)#B) (dbproj i D)) (trpc A D))"
| "trpc ((i,∀X⟨∨≠: F ∨∉: F' ⟩)#A) D =
    map ((@) (map (λG. (i,∀X⟨∨≠: (F@G)⟩st)) (trpairs F' (map snd (dbproj i D))))) (trpc A D)"
```

6.2.3 Small Lemmata

```
lemma par_complsst_nil:
  assumes "ground Sec" "∀s ∈ Sec. ∀s'∈subterms s. {} ⊢c s' ∨ s' ∈ Sec"
  shows "par_complsst [] Sec"
using assms unfolding par_complsst_def by simp

lemma par_complsst_subset:
  assumes A: "par_complsst A Sec"
    and BA: "set B ⊆ set A"
  shows "par_complsst B Sec"
proof -
  let ?L = "λn A. trmssst (proj_unl n A) ∪ pair ` setopssst (proj_unl n A)"

  have "?L n B ⊆ ?L n A" for n
    using trmssst_mono[OF proj_set_mono(2)[OF BA]] setopssst_mono[OF proj_set_mono(2)[OF BA]]
    by blast
  hence "GSMP_disjoint (?L m B) (?L n B) Sec" when nm: "m ≠ n" for n m::'lbl
    using GSMP_disjoint_subset[of "?L m A" "?L n A" Sec "?L m B" "?L n B"] A nm
    unfolding par_complsst_def by simp
  thus "par_complsst B Sec"
    using A setopslsst_mono[OF BA]
    unfolding par_complsst_def by blast
qed

lemma par_complsst_split:
  assumes "par_complsst (A@B) Sec"
  shows "par_complsst A Sec" "par_complsst B Sec"
using par_complsst_subset[OF assms] by simp_all

lemma par_complsst_proj:
  assumes "par_complsst A Sec"
  shows "par_complsst (proj n A) Sec"
using par_complsst_subset[OF assms] by simp

lemma par_complsst_duallsst:
  assumes A: "par_complsst A S"
```

```
  shows "par_comp_lsst (dual_lsst A) S"
proof (unfold par_comp_lsst_def case_prod_unfold; intro conjI)
  show "ground S" "∀s ∈ S. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ S"
    using A unfolding par_comp_lsst_def by fast+

  let ?M = "λl B. (trms_lsst (proj l B) ∪ pair ` setops_sst (proj_unl l B))"
  let ?P = "λB. ∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (?M l1 B) (?M l2 B) S"
  let ?Q = "λB. ∀p ∈ setops_lsst B. ∀q ∈ setops_lsst B.
    (∃δ. Unifier δ (pair (snd p)) (pair (snd q))) ⟶ fst p = fst q"

  have "?P A" "?Q A" using A unfolding par_comp_lsst_def case_prod_unfold by blast+
  thus "?P (dual_lsst A)" "?Q (dual_lsst A)"
    by (metis setops_sst_unlabel_dual_lsst_eq trms_sst_unlabel_dual_lsst_eq proj_dual_lsst,
        metis setops_lsst_dual_lsst_eq)
qed

lemma par_comp_lsst_subst:
  assumes A: "par_comp_lsst A S"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)" "subst_domain δ ∩ bvars_lsst A = {}"
  shows "par_comp_lsst (A ·lsst δ) S"
proof (unfold par_comp_lsst_def case_prod_unfold; intro conjI)
  show "ground S" "∀s ∈ S. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ S"
    using A unfolding par_comp_lsst_def by fast+

  let ?N = "λl B. trms_lsst (proj l B) ∪ pair ` setops_sst (proj_unl l B)"
  define M where "M ≡ λl (B::('fun,'var,'lbl) labeled_stateful_strand). ?N l B"
  let ?P = "λp q. ∃δ. Unifier δ (pair (snd p)) (pair (snd q))"
  let ?Q = "λB. ∀p ∈ setops_lsst B. ∀q ∈ setops_lsst B. ?P p q ⟶ fst p = fst q"
  let ?R = "λB. ∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (?N l1 B) (?N l2 B) S"

  have d: "bvars_lsst (proj l A) ∩ subst_domain δ = {}" for l
    using δ(3) unfolding proj_def bvars_sst_def unlabel_def by auto

  have "GSMP_disjoint (M l1 A) (M l2 A) S" when l: "l1 ≠ l2" for l1 l2
    using l A unfolding par_comp_lsst_def M_def by presburger
  moreover have "M l (A ·lsst δ) = (M l A) ·set δ" for l
    using fun_pair_subst_set[of δ "setops_sst (proj_unl l A)", symmetric]
          trms_sst_subst[OF d[of l]] setops_sst_subst[OF d[of l]] proj_subst[of l A δ]
    unfolding M_def unlabel_subst by auto
  ultimately have "GSMP_disjoint (M l1 (A ·lsst δ)) (M l2 (A ·lsst δ)) S" when l: "l1 ≠ l2" for l1 l2
    using l GSMP_wt_subst_subset[OF _ δ(1,2), of _ "M l1 A"]
          GSMP_wt_subst_subset[OF _ δ(1,2), of _ "M l2 A"]
    unfolding GSMP_disjoint_def by fastforce
  thus "?R (A ·lsst δ)" unfolding M_def by blast

  have "?Q A" using A unfolding par_comp_lsst_def by force
  thus "?Q (A ·lsst δ)" using δ(3)
  proof (induction A)
    case (Cons a A)
    obtain l b where a: "a = (l,b)" by (metis surj_pair)

    have 0: "bvars_lsst (a#A) = set (bvars_sstp (snd a)) ∪ bvars_lsst A"
      unfolding bvars_sst_def unlabel_def by simp

    have "?Q A" "subst_domain δ ∩ bvars_lsst A = {}"
      using Cons.prems 0 unfolding setops_lsst_def by auto
    hence IH: "?Q (A ·lsst δ)" using Cons.IH unfolding setops_lsst_def by blast

    have 1: "fst p = fst q"
      when p: "p ∈ setops_lsstp (a ·lsstp δ)"
        and q: "q ∈ setops_lsstp (a ·lsstp δ)"
        and pq: "?P p q"
      for p q
```

```
      using a p q pq by (cases b) auto

    have 2: "fst p = fst q"
      when p: "p ∈ setops_lsst (A ·_lsst δ)"
        and q: "q ∈ setops_lsstp (a ·_lsstp δ)"
        and pq: "?P p q"
      for p q
    proof -
      obtain p' X where p':
          "p' ∈ setops_lsst A" "fst p = fst p'"
          "X ⊆ bvars_lsst (a#A)" "snd p = snd p' ·_p rm_vars X δ"
        using setops_lsst_in_subst[OF p] 0 by blast

      obtain q' Y where q':
          "q' ∈ setops_lsstp a" "fst q = fst q'"
          "Y ⊆ bvars_lsst (a#A)" "snd q = snd q' ·_p rm_vars Y δ"
        using setops_lsstp_in_subst[OF q] 0 by blast

      have "pair (snd p) = pair (snd p') · δ"
           "pair (snd q) = pair (snd q') · δ"
        using fun_pair_subst[of "snd p'" "rm_vars X δ"] fun_pair_subst[of "snd q'" "rm_vars Y δ"]
              p'(3,4) q'(3,4) Cons.prems(2) rm_vars_apply'[of δ X] rm_vars_apply'[of δ Y]
        by fastforce+
      hence "∃δ. Unifier δ (pair (snd p')) (pair (snd q'))"
        using pq Unifier_comp' by metis
      thus ?thesis using Cons.prems p'(1,2) q'(1,2) by simp
    qed

    show ?case by (metis 1 2 IH Un_iff setops_lsst_cons subst_lsst_cons)
  qed simp
qed

lemma wf_pair_negchecks_map':
  assumes "wf_st X (unlabel A)"
  shows "wf_st X (unlabel (map (λG. (i,∀Y⟨∨≠: (F@G)⟩_st)) M@A))"
using assms by (induct M) auto

lemma wf_pair_eqs_ineqs_map':
  fixes A::"('fun,'var,'lbl) labeled_strand"
  assumes "wf_st X (unlabel A)"
          "Di ∈ set (subseqs (dbproj i D))"
          "fv_pairs (unlabel D) ⊆ X"
  shows "wf_st X (unlabel (
             (map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di)@
             (map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) [d←dbproj i D. d ∉ set Di])@A))"
proof -
  let ?f = "[d←dbproj i D. d ∉ set Di]"
  define c1 where c1: "c1 = map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di"
  define c2 where c2: "c2 = map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) ?f"
  define c3 where c3: "c3 = map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩_st) (unlabel Di)"
  define c4 where c4: "c4 = map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩_st) (unlabel ?f)"
  have ci_eqs: "c3 = unlabel c1" "c4 = unlabel c2" unfolding c1 c2 c3 c4 unlabel_def by auto
  have 1: "wf_st X (unlabel (c2@A))"
    using wf_fun_pair_ineqs_map[OF assms(1)] ci_eqs(2) unlabel_append[of c2 A] c4
    by metis
  have 2: "fv_pairs (unlabel Di) ⊆ X"
    using assms(3) subseqs_set_subset(1)[OF assms(2)]
    unfolding unlabel_def
    by fastforce
  { fix B::"('fun,'var) strand" assume "wf_st X B"
    hence "wf_st X (unlabel c1@B)" using 2 unfolding c1 unlabel_def by (induct Di) auto
  } thus ?thesis using 1 unfolding c1 c2 unlabel_def by simp
qed
```

```
lemma trms_sst_setops_sst_wt_instance_ex:
  defines "M ≡ λA. trms_lsst A ∪ pair ‘ setops_sst (unlabel A)"
  assumes B: "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·_lsstp δ ∧ wt_subst δ ∧ wf_trms (subst_range δ)"
  shows "∀t ∈ M B. ∃s ∈ M A. ∃δ. t = s · δ ∧ wt_subst δ ∧ wf_trms (subst_range δ)"
proof
  let ?P = "λδ. wt_subst δ ∧ wf_trms (subst_range δ)"

  fix t assume "t ∈ M B"
  then obtain b where b: "b ∈ set B" "t ∈ trms_sstp (snd b) ∪ pair ‘ setops_sstp (snd b)"
    unfolding M_def unfolding unlabel_def trms_sst_def setops_sst_def by auto
  then obtain a δ where a: "a ∈ set A" "b = a ·_lsstp δ" and δ: "wt_subst δ" "wf_trms (subst_range δ)"
    using B by meson

  note δ' = wt_subst_rm_vars[OF δ(1)] wf_trms_subst_rm_vars'[OF δ(2)]

  have "t ∈ M (A ·_lsst δ)"
    using b(2) a
    unfolding M_def subst_apply_labeled_stateful_strand_def unlabel_def trms_sst_def setops_sst_def
    by auto
  moreover have "∃s ∈ M A. ∃δ. t = s · δ ∧ ?P δ" when "t ∈ trms_lsst (A ·_lsst δ)"
    using trms_sst_unlabel_subst'[OF that] δ' unfolding M_def by blast
  moreover have "∃s ∈ M A. ∃δ. t = s · δ ∧ ?P δ" when t: "t ∈ pair ‘ setops_sst (unlabel A ·_sst δ)"
  proof -
    obtain p where p: "p ∈ setops_sst (unlabel A ·_sst δ)" "t = pair p" using t by blast
    then obtain q X where q: "q ∈ setops_sst (unlabel A)" "p = q ·_p rm_vars (set X) δ"
      using setops_sst_subst'[OF p(1)] by blast
    hence "t = pair q · rm_vars (set X) δ"
      using fun_pair_subst[of q "rm_vars (set X) δ"] p(2) by presburger
    thus ?thesis using δ'[of "set X"] q(1) unfolding M_def by blast
  qed
  ultimately show "∃s ∈ M A. ∃δ. t = s · δ ∧ ?P δ" unfolding M_def unlabel_subst by fast
qed

lemma setops_lsst_wt_instance_ex:
  assumes B: "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·_lsstp δ ∧ wt_subst δ ∧ wf_trms (subst_range δ)"
  shows "∀p ∈ setops_lsst B. ∃q ∈ setops_lsst A. ∃δ.
    fst p = fst q ∧ snd p = snd q ·_p δ ∧ wt_subst δ ∧ wf_trms (subst_range δ)"
proof
  let ?P = "λδ. wt_subst δ ∧ wf_trms (subst_range δ)"

  fix p assume "p ∈ setops_lsst B"
  then obtain b where b: "b ∈ set B" "p ∈ setops_lsstp b" unfolding setops_lsst_def by blast
  then obtain a δ where a: "a ∈ set A" "b = a ·_lsstp δ" and δ: "wt_subst δ" "wf_trms (subst_range δ)"
    using B by meson
  hence p: "p ∈ setops_lsst (A ·_lsst δ)"
    using b(2) unfolding setops_lsst_def subst_apply_labeled_stateful_strand_def by auto

  obtain X q where q:
      "q ∈ setops_lsst A" "fst p = fst q" "snd p = snd q ·_p rm_vars X δ"
    using setops_lsst_in_subst[OF p] by blast

  show "∃q ∈ setops_lsst A. ∃δ. fst p = fst q ∧ snd p = snd q ·_p δ ∧ ?P δ"
    using q wt_subst_rm_vars[OF δ(1)] wf_trms_subst_rm_vars'[OF δ(2)] by blast
qed
```

6.2.4 Lemmata: Properties of the Constraint Translation Function

```
lemma tr_par_labeled_rcv_iff:
  "B ∈ set (tr_pc A D) ⟹ (i, receive⟨t⟩_st) ∈ set B ⟷ (i, receive⟨t⟩) ∈ set A"
by (induct A D arbitrary: B rule: tr_pc.induct) auto

lemma tr_par_declassified_eq:
```

```
  "B ∈ set (tr_pc A D) ⟹ declassified_lst B I = declassified_lsst A I"
using tr_par_labeled_rcv_iff unfolding declassified_lst_def declassified_lsst_def by simp

lemma tr_par_ik_eq:
  assumes "B ∈ set (tr_pc A D)"
  shows "ik_st (unlabel B) = ik_sst (unlabel A)"
proof -
  have "{t. ∃i. (i, receive⟨t⟩_st) ∈ set B} = {t. ∃i. (i, receive⟨t⟩) ∈ set A}"
    using tr_par_labeled_rcv_iff[OF assms] by simp
  moreover have
      "⋀C. {t. ∃i. (i, receive⟨t⟩_st) ∈ set C} = {t. receive⟨t⟩_st ∈ set (unlabel C)}"
      "⋀C. {t. ∃i. (i, receive⟨t⟩) ∈ set C} = {t. receive⟨t⟩ ∈ set (unlabel C)}"
    unfolding unlabel_def by force+
  ultimately show ?thesis by (metis ik_sst_def ik_st_is_rcv_set)
qed

lemma tr_par_deduct_iff:
  assumes "B ∈ set (tr_pc A D)"
  shows "ik_st (unlabel B) ·_set I ⊢ t ⟷ ik_sst (unlabel A) ·_set I ⊢ t"
using tr_par_ik_eq[OF assms] by metis

lemma tr_par_vars_subset:
  assumes "A' ∈ set (tr_pc A D)"
  shows "fv_lst A' ⊆ fv_sst (unlabel A) ∪ fv_pairs (unlabel D)" (is ?P)
  and "bvars_lst A' ⊆ bvars_sst (unlabel A)" (is ?Q)
proof -
  show ?P using assms
  proof (induction "unlabel A" arbitrary: A A' D rule: strand_sem_stateful_induct)
    case (ConsIn A' D ac t s AA A A')
    then obtain i B where iB: "A = (i,⟨ac: t ∈ s⟩)#B" "AA = unlabel B"
      unfolding unlabel_def by moura
    then obtain A'' d where *:
        "d ∈ set (dbproj i D)"
        "A' = (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#A''"
        "A'' ∈ set (tr_pc B D)"
      using ConsIn.prems(1) by moura
    hence "fv_lst A'' ⊆ fv_sst (unlabel B) ∪ fv_pairs (unlabel D)"
          "fv (pair (snd d)) ⊆ fv_pairs (unlabel D)"
      apply (metis ConsIn.hyps(1)[OF iB(2)])
      using fv_pairs_mono[OF dbproj_subset[of i D]]
            fv_pair_fv_pairs_subset[OF *(1)]
      by blast
    thus ?case using * iB unfolding pair_def by auto
  next
    case (ConsDel A' D t s AA A A')
    then obtain i B where iB: "A = (i,delete⟨t,s⟩)#B" "AA = unlabel B"
      unfolding unlabel_def by moura

    define fltD1 where "fltD1 = (λDi. filter (λd. d ∉ set Di) D)"
    define fltD2 where "fltD2 = (λDi. filter (λd. d ∉ set Di) (dbproj i D))"
    define constr where "constr =
      (λDi. (map (λd. (i, ⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di)@
            (map (λd. (i, ∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) (fltD2 Di)))"

    from iB obtain A'' Di where *:
        "Di ∈ set (subseqs (dbproj i D))" "A' = (constr Di)@A''" "A'' ∈ set (tr_pc B (fltD1 Di))"
      using ConsDel.prems(1) unfolding constr_def fltD1_def fltD2_def by moura
    hence "fv_lst A'' ⊆ fv_sst AA ∪ fv_pairs (unlabel (fltD1 Di))"
      unfolding constr_def fltD1_def by (metis ConsDel.hyps(1) iB(2))
    hence 1: "fv_lst A'' ⊆ fv_sst AA ∪ fv_pairs (unlabel D)"
      using fv_pairs_mono[of "unlabel (fltD1 Di)" "unlabel D"]
      unfolding unlabel_def fltD1_def by force
```

```
    have 2: "fv_pairs (unlabel Di) ∪ fv_pairs (unlabel (fltD1 Di)) ⊆ fv_pairs (unlabel D)"
      using subseqs_set_subset(1)[OF *(1)]
      unfolding fltD1_def unlabel_def
      by auto

    have 5: "fv_lst A' = fv_lst (constr Di) ∪ fv_lst A''" using * unfolding unlabel_def by force

    have "fv_lst (constr Di) ⊆ fv t ∪ fv s ∪ fv_pairs (unlabel Di) ∪ fv_pairs (unlabel (fltD1 Di))"
      unfolding unlabel_def constr_def fltD1_def fltD2_def pair_def by auto
    hence 3: "fv_lst (constr Di) ⊆ fv t ∪ fv s ∪ fv_pairs (unlabel D)" using 2 by blast

    have 4: "fv_sst (unlabel A) = fv t ∪ fv s ∪ fv_sst AA" using iB by auto

    have "fv_st (unlabel A') ⊆ fv_sst (unlabel A) ∪ fv_pairs (unlabel D)" using 1 3 4 5 by blast
    thus ?case by metis
  next
    case (ConsNegChecks A' D X F F' AA A A')
    then obtain i B where iB: "A = (i,NegChecks X F F')#B" "AA = unlabel B"
      unfolding unlabel_def by moura

    define D' where "D' ≡ ⋃(fv_pairs ` set (tr_pairs F' (unlabel (dbproj i D))))"
    define constr where "constr = map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd (dbproj i D)))"

    from iB obtain A'' where *: "A'' ∈ set (tr_pc B D)" "A' = constr@A''"
      using ConsNegChecks.prems(1) unfolding constr_def by moura
    hence "fv_lst A'' ⊆ fv_sst AA ∪ fv_pairs (unlabel D)"
      by (metis ConsNegChecks.hyps(1) iB(2))
    hence **: "fv_lst A'' ⊆ fv_sst AA ∪ fv_pairs (unlabel D)" by auto

    have 1: "fv_lst constr ⊆ (D' ∪ fv_pairs F) - set X"
      unfolding D'_def constr_def unlabel_def by auto

    have "set (unlabel (dbproj i D)) ⊆ set (unlabel D)" unfolding unlabel_def by auto
    hence 2: "D' ⊆ fv_pairs F' ∪ fv_pairs (unlabel D)"
      using tr_pairs_vars_subset'[of F' "unlabel (dbproj i D)"] fv_pairs_mono
      unfolding D'_def by blast

    have 3: "fv_lst A' ⊆ ((fv_pairs F' ∪ fv_pairs F) - set X) ∪ fv_pairs (unlabel D) ∪ fv_lst A''"
      using 1 2 *(2) unfolding unlabel_def by fastforce

    have 4: "fv_sst AA ⊆ fv_sst (unlabel A)" by (metis ConsNegChecks.hyps(2) fv_sst_cons_subset)

    have 5: "fv_pairs F' ∪ fv_pairs F - set X ⊆ fv_sst (unlabel A)"
      using ConsNegChecks.hyps(2) unfolding unlabel_def by force

    show ?case using ** 3 4 5 by blast
  qed (fastforce simp add: unlabel_def)+

  show ?Q using assms
    apply (induct "unlabel A" arbitrary: A A' D rule: strand_sem_stateful_induct)
    by (fastforce simp add: unlabel_def)+
qed

lemma tr_par_vars_disj:
  assumes "A' ∈ set (tr_pc A D)" "fv_pairs (unlabel D) ∩ bvars_sst (unlabel A) = {}"
  and "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}"
  shows "fv_lst A' ∩ bvars_lst A' = {}"
using assms tr_par_vars_subset by fast

lemma tr_par_trms_subset:
  assumes "A' ∈ set (tr_pc A D)"
  shows "trms_lst A' ⊆ trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A) ∪ pair ` snd ` set D"
using assms
```

```
proof (induction A D arbitrary: A' rule: trpc.induct)
  case 1 thus ?case by simp
next
  case (2 i t A D)
  then obtain A'' where A'': "A' = (i,send⟨t⟩st)#A''" "A'' ∈ set (trpc A D)" by moura
  hence "trmslst A'' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D"
    by (metis "2.IH")
  thus ?case using A'' by (auto simp add: setopssst_def)
next
  case (3 i t A D)
  then obtain A'' where A'': "A' = (i,receive⟨t⟩st)#A''" "A'' ∈ set (trpc A D)"
    by moura
  hence "trmslst A'' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D"
    by (metis "3.IH")
  thus ?case using A'' by (auto simp add: setopssst_def)
next
  case (4 i ac t t' A D)
  then obtain A'' where A'': "A' = (i,⟨ac: t ≐ t'⟩st)#A''" "A'' ∈ set (trpc A D)"
    by moura
  hence "trmslst A'' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D"
    by (metis "4.IH")
  thus ?case using A'' by (auto simp add: setopssst_def)
next
  case (5 i t s A D)
  hence "A' ∈ set (trpc A (List.insert (i,t,s) D))" by simp
  hence "trmslst A' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪
                    pair ` snd ` set (List.insert (i,t,s) D)"
    by (metis "5.IH")
  thus ?case by (auto simp add: setopssst_def)
next
  case (6 i t s A D)
  from 6 obtain Di A'' B C where A'':
      "Di ∈ set (subseqs (dbproj i D))" "A'' ∈ set (trpc A [d←D. d ∉ set Di])" "A' = (B@C)@A''"
      "B = map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩st)) Di"
      "C = map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩st)) [d←dbproj i D. d ∉ set Di]"
    by moura
  hence "trmslst A'' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪
                     pair ` snd ` set [d←D. d ∉ set Di]"
    by (metis "6.IH")
  moreover have "set [d←D. d ∉ set Di] ⊆ set D" using set_filter by auto
  ultimately have
      "trmslst A'' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D"
    by blast
  hence "trmslst A'' ⊆ trmssst (unlabel ((i,delete⟨t,s⟩)#A)) ∪
                      pair ` setopssst (unlabel ((i,delete⟨t,s⟩)#A)) ∪
                      pair ` snd ` set D"
    using setopssst_cons_subset trmssst_cons
    by (auto simp add: setopssst_def)
  moreover have "set Di ⊆ set D" "set [d←dbproj i D . d ∉ set Di] ⊆ set D"
    using subseqs_set_subset[OF A''(1)] by auto
  hence "trmsst (unlabel B) ⊆ insert (pair (t, s)) (pair ` snd ` set D)"
        "trmsst (unlabel C) ⊆ insert (pair (t, s)) (pair ` snd ` set D)"
    using A''(4,5) unfolding unlabel_def by auto
  hence "trmsst (unlabel (B@C)) ⊆ insert (pair (t,s)) (pair ` snd ` set D)"
    using unlabel_append[of B C] by auto
  moreover have "pair (t,s) ∈ pair ` setopssst (delete⟨t,s⟩#unlabel A)" by (simp add: setopssst_def)
  ultimately show ?case
    using A''(3) trmsst_append[of "unlabel (B@C)" "unlabel A'"] unlabel_append[of "B@C" A'']
    by (auto simp add: setopssst_def)
next
  case (7 i ac t s A D)
  from 7 obtain d A'' where A'':
      "d ∈ set (dbproj i D)" "A'' ∈ set (trpc A D)"
```

```
      "A' = (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#A''"
    by moura
  hence "trms_lst A'' ⊆ trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A) ∪
                        pair ` snd ` set D"
    by (metis "7.IH")
  moreover have "trms_st (unlabel A') = {pair (t,s), pair (snd d)} ∪ trms_st (unlabel A'')"
    using A''(1,3) by auto
  ultimately show ?case using A''(1) by (auto simp add: setops_sst_def)
next
  case (8 i X F F' A D)
  define constr where "constr = map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd (dbproj i D)))"
  define B where "B ≡ ⋃(trms_pairs ` set (tr_pairs F' (map snd (dbproj i D))))"

  from 8 obtain A'' where A'':
      "A'' ∈ set (tr_pc A D)" "A' = constr@A''"
    unfolding constr_def by moura

  have "trms_st (unlabel A'') ⊆ trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A) ∪ pair`snd`set D"
    by (metis A''(1) "8.IH")
  moreover have "trms_st (unlabel constr) ⊆ B ∪ trms_pairs F ∪ pair ` snd ` set D"
    unfolding unlabel_def constr_def B_def by auto
  ultimately have "trms_st (unlabel A') ⊆ B ∪ trms_pairs F ∪ trms_sst (unlabel A) ∪
                                          pair ` setops_sst (unlabel A) ∪ pair ` snd ` set D"
    using A'' unlabel_append[of constr A''] by auto
  moreover have "set (dbproj i D) ⊆ set D" by auto
  hence "B ⊆ pair ` set F' ∪ pair ` snd ` set D"
    using tr_pairs_trms_subset'[of F' "map snd (dbproj i D)"]
    unfolding B_def by force
  moreover have
      "pair ` setops_sst (unlabel ((i, ∀X⟨∨≠: F ∨∉: F'⟩)#A)) =
       pair ` set F' ∪ pair ` setops_sst (unlabel A)"
    by auto
  ultimately show ?case by (auto simp add: setops_sst_def)
qed

lemma tr_par_wf_trms:
  assumes "A' ∈ set (tr_pc A [])" "wf_trms (trms_sst (unlabel A))"
  shows "wf_trms (trms_lst A')"
using tr_par_trms_subset[OF assms(1)] setops_sst_wf_trms(2)[OF assms(2)]
by auto

lemma tr_par_wf':
  assumes "fv_pairs (unlabel D) ∩ bvars_sst (unlabel A) = {}"
  and "fv_pairs (unlabel D) ⊆ X"
  and "wf'_sst X (unlabel A)" "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}"
  and "A' ∈ set (tr_pc A D)"
  shows "wf_lst X A'"
proof -
  define P where
    "P = (λ(D::('fun,'var,'lbl) labeleddbstatelist) (A::('fun,'var,'lbl) labeled_stateful_strand).
          (fv_pairs (unlabel D) ∩ bvars_sst (unlabel A) = {}) ∧
          fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {})"

  have "P D A" using assms(1,4) by (simp add: P_def)
  with assms(5,3,2) show ?thesis
  proof (induction A arbitrary: X A' D)
    case Nil thus ?case by simp
  next
    case (Cons a A)
    obtain i s where i: "a = (i,s)" by (metis surj_pair)
    note prems = Cons.prems
    note IH = Cons.IH
    show ?case
```

```
proof (cases s)
  case (Receive t)
  note si = Receive i
  then obtain A'' where A'': "A' = (i,receive⟨t⟩st)#A''" "A'' ∈ set (trpc A D)" "fv t ⊆ X"
    using prems unlabel_Cons(1)[of i s A] by moura
  have *: "wf'sst X (unlabel A)"
          "fvpairs (unlabel D) ⊆ X"
          "P D A"
    using prems si apply (force, force)
    using prems(4) si unfolding P_def by fastforce
  show ?thesis using IH[OF A''(2) *] A''(1,3) by simp
next
  case (Send t)
  note si = Send i
  then obtain A'' where A'': "A' = (i,send⟨t⟩st)#A''" "A'' ∈ set (trpc A D)"
    using prems by moura
  have *: "wf'sst (X ∪ fv t) (unlabel A)"
          "fvpairs (unlabel D) ⊆ X ∪ fv t"
          "P D A"
    using prems si apply (force, force)
    using prems(4) si unfolding P_def by fastforce
  show ?thesis using IH[OF A''(2) *] A''(1) by simp
next
  case (Equality ac t t')
  note si = Equality i
  then obtain A'' where A'':
      "A' = (i,⟨ac: t ≐ t'⟩st)#A''" "A'' ∈ set (trpc A D)"
      "ac = Assign ⟹ fv t' ⊆ X"
    using prems unlabel_Cons(1)[of i s] by moura
  have *: "ac = Assign ⟹ wf'sst (X ∪ fv t) (unlabel A)"
          "ac = Check ⟹ wf'sst X (unlabel A)"
          "ac = Assign ⟹ fvpairs (unlabel D) ⊆ X ∪ fv t"
          "ac = Check ⟹ fvpairs (unlabel D) ⊆ X"
          "P D A"
    using prems si apply (force, force, force, force)
    using prems(4) si unfolding P_def by fastforce
  show ?thesis
    using IH[OF A''(2) *(1,3,5)] IH[OF A''(2) *(2,4,5)] A''(1,3)
    by (cases ac) simp_all
next
  case (Insert t t')
  note si = Insert i
  hence A': "A' ∈ set (trpc A (List.insert (i,t,t') D))" "fv t ⊆ X" "fv t' ⊆ X"
    using prems by auto
  have *: "wf'sst X (unlabel A)" "fvpairs (unlabel (List.insert (i,t,t') D)) ⊆ X"
    using prems si by (auto simp add: unlabel_def)
  have **: "P (List.insert (i,t,t') D) A"
    using prems(4) si
    unfolding P_def unlabel_def
    by fastforce
  show ?thesis using IH[OF A'(1) * **] A'(2,3) by simp
next
  case (Delete t t')
  note si = Delete i
  define constr where "constr = (λDi.
    (map (λd. (i,⟨check: (pair (t,t')) ≐ (pair (snd d))⟩st)) Di)@
    (map (λd. (i,∀ []⟨∨≠: [(pair (t,t'), pair (snd d))]⟩st)) [d←dbproj i D. d ∉ set Di]))"
  from prems si obtain Di A'' where A'':
      "A' = constr Di@A''" "A'' ∈ set (trpc A [d←D. d ∉ set Di])"
      "Di ∈ set (subseqs (dbproj i D))"
    unfolding constr_def by auto
  have *: "wf'sst X (unlabel A)"
          "fvpairs (unlabel (filter (λd. d ∉ set Di) D)) ⊆ X"
```

```
        using prems si apply simp
        using prems si by (fastforce simp add: unlabel_def)

      have "fv_pairs (unlabel (filter (λd. d ∉ set Di) D)) ⊆ fv_pairs (unlabel D)"
        by (auto simp add: unlabel_def)
      hence **: "P [d←D. d ∉ set Di] A"
        using prems si unfolding P_def
        by fastforce

      have ***: "fv_pairs (unlabel D) ⊆ X" using prems si by auto
      show ?thesis
        using IH[OF A''(2) * **] A''(1) wf_pair_eqs_ineqs_map'[OF _ A''(3) ***]
        unfolding constr_def by simp
    next
      case (InSet ac t t')
      note si = InSet i
      then obtain d A'' where A'':
          "A' = (i,⟨ac: (pair (t,t')) ≐ (pair (snd d))⟩_st)#A''"
          "A'' ∈ set (tr_pc A D)"
          "d ∈ set D"
        using prems by moura
      have *:
          "ac = Assign ⟹ wf'_sst (X ∪ fv t ∪ fv t') (unlabel A)"
          "ac = Check ⟹ wf'_sst X (unlabel A)"
          "ac = Assign ⟹ fv_pairs (unlabel D) ⊆ X ∪ fv t ∪ fv t'"
          "ac = Check ⟹ fv_pairs (unlabel D) ⊆ X"
          "P D A"
        using prems si apply (force, force, force, force)
        using prems(4) si unfolding P_def by fastforce
      have **: "fv (pair (snd d)) ⊆ X"
        using A''(3) prems(3) fv_pair_fv_pairs_subset
        by fast
      have ***: "fv (pair (t,t')) = fv t ∪ fv t'" unfolding pair_def by auto
      show ?thesis
        using IH[OF A''(2) *(1,3,5)] IH[OF A''(2) *(2,4,5)] A''(1) ** ***
        by (cases ac) (simp_all add: Un_assoc)
    next
      case (NegChecks Y F F')
      note si = NegChecks i
      then obtain A'' where A'':
          "A' = (map (λG. (i,∀Y⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd (dbproj i D))))@A''"
          "A'' ∈ set (tr_pc A D)"
        using prems by moura

      have *: "wf'_sst X (unlabel A)" "fv_pairs (unlabel D) ⊆ X" using prems si by auto

      have "bvars_sst (unlabel A) ⊆ bvars_sst (unlabel ((i,∀Y⟨∨≠: F ∨∉: F'⟩)#A))"
           "fv_sst (unlabel A) ⊆ fv_sst (unlabel ((i,∀Y⟨∨≠: F ∨∉: F'⟩)#A))"
        by auto
      hence **:  "P D A" using prems si unfolding P_def by blast

      show ?thesis using IH[OF A''(2) * **] A''(1) wf_pair_negchecks_map' by simp
    qed
  qed
qed

lemma tr_par_wf:
  assumes "A' ∈ set (tr_pc A [])"
    and "wf_sst (unlabel A)"
    and "wf_trms (trms_lsst A)"
  shows "wf_lst {} A'"
    and "wf_trms (trms_lst A')"
    and "fv_lst A' ∩ bvars_lst A' = {}"
```

```
using tr_par_wf'[OF _ _ _ _ assms(1)]
      tr_par_wf_trms[OF assms(1,3)]
      tr_par_vars_disj[OF assms(1)]
      assms(2)
by fastforce+

lemma tr_par_tfr_sstp:
  assumes "A' ∈ set (tr_pc A D)" "list_all tfr_sstp (unlabel A)"
  and "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}" (is "?P0 A D")
  and "fv_pairs (unlabel D) ∩ bvars_sst (unlabel A) = {}" (is "?P1 A D")
  and "∀t ∈ pair ` setops_sst (unlabel A) ∪ pair ` snd ` set D.
       ∀t' ∈ pair ` setops_sst (unlabel A) ∪ pair ` snd ` set D.
          (∃δ. Unifier δ t t') ⟶ Γ t = Γ t'" (is "?P3 A D")
  shows "list_all tfr_stp (unlabel A')"
proof -
  have sublmm: "list_all tfr_sstp (unlabel A)" "?P0 A D" "?P1 A D" "?P3 A D"
    when p: "list_all tfr_sstp (unlabel (a#A))" "?P0 (a#A) D" "?P1 (a#A) D" "?P3 (a#A) D"
    for a A D
  proof -
    show "list_all tfr_sstp (unlabel A)" using p(1) by (simp add: unlabel_def tfr_sst_def)
    show "?P0 A D" using p(2) fv_sst_cons_subset unfolding unlabel_def by fastforce
    show "?P1 A D" using p(3) bvars_sst_cons_subset unfolding unlabel_def by fastforce
    have "setops_sst (unlabel A) ⊆ setops_sst (unlabel (a#A))"
      using setops_sst_cons_subset unfolding unlabel_def by auto
    thus "?P3 A D" using p(4) by blast
  qed

  show ?thesis using assms
  proof (induction A D arbitrary: A' rule: tr_pc.induct)
    case 1 thus ?case by simp
  next
    case (2 i t A D)
    note prems = "2.prems"
    note IH = "2.IH"
    from prems(1) obtain A'' where A'': "A' = (i,send⟨t⟩_st)#A''" "A'' ∈ set (tr_pc A D)" by moura
    have "list_all tfr_stp (unlabel A'')"
      using IH[OF A''(2)] prems(5) sublmm[OF prems(2,3,4,5)]
      by meson
    thus ?case using A''(1) by simp
  next
    case (3 i t A D)
    note prems = "3.prems"
    note IH = "3.IH"
    from prems(1) obtain A'' where A'': "A' = (i,receive⟨t⟩_st)#A''" "A'' ∈ set (tr_pc A D)" by moura
    have "list_all tfr_stp (unlabel A'')"
      using IH[OF A''(2)] prems(5) sublmm[OF prems(2,3,4,5)]
      by meson
    thus ?case using A''(1) by simp
  next
    case (4 i ac t t' A D)
    note prems = "4.prems"
    note IH = "4.IH"
    from prems(1) obtain A'' where A'': "A' = (i,⟨ac: t ≐ t'⟩_st)#A''" "A'' ∈ set (tr_pc A D)" by
moura
    have "list_all tfr_stp (unlabel A'')"
      using IH[OF A''(2)] prems(5) sublmm[OF prems(2,3,4,5)]
      by meson
    thus ?case using A''(1) prems(2) by simp
  next
    case (5 i t s A D)
    note prems = "5.prems"
    note IH = "5.IH"
    from prems(1) have A': "A' ∈ set (tr_pc A (List.insert (i,t,s) D))" by simp
```

```
    have 1: "list_all tfr_sstp (unlabel A)" using sublmm[OF prems(2,3,4,5)] by simp

    have "pair ‘ setops_sst (unlabel ((i,insert⟨t,s⟩)#A)) ∪ pair‘snd‘set D =
          pair ‘ setops_sst (unlabel A) ∪ pair‘snd‘set (List.insert (i,t,s) D)"
      by (auto simp add: setops_sst_def)
    hence 3: "?P3 A (List.insert (i,t,s) D)" using prems(5) by metis
    moreover have "?P1 A (List.insert (i,t,s) D)"
      using prems(3,4) bvars_sst_cons_subset[of "unlabel A" "insert⟨t,s⟩"]
      unfolding unlabel_def
      by fastforce
    ultimately have "list_all tfr_stp (unlabel A')"
      using IH[OF A' sublmm(1,2)[OF prems(2,3,4,5)] _ 3] by metis
    thus ?case using A'(1) by auto
  next
    case (6 i t s A D)
    note prems = "6.prems"
    note IH = "6.IH"

    define constr where constr: "constr ≡ (λDi.
      (map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di)@
      (map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) (filter (λd. d ∉ set Di) (dbproj i
D))))"

    from prems(1) obtain Di A'' where A'':
        "A' = constr Di@A''" "A'' ∈ set (tr_pc A (filter (λd. d ∉ set Di) D))"
        "Di ∈ set (subseqs (dbproj i D))"
      unfolding constr by fastforce

    define Q1 where "Q1 ≡ (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
        ∀x ∈ (fv_pairs F) - set X. ∃a. Γ (Var x) = TAtom a)"
    define Q2 where "Q2 ≡ (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
        ∀f T. Fun f T ∈ subterms_set (trms_pairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ‘ set X))"

    have "pair ‘ setops_sst (unlabel A) ∪ pair‘snd‘set [d←D. d ∉ set Di]
            ⊆ pair ‘ setops_sst (unlabel ((i,delete⟨t,s⟩)#A)) ∪ pair‘snd‘set D"
      using subseqs_set_subset[OF A''(3)] by (force simp add: setops_sst_def)
    moreover have "∀a∈M. ∀b∈M. P a b"
      when "M ⊆ N" "∀a∈N. ∀b∈N. P a b"
      for M N::"('fun, 'var) terms" and  P
      using that by blast
    ultimately have *: "?P3 A (filter (λd. d ∉ set Di) D)"
      using prems(5) by presburger

    have **: "?P1 A (filter (λd. d ∉ set Di) D)"
      using prems(4) bvars_sst_cons_subset[of "unlabel A" "delete⟨t,s⟩"]
      unfolding unlabel_def by fastforce

    have 1: "list_all tfr_stp (unlabel A'')"
      using IH[OF A''(3,2) sublmm(1,2)[OF prems(2,3,4,5)] ** *]
      by metis

    have 2: "⟨ac: u ≐ u'⟩_st ∈ set (unlabel A'') ∨
            (∃d ∈ set Di. u = pair (t,s) ∧ u' = pair (snd d))"
      when "⟨ac: u ≐ u'⟩_st ∈ set (unlabel A')" for ac u u'
      using that A''(1) unfolding constr unlabel_def by force
    have 3:
        "∀X⟨∨≠: u⟩_st ∈ set (unlabel A'') ∨
         (∃d ∈ set (filter (λd. d ∉ set Di) D). u = [(pair (t,s), pair (snd d))] ∧ Q2 u X)"
      when "∀X⟨∨≠: u⟩_st ∈ set (unlabel A')" for X u
      using that A''(1) unfolding Q2_def constr unlabel_def by force
    have 4: "∀d∈set D. (∃δ. Unifier δ (pair (t,s)) (pair (snd d)))
                    ⟶ Γ (pair (t,s)) = Γ (pair (snd d))"
```

```
      using prems(5) by (simp add: setops_sst_def)

    { fix ac u u'
      assume a: "⟨ac: u ≐ u'⟩_st ∈ set (unlabel A')" "∃δ. Unifier δ u u'"
      hence "⟨ac: u ≐ u'⟩_st ∈ set (unlabel A'') ∨ (∃d ∈ set Di. u = pair (t,s) ∧ u' = pair (snd
d))"
        using 2 by metis
      moreover {
        assume "⟨ac: u ≐ u'⟩_st ∈ set (unlabel A'')"
        hence "tfr_stp (⟨ac: u ≐ u'⟩_st)"
          using 1 Ball_set_list_all[of "unlabel A''" tfr_stp]
          by fast
      } moreover {
        fix d assume "d ∈ set Di" "u = pair (t,s)" "u' = pair (snd d)"
        hence "∃δ. Unifier δ u u' ⟹ Γ u = Γ u'"
          using 4 dbproj_subseq_subset A''(3)
          by fast
        hence "tfr_stp (⟨ac: u ≐ u'⟩_st)"
          using Ball_set_list_all[of "unlabel A''" tfr_stp]
          by simp
        hence "Γ u = Γ u'" using tfr_stp_list_all_alt_def[of "unlabel A''"]
          using a(2) unfolding unlabel_def by auto
      } ultimately have "Γ u = Γ u'"
          using tfr_stp_list_all_alt_def[of "unlabel A''"] a(2)
          unfolding unlabel_def by auto
    } moreover {
      fix u U
      assume "∀U⟨∨≠: u⟩_st ∈ set (unlabel A')"
      hence "∀U⟨∨≠: u⟩_st ∈ set (unlabel A'') ∨
             (∃d ∈ set (filter (λd. d ∉ set Di) D). u = [(pair (t,s), pair (snd d))] ∧ Q2 u U)"
        using 3 by metis
      hence "Q1 u U ∨ Q2 u U"
        using 1 4 subseqs_set_subset[OF A''(3)] tfr_stp_list_all_alt_def[of "unlabel A''"]
        unfolding Q1_def Q2_def
        by blast
    } ultimately show ?case
      using tfr_stp_list_all_alt_def[of "unlabel A'"] unfolding Q1_def Q2_def unlabel_def by blast
  next
    case (7 i ac t s A D)
    note prems = "7.prems"
    note IH = "7.IH"

    from prems(1) obtain d A'' where A'':
        "A' = (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#A''"
        "A'' ∈ set (tr_pc A D)"
        "d ∈ set (dbproj i D)"
      by moura

    have 1: "list_all tfr_stp (unlabel A'')"
      using IH[OF A''(2) sublmm(1,2,3)[OF prems(2,3,4,5)] sublmm(4)[OF prems(2,3,4,5)]]
      by metis

    have 2: "Γ (pair (t,s)) = Γ (pair (snd d))"
      when "∃δ. Unifier δ (pair (t,s)) (pair (snd d))"
      using that prems(2,5) A''(3) unfolding tfr_sst_def by (simp add: setops_sst_def)

    show ?case using A''(1) 1 2 by fastforce
  next
    case (8 i X F F' A D)
    note prems = "8.prems"
    note IH = "8.IH"

    define constr where
```

```
      "constr = map (λG. (i,∀X⟨∨≠: (F@G)⟩st)) (trpairs F' (map snd (dbproj i D)))"

    define Q1 where "Q1 ≡ (λ(F::(('fun,'var) term × ('fun,'var) term) list) X.
        ∀x ∈ (fvpairs F) - set X. ∃a. Γ (Var x) = TAtom a)"

    define Q2 where "Q2 ≡ (λ(M::('fun,'var) terms) X.
        ∀f T. Fun f T ∈ subtermsset M ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))"

    have Q2_subset: "Q2 M' X" when "M' ⊆ M" "Q2 M X" for X M M'
      using that unfolding Q2_def by auto

    have Q2_supset: "Q2 (M ∪ M') X" when "Q2 M X" "Q2 M' X" for X M M'
      using that unfolding Q2_def by auto

    from prems obtain A'' where A'': "A' = constr@A''" "A'' ∈ set (trpc A D)"
      using constr_def by moura

    have 0: "constr = [(i,∀X⟨∨≠: F⟩st)]" when "F' = []" using that unfolding constr_def by simp

    have 1: "list_all tfrstp (unlabel A'')"
      using IH[OF A''(2) sublmm(1,2,3)[OF prems(2,3,4,5)] sublmm(4)[OF prems(2,3,4,5)]]
      by metis

    have 2: "(F' = [] ∧ Q1 F X) ∨ Q2 (trmspairs F ∪ pair ` set F') X"
      using prems(2) unfolding Q1_def Q2_def by simp

    have 3: "F' = [] ⟹ Q1 F X ⟹ list_all tfrstp (unlabel constr)"
      using 0 2 tfrstp_list_all_alt_def[of "unlabel constr"] unfolding Q1_def by auto

    { fix c assume "c ∈ set (unlabel constr)"
      hence "∃G ∈ set (trpairs F' (map snd (dbproj i D))). c = ∀X⟨∨≠: (F@G)⟩st"
        unfolding constr_def unlabel_def by force
    } moreover {
      fix G
      assume G: "G ∈ set (trpairs F' (map snd (dbproj i D)))"
         and c: "∀X⟨∨≠: (F@G)⟩st ∈ set (unlabel constr)"
         and e: "Q2 (trmspairs F ∪ pair ` set F') X"

      have d_Q2: "Q2 (pair ` set (map snd D)) X" unfolding Q2_def
      proof (intro allI impI)
        fix f T assume "Fun f T ∈ subtermsset (pair ` set (map snd D))"
        then obtain d where d: "d ∈ set (map snd D)" "Fun f T ∈ subterms (pair d)" by force
        hence "fv (pair d) ∩ set X = {}"
          using prems(4) unfolding pair_def by (force simp add: unlabel_def)
        thus "T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)"
          by (metis fv_disj_Fun_subterm_param_cases d(2))
      qed

      have "trmspairs (F@G) ⊆ trmspairs F ∪ pair ` set F' ∪ pair ` set (map snd D)"
        using trpairs_trms_subset[OF G] by force
      hence "Q2 (trmspairs (F@G)) X" using Q2_subset[OF _ Q2_supset[OF e d_Q2]] by metis
      hence "tfrstp (∀X⟨∨≠: (F@G)⟩st)" by (metis Q2_def tfrstp.simps(2))
    } ultimately have 4:
        "Q2 (trmspairs F ∪ pair ` set F') X ⟹ list_all tfrstp (unlabel constr)"
      using Ball_set by blast

    have 5: "list_all tfrstp (unlabel constr)" using 2 3 4 by metis

    show ?case using 1 5 A''(1) by (simp add: unlabel_def)
  qed
qed

lemma tr_par_tfr:
```

```
  assumes "A' ∈ set (tr_pc A [])" and "tfr_sst (unlabel A)"
    and "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}"
  shows "tfr_st (unlabel A')"
proof -
  have *: "trms_lst A' ⊆ trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A)"
    using tr_par_trms_subset[OF assms(1)] by simp
  hence "SMP (trms_lst A') ⊆ SMP (trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A))"
    using SMP_mono by simp
  moreover have "tfr_set (trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A))"
    using assms(2) unfolding tfr_sst_def by fast
  ultimately have 1: "tfr_set (trms_lst A')" by (metis tfr_subset(2)[OF _ *])

  have **: "list_all tfr_sstp (unlabel A)" using assms(2) unfolding tfr_sst_def by fast
  have "pair ` setops_sst (unlabel A) ⊆
        SMP (trms_sst (unlabel A) ∪ pair ` setops_sst (unlabel A)) - Var`𝒱"
    using setops_sst_are_pairs unfolding pair_def by auto
  hence "Γ t = Γ t'"
    when "∃δ. Unifier δ t t'" "t ∈ pair ` setops_sst (unlabel A)" "t' ∈ pair ` setops_sst (unlabel A)"
    for t t'
    using that assms(2) unfolding tfr_sst_def tfr_set_def by blast
  moreover have "fv_pairs (unlabel []) = {}" "pair ` snd ` set [] = {}" by auto
  ultimately have 2: "list_all tfr_stp (unlabel A')"
    using tr_par_tfr_sstp[OF assms(1) ** assms(3)] by simp

  show ?thesis by (metis 1 2 tfr_st_def)
qed


lemma tr_par_proj:
  assumes "B ∈ set (tr_pc A D)"
  shows "proj n B ∈ set (tr_pc (proj n A) (proj n D))"
using assms
proof (induction A D arbitrary: B rule: tr_pc.induct)
  case (5 i t s S D)
  note prems = "5.prems"
  note IH = "5.IH"
  have IH': "proj n B ∈ set (tr_pc (proj n S) (proj n (List.insert (i,t,s) D)))"
    using prems IH by auto
  show ?case
  proof (cases "(i = ln n) ∨ (i = ⋆)")
    case True thus ?thesis
      using IH' proj_list_insert(1,2)[of n "(t,s)" D] proj_list_Cons(1,2)[of n _ S]
      by auto
  next
    case False
    then obtain m where "i = ln m" "n ≠ m" by (cases i) simp_all
    thus ?thesis
      using IH' proj_list_insert(3)[of n _ "(t,s)" D] proj_list_Cons(3)[of n _ "insert⟨t,s⟩" S]
      by auto
  qed
next
  case (6 i t s S D)
  note prems = "6.prems"
  note IH = "6.IH"
  define constr where "constr = (λDi D.
      (map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di)@
      (map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) [d←dbproj i D. d ∉ set Di]))"

  obtain Di B' where B':
      "B = constr Di D@B'"
      "Di ∈ set (subseqs (dbproj i D))"
      "B' ∈ set (tr_pc S [d←D. d ∉ set Di])"
    using prems constr_def by fastforce
  hence "proj n B' ∈ set (tr_pc (proj n S) (proj n [d←D. d ∉ set Di]))" using IH by simp
```

```
  hence IH': "proj n B' ∈ set (tr_pc (proj n S) [d←proj n D. d ∉ set Di])" by (metis proj_filter)
  show ?case
  proof (cases "(i = ln n) ∨ (i = ⋆)")
    case True
    hence "proj n B = constr Di D@proj n B'" "Di ∈ set (subseqs (dbproj i (proj n D)))"
      using B'(1,2) proj_dbproj(1,2)[of n D] unfolding proj_def constr_def by auto
    moreover have "constr Di (proj n D) = constr Di D"
      using True proj_dbproj(1,2)[of n D] unfolding constr_def by presburger
    ultimately have "proj n B ∈ set (tr_pc ((i, delete⟨t,s⟩)#proj n S) (proj n D))"
      using IH' unfolding constr_def by force
    thus ?thesis by (metis proj_list_Cons(1,2) True)
  next
    case False
    then obtain m where m: "i = ln m" "n ≠ m" by (cases i) simp_all
    hence *: "(ln n) ≠ i" by simp
    have "proj n B = proj n B'" using B'(1) False unfolding constr_def proj_def by auto
    moreover have "[d←proj n D. d ∉ set Di] = proj n D"
      using proj_subseq[OF _ m(2)[symmetric]] m(1) B'(2) by simp
    ultimately show ?thesis using m(1) IH' proj_list_Cons(3)[OF m(2), of _ S] by auto
  qed
next
  case (7 i ac t s S D)
  note prems = "7.prems"
  note IH = "7.IH"
  define constr where "constr = (
    λd::'lbl strand_label × ('fun,'var) term × ('fun,'var) term.
      (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st))"

  obtain d B' where B':
      "B = constr d#B'"
      "d ∈ set (dbproj i D)"
      "B' ∈ set (tr_pc S D)"
    using prems constr_def by fastforce
  hence IH': "proj n B' ∈ set (tr_pc (proj n S) (proj n D))" using IH by auto

  show ?case
  proof (cases "(i = ln n) ∨ (i = ⋆)")
    case True
    hence "proj n B = constr d#proj n B'" "d ∈ set (dbproj i (proj n D))"
      using B' proj_list_Cons(1,2)[of n _ B']
      unfolding constr_def
      by (force, metis proj_dbproj(1,2))
    hence "proj n B ∈ set (tr_pc ((i, InSet ac t s)#proj n S) (proj n D))"
      using IH' unfolding constr_def by auto
    thus ?thesis using proj_list_Cons(1,2)[of n _ S] True by metis
  next
    case False
    then obtain m where m: "i = ln m" "n ≠ m" by (cases i) simp_all
    hence "proj n B = proj n B'" using B'(1) proj_list_Cons(3) unfolding constr_def by auto
    thus ?thesis
      using IH' m proj_list_Cons(3)[OF m(2), of "InSet ac t s" S]
      unfolding constr_def
      by auto
  qed
next
  case (8 i X F F' S D)
  note prems = "8.prems"
  note IH = "8.IH"

  define constr where
    "constr = (λD. map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd (dbproj i D))))"

  obtain B' where B':
```

```
      "B = constr D@B'"
      "B' ∈ set (tr_pc S D)"
    using prems constr_def by fastforce
  hence IH': "proj n B' ∈ set (tr_pc (proj n S) (proj n D))" using IH by auto

  show ?case
  proof (cases "(i = ln n) ∨ (i = ⋆)")
    case True
    hence "proj n B = constr (proj n D)@proj n B'"
      using B'(1,2) proj_dbproj(1,2)[of n D] unfolding proj_def constr_def by auto
    hence "proj n B ∈ set (tr_pc ((i, NegChecks X F F')#proj n S) (proj n D))"
      using IH' unfolding constr_def by auto
    thus ?thesis using proj_list_Cons(1,2)[of n _ S] True by metis
  next
    case False
    then obtain m where m: "i = ln m" "n ≠ m" by (cases i) simp_all
    hence "proj n B = proj n B'" using B'(1) unfolding constr_def proj_def by auto
    thus ?thesis
      using IH' m proj_list_Cons(3)[OF m(2), of "NegChecks X F F'" S]
      unfolding constr_def
      by auto
  qed
qed (force simp add: proj_def)+

lemma tr_par_preserves_typing_cond:
  assumes "par_comp_lsst A Sec" "typing_cond_sst (unlabel A)" "A' ∈ set (tr_pc A [])"
  shows "typing_cond (unlabel A')"
proof -
  have "wf'_sst {} (unlabel A)"
       "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}"
       "wf_trms (trms_sst (unlabel A))"
    using assms(2) unfolding typing_cond_sst_def by simp_all
  hence 1: "wf_st {} (unlabel A')"
           "fv_st (unlabel A') ∩ bvars_st (unlabel A') = {}"
           "wf_trms (trms_st (unlabel A'))"
           "Ana_invar_subst (ik_st (unlabel A') ∪ assignment_rhs_st (unlabel A'))"
    using tr_par_wf[OF assms(3)] Ana_invar_subst' by metis+

  have 2: "tfr_st (unlabel A')" by (metis tr_par_tfr assms(2,3) typing_cond_sst_def)

  show ?thesis by (metis 1 2 typing_cond_def)
qed

lemma tr_par_preserves_par_comp:
  assumes "par_comp_lsst A Sec" "A' ∈ set (tr_pc A [])"
  shows "par_comp A' Sec"
proof -
  let ?M = "λl. trms_sst (proj_unl l A) ∪ pair ` setops_sst (proj_unl l A)"
  let ?N = "λl. trms_proj_lst l A'"

  have 0: "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (?M l1) (?M l2) Sec"
    using assms(1) unfolding par_comp_lsst_def by simp_all

  { fix l1 l2::'lbl assume *: "l1 ≠ l2"
    hence "GSMP_disjoint (?M l1) (?M l2) Sec" using 0(1) by metis
    moreover have "pair ` snd ` set (proj n []) = {}" for n::'lbl unfolding proj_def by simp
    hence "?N l1 ⊆ ?M l1" "?N l2 ⊆ ?M l2"
      using tr_par_trms_subset[OF tr_par_proj[OF assms(2)]] by (metis Un_empty_right)+
    ultimately have "GSMP_disjoint (?N l1) (?N l2) Sec"
      using GSMP_disjoint_subset by presburger
  } hence 1: "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (trms_proj_lst l1 A') (trms_proj_lst l2 A') Sec"
    using 0(1) by metis
```

```
  have 2: "ground Sec" "∀s ∈ Sec. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Sec"
    using assms(1) unfolding par_comp_lsst_def by metis+

  show ?thesis using 1 2 unfolding par_comp_def by metis
qed

lemma tr_leaking_prefix_exists:
  assumes "A' ∈ set (tr_pc A [])" "prefix B A'" "ik_st (proj_unl n B) ·set I ⊢ t · I"
  shows "∃C D. prefix C B ∧ prefix D A ∧ C ∈ set (tr_pc D []) ∧ (ik_st (proj_unl n C) ·set I ⊢ t ·
I)"
proof -
  let ?P = "λB C C'. B = C@C' ∧ (∀n t. (n, receive⟨t⟩st) ∉ set C') ∧
                    (C = [] ∨ (∃n t. suffix [(n,receive⟨t⟩st)] C))"
  have "∃C C'. ?P B C C'"
  proof (induction B)
    case (Cons b B)
    then obtain C C' n s where *: "?P B C C'" "b = (n,s)" by moura
    show ?case
    proof (cases "C = []")
      case True
      note T = True
      show ?thesis
      proof (cases "∃t. s = receive⟨t⟩st")
        case True
        hence "?P (b#B) [b] C'" using * T  by auto
        thus ?thesis by metis
      next
        case False
        hence "?P (b#B) [] (b#C')" using * T by auto
        thus ?thesis by metis
      qed
    next
      case False
      hence "?P (b#B) (b#C) C'" using * unfolding suffix_def by auto
      thus ?thesis by metis
    qed
  qed simp
  then obtain C C' where C:
      "B = C@C'" "∀n t. (n, receive⟨t⟩st) ∉ set C'"
      "C = [] ∨ (∃n t. suffix [(n,receive⟨t⟩st)] C)"
    by moura
  hence 1: "prefix C B" by simp
  hence 2: "prefix C A'" using assms(2) by simp

  have "⋀m t. (m,receive⟨t⟩st) ∈ set B ⟹ (m,receive⟨t⟩st) ∈ set C" using C by auto
  hence "⋀t. receive⟨t⟩st ∈ set (proj_unl n B) ⟹ receive⟨t⟩st ∈ set (proj_unl n C)"
    unfolding unlabel_def proj_def by force
  hence "ik_st (proj_unl n B) ⊆ ik_st (proj_unl n C)" using ik_st_is_rcv_set by auto
  hence 3: "ik_st (proj_unl n C) ·set I ⊢ t · I" by (metis ideduct_mono[OF assms(3)] subst_all_mono)

  { fix D E m t assume  "suffix [(m, receive⟨t⟩st)] E" "prefix E A'" "A' ∈ set (tr_pc A D)"
    hence "∃F. prefix F A ∧ E ∈ set (tr_pc F D)"
    proof (induction A D arbitrary: A' E rule: tr_pc.induct)
      case (1 D) thus ?case by simp
    next
      case (2 i t' S D)
      note prems = "2.prems"
      note IH = "2.IH"
      obtain A'' where *: "A' = (i,send⟨t'⟩st)#A''" "A'' ∈ set (tr_pc S D)"
        using prems(3) by auto
      have "E ≠ []" using prems(1) by auto
      then obtain E' where **: "E = (i,send⟨t'⟩st)#E'"
        using *(1) prems(2) by (cases E) auto
```

```
    hence "suffix [(m, receive⟨t⟩st)] E'" "prefix E' A''"
      using *(1) prems(1,2) suffix_Cons[of _ _ E'] by auto
    then obtain F where "prefix F S" "E' ∈ set (trpc F D)"
      using *(2) ** IH by metis
    hence "prefix ((i,Send t')#F) ((i,Send t')#S)" "E ∈ set (trpc ((i,Send t')#F) D)"
      using ** by auto
    thus ?case by metis
  next
    case (3 i t' S D)
    note prems = "3.prems"
    note IH = "3.IH"
    obtain A'' where *: "A' = (i,receive⟨t'⟩st)#A''" "A'' ∈ set (trpc S D)"
      using prems(3) by auto
    have "E ≠ []" using prems(1) by auto
    then obtain E' where **: "E = (i,receive⟨t'⟩st)#E'"
      using *(1) prems(2) by (cases E) auto
    show ?case
    proof (cases "(m, receive⟨t⟩st) = (i, receive⟨t'⟩st)")
      case True
      note T = True
      show ?thesis
      proof (cases "suffix [(m, receive⟨t⟩st)] E'")
        case True
        hence "suffix [(m, receive⟨t⟩st)] E'" "prefix E' A''"
          using ** *(1) prems(1,2) by auto
        then obtain F where "prefix F S" "E' ∈ set (trpc F D)"
          using *(2) ** IH by metis
        hence "prefix ((i,receive⟨t'⟩)#F) ((i,receive⟨t'⟩)#S)"
              "E ∈ set (trpc ((i,receive⟨t'⟩)#F) D)"
          using ** by auto
        thus ?thesis by metis
      next
        case False
        hence "E' = []"
          using **(1) T prems(1)
                suffix_Cons[of "[(m, receive⟨t⟩st)]" "(m, receive⟨t⟩st)" E']
          by auto
        hence "prefix [(i,receive⟨t'⟩)] ((i,receive⟨t'⟩) # S) ∧ E ∈ set (trpc [(i,receive⟨t'⟩)] D)"
          using * ** prems by auto
        thus ?thesis by metis
      qed
    next
      case False
      hence "suffix [(m, receive⟨t⟩st)] E'" "prefix E' A''"
        using ** *(1) prems(1,2) suffix_Cons[of _ _ E'] by auto
      then obtain F where "prefix F S" "E' ∈ set (trpc F D)" using *(2) ** IH by metis
      hence "prefix ((i,receive⟨t'⟩)#F) ((i,receive⟨t'⟩)#S)" "E ∈ set (trpc ((i,receive⟨t'⟩)#F) D)"
        using ** by auto
      thus ?thesis by metis
    qed
  next
    case (4 i ac t' t'' S D)
    note prems = "4.prems"
    note IH = "4.IH"
    obtain A'' where *: "A' = (i,⟨ac: t' ≐ t''⟩st)#A''" "A'' ∈ set (trpc S D)"
      using prems(3) by auto
    have "E ≠ []" using prems(1) by auto
    then obtain E' where **: "E = (i,⟨ac: t' ≐ t''⟩st)#E'"
      using *(1) prems(2) by (cases E) auto
    hence "suffix [(m, receive⟨t⟩st)] E'" "prefix E' A''"
      using *(1) prems(1,2) suffix_Cons[of _ _ E'] by auto
    then obtain F where "prefix F S" "E' ∈ set (trpc F D)"
      using *(2) ** IH by metis
```

```
    hence "prefix ((i,Equality ac t' t'')#F) ((i,Equality ac t' t'')#S)"
          "E ∈ set (tr_pc ((i,Equality ac t' t'')#F) D)"
      using ** by auto
    thus ?case by metis
  next
    case (5 i t' s S D)
    note prems = "5.prems"
    note IH = "5.IH"
    have *: "A' ∈ set (tr_pc S (List.insert (i,t',s) D))" using prems(3) by auto
    have "E ≠ []" using prems(1) by auto
    hence "suffix [(m, receive⟨t⟩_st)] E" "prefix E A'"
      using *(1) prems(1,2) suffix_Cons[of _ _ E] by auto
    then obtain F where "prefix F S" "E ∈ set (tr_pc F (List.insert (i,t',s) D))"
      using * IH by metis
    hence "prefix ((i,insert⟨t',s⟩)#F) ((i,insert⟨t',s⟩)#S)"
          "E ∈ set (tr_pc ((i,insert⟨t',s⟩)#F) D)"
      by auto
    thus ?case by metis
  next
    case (6 i t' s S D)
    note prems = "6.prems"
    note IH = "6.IH"

    define constr where "constr = (λDi.
      (map (λd. (i,⟨check: (pair (t',s)) ≐ (pair (snd d))⟩_st)) Di)@
      (map (λd. (i,∀ []⟨∨≠: [(pair (t',s), pair (snd d))]⟩_st))
        (filter (λd. d ∉ set Di) (dbproj i D))))"

    obtain A'' Di where *:
        "A' = constr Di@A''" "A'' ∈ set (tr_pc S (filter (λd. d ∉ set Di) D))"
        "Di ∈ set (subseqs (dbproj i D))"
      using prems(3) constr_def by auto
    have ***:  "(m, receive⟨t⟩_st) ∉ set (constr Di)" using constr_def by auto
    have "E ≠ []" using prems(1) by auto
    then obtain E' where **: "E = constr Di@E'"
      using *(1) prems(1,2) ***
      by (metis (mono_tags, lifting) Un_iff list.set_intros(1) prefixI prefix_def
                                     prefix_same_cases set_append suffix_def)
    hence "suffix [(m, receive⟨t⟩_st)] E'" "prefix E' A''"
      using *(1) prems(1,2) suffix_append[of "[(m,receive⟨t⟩_st)]" "constr Di" E'] ***
      by (metis (no_types, hide_lams) Nil_suffix append_Nil2 in_set_conv_decomp rev_exhaust
                                      snoc_suffix_snoc suffix_appendD,
          auto)
    then obtain F where "prefix F S" "E' ∈ set (tr_pc F (filter (λd. d ∉ set Di) D))"
      using *(2,3) ** IH by metis
    hence "prefix ((i,delete⟨t',s⟩)#F) ((i,delete⟨t',s⟩)#S)"
          "E ∈ set (tr_pc ((i,delete⟨t',s⟩)#F) D)"
      using *(3) ** constr_def by auto
    thus ?case by metis
  next
    case (7 i ac t' s S D)
    note prems = "7.prems"
    note IH = "7.IH"

    define constr where "constr = (
      λd::(('lbl strand_label × ('fun,'var) term × ('fun,'var) term)).
        (i,⟨ac: (pair (t',s)) ≐ (pair (snd d))⟩_st))"

    obtain A'' d where *: "A' = constr d#A''" "A'' ∈ set (tr_pc S D)" "d ∈ set (dbproj i D)"
      using prems(3) constr_def by auto
    have "E ≠ []" using prems(1) by auto
    then obtain E' where **: "E = constr d#E'" using *(1) prems(2) by (cases E) auto
    hence "suffix [(m, receive⟨t⟩_st)] E'" "prefix E' A''"
```

```
      using *(1) prems(1,2) suffix_Cons[of _ _ E'] using constr_def by auto
    then obtain F where "prefix F S" "E' ∈ set (tr_pc F D)" using *(2) ** IH by metis
    hence "prefix ((i,InSet ac t' s)#F) ((i,InSet ac t' s)#S)"
          "E ∈ set (tr_pc ((i,InSet ac t' s)#F) D)"
      using *(3) ** unfolding constr_def by auto
    thus ?case by metis
  next
    case (8 i X G G' S D)
    note prems = "8.prems"
    note IH = "8.IH"

    define constr where
      "constr = map (λH. (i,∀X⟨∨≠: (G@H)⟩_st)) (tr_pairs G' (map snd (dbproj i D)))"

    obtain A'' where *: "A' = constr@A''" "A'' ∈ set (tr_pc S D)"
      using prems(3) constr_def by auto
    have ***:  "(m, receive⟨t⟩_st) ∉ set constr" using constr_def by auto
    have "E ≠ []" using prems(1) by auto
    then obtain E' where **: "E = constr@E'"
      using *(1) prems(1,2) ***
      by (metis (mono_tags, lifting) Un_iff list.set_intros(1) prefixI prefix_def
                                     prefix_same_cases set_append suffix_def)
    hence "suffix [(m, receive⟨t⟩_st)] E'" "prefix E' A''"
      using *(1) prems(1,2) suffix_append[of "[(m,receive⟨t⟩_st)]" constr E'] ***
      by (metis (no_types, hide_lams) Nil_suffix append_Nil2 in_set_conv_decomp rev_exhaust
                                      snoc_suffix_snoc suffix_appendD,
          auto)
    then obtain F where "prefix F S" "E' ∈ set (tr_pc F D)" using *(2) ** IH by metis
    hence "prefix ((i,NegChecks X G G')#F) ((i,NegChecks X G G')#S)"
          "E ∈ set (tr_pc ((i,NegChecks X G G')#F) D)"
      using ** constr_def by auto
    thus ?case by metis
  qed
}
moreover have "prefix [] A" "[] ∈ set (tr_pc [] [])" by auto
ultimately have 4: "∃D. prefix D A ∧ C ∈ set (tr_pc D [])" using C(3) assms(1) 2 by blast

show ?thesis by (metis 1 3 4)
qed
```

6.2.5 Theorem: Semantic Equivalence of Translation

```
context
begin
```

An alternative version of the translation that does not perform database-state projections. It is used as an intermediate step in the proof of semantic equivalence.

```
private fun tr'_pc::
  "('fun,'var,'lbl) labeled_stateful_strand ⇒ ('fun,'var,'lbl) labeleddbstatelist
   ⇒ ('fun,'var,'lbl) labeled_strand list"
where
  "tr'_pc [] D = [[]]"
| "tr'_pc ((i,send⟨t⟩)#A) D = map ((#) (i,send⟨t⟩_st)) (tr'_pc A D)"
| "tr'_pc ((i,receive⟨t⟩)#A) D = map ((#) (i,receive⟨t⟩_st)) (tr'_pc A D)"
| "tr'_pc ((i,⟨ac: t ≐ t'⟩)#A) D = map ((#) (i,⟨ac: t ≐ t'⟩_st)) (tr'_pc A D)"
| "tr'_pc ((i,insert⟨t,s⟩)#A) D = tr'_pc A (List.insert (i,(t,s)) D)"
| "tr'_pc ((i,delete⟨t,s⟩)#A) D = (
    concat (map (λDi. map (λB. (map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di)@
                               (map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st))
                                    [d←D. d ∉ set Di])@B)
                          (tr'_pc A [d←D. d ∉ set Di]))
                (subseqs D)))"
| "tr'_pc ((i,⟨ac: t ∈ s⟩)#A) D =
```

```
    concat (map (λB. map (λd. (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st)#B) D) (tr'pc A D))"
| "tr'pc ((i,∀X⟨∨≠: F ∨∉: F'⟩)#A) D =
    map ((@) (map (λG. (i,∀X⟨∨≠: (F@G)⟩st)) (trpairs F' (map snd D)))) (tr'pc A D)"
```

Part 1

```
private lemma tr'_par_iff_unlabel_tr:
  assumes "∀(i,p) ∈ setopslsst A ∪ set D.
           ∀(j,q) ∈ setopslsst A ∪ set D.
              p = q ⟶ i = j"
  shows "(∃C ∈ set (tr'pc A D). B = unlabel C) ⟷ B ∈ set (tr (unlabel A) (unlabel D))"
    (is "?A ⟷ ?B")
proof
  { fix C have "C ∈ set (tr'pc A D) ⟹ unlabel C ∈ set (tr (unlabel A) (unlabel D))" using assms
    proof (induction A D arbitrary: C rule: tr'pc.induct)
      case (5 i t s S D)
      hence "unlabel C ∈ set (tr (unlabel S) (unlabel (List.insert (i, t, s) D)))"
        by (auto simp add: setopslsst_def)
      moreover have
          "insert (i,t,s) (set D) ⊆ setopslsst ((i,insert⟨t,s⟩)#S) ∪ set D"
        by (auto simp add: setopslsst_def)
      hence "∀(j,p) ∈ insert (i,t,s) (set D). ∀(k,q) ∈ insert (i,t,s) (set D). p = q ⟶ j = k"
        using "5.prems"(2) by blast
      hence "unlabel (List.insert (i, t, s) D) = (List.insert (t, s) (unlabel D))"
        using map_snd_list_insert_distrib[of "(i,t,s)" D] unfolding unlabel_def by simp
      ultimately show ?case by auto
    next
      case (6 i t s S D)
      let ?f1 = "λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st"
      let ?g1 = "λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st"
      let ?f2 = "λd. (i, ?f1 (snd d))"
      let ?g2 = "λd. (i, ?g1 (snd d))"

      define constr1 where "constr1 = (λDi. (map ?f1 Di)@(map ?g1 [d←unlabel D. d ∉ set Di]))"
      define constr2 where "constr2 = (λDi. (map ?f2 Di)@(map ?g2 [d←D. d ∉ set Di]))"

      obtain C' Di where C':
          "Di ∈ set (subseqs D)"
          "C = constr2 Di@C'"
          "C' ∈ set (tr'pc S [d←D. d ∉ set Di])"
        using "6.prems"(1) unfolding constr2_def by moura

      have 0: "set [d←D. d ∉ set Di] ⊆ set D"
              "setopslsst S ⊆ setopslsst ((i, delete⟨t,s⟩)#S)"
        by (auto simp add: setopslsst_def)
      hence 1:
          "∀(j, p) ∈ setopslsst S ∪ set [d←D. d ∉ set Di].
           ∀(k, q) ∈ setopslsst S ∪ set [d←D. d ∉ set Di].
             p = q ⟶ j = k"
        using "6.prems"(2) by blast

      have "∀(i,p) ∈ set D ∪ set Di. ∀(j,q) ∈ set D ∪ set Di. p = q ⟶ i = j"
        using "6.prems"(2) subseqs_set_subset(1)[OF C'(1)] by blast
      hence 2: "unlabel [d←D. d ∉ set Di] = [d←unlabel D. d ∉ set (unlabel Di)]"
        using unlabel_filter_eq[of D "set Di"] unfolding unlabel_def by simp

      have 3:
          "⋀f g::('a × 'a ⇒ 'c). ⋀A B::(('b × 'a × 'a) list).
              map snd ((map (λd. (i, f (snd d))) A)@(map (λd. (i, g (snd d))) B)) =
              map f (map snd A)@map g (map snd B)"
        by simp
      have "unlabel (constr2 Di) = constr1 (unlabel Di)"
        using 2 3[of ?f1 Di ?g1 "[d←D. d ∉ set Di]"]
```

```
        by (simp add: constr1_def constr2_def unlabel_def)
      hence 4: "unlabel C = constr1 (unlabel Di)@unlabel C'"
        using C'(2) unlabel_append by metis

      have "unlabel Di ∈ set (map unlabel (subseqs D))"
        using C'(1) unfolding unlabel_def by simp
      hence 5: "unlabel Di ∈ set (subseqs (unlabel D))"
        using map_subseqs[of snd D] unfolding unlabel_def by simp

      show ?case using "6.IH"[OF C'(1,3) 1] 2 4 5 unfolding constr1_def by auto
    next
      case (7 i ac t s S D)
      obtain C' d  where C':
          "C = (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#C'"
          "C' ∈ set (tr'_pc S D)" "d ∈ set D"
        using "7.prems"(1) by moura

      have "setops_lsst S ∪ set D ⊆ setops_lsst ((i,InSet ac t s)#S) ∪ set D"
        by (auto simp add: setops_lsst_def)
      hence "∀(j, p) ∈ setops_lsst S ∪ set D.
              ∀(k, q) ∈ setops_lsst S ∪ set D.
               p = q ⟶ j = k"
        using "7.prems"(2) by blast
      hence "unlabel C' ∈ set (tr (unlabel S) (unlabel D))" using "7.IH"[OF C'(2)] by auto
      thus ?case using C' unfolding unlabel_def by force
    next
      case (8 i X F F' S D)
      obtain C' where C':
          "C = map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd D))@C'"
          "C' ∈ set (tr'_pc S D)"
        using "8.prems"(1) by moura

      have "setops_lsst S ∪ set D ⊆ setops_lsst ((i,NegChecks X F F')#S) ∪ set D"
        by (auto simp add: setops_lsst_def)
      hence "∀(j, p) ∈ setops_lsst S ∪ set D.
              ∀(k, q) ∈ setops_lsst S ∪ set D.
               p = q ⟶ j = k"
        using "8.prems"(2) by blast
      hence "unlabel C' ∈ set (tr (unlabel S) (unlabel D))" using "8.IH"[OF C'(2)] by auto
      thus ?case using C' unfolding unlabel_def by auto
    qed (auto simp add: setops_lsst_def)
  } thus "?A ⟹ ?B" by blast

  show "?B ⟹ ?A" using assms
  proof (induction A arbitrary: B D)
    case (Cons a A)
    obtain ia sa where a: "a = (ia,sa)" by moura

    have "setops_lsst A ⊆ setops_lsst (a#A)" using a by (cases sa) (auto simp add: setops_lsst_def)
    hence 1: "∀(j, p) ∈ setops_lsst A ∪ set D.
               ∀(k, q) ∈ setops_lsst A ∪ set D.
                 p = q ⟶ j = k"
      using Cons.prems(2) by blast

    show ?case
    proof (cases sa)
      case (Send t)
      then obtain B' where B':
          "B = send⟨t⟩_st#B'"
          "B' ∈ set (tr (unlabel A) (unlabel D))"
        using Cons.prems(1) a by auto
      thus ?thesis using Cons.IH[OF B'(2) 1] a B'(1) Send by auto
    next
```

```
    case (Receive t)
    then obtain B' where B':
        "B = receive⟨t⟩st#B'"
        "B' ∈ set (tr (unlabel A) (unlabel D))"
      using Cons.prems(1) a by auto
    thus ?thesis using Cons.IH[OF B'(2) 1] a B'(1) Receive by auto
  next
    case (Equality ac t t')
    then obtain B' where B':
        "B = ⟨ac: t ≐ t'⟩st#B'"
        "B' ∈ set (tr (unlabel A) (unlabel D))"
      using Cons.prems(1) a by auto
    thus ?thesis using Cons.IH[OF B'(2) 1] a B'(1) Equality by auto
  next
    case (Insert t s)
    hence B: "B ∈ set (tr (unlabel A) (List.insert (t,s) (unlabel D)))"
      using Cons.prems(1) a by auto

    let ?P = "λi. List.insert (t,s) (unlabel D) = unlabel (List.insert (i,t,s) D)"

    { obtain j where j: "?P j" "j = ia ∨ (j,t,s) ∈ set D"
        using labeled_list_insert_eq_ex_cases[of "(t,s)" D ia] by moura
      hence "j = ia" using Cons.prems(2) a Insert by (auto simp add: setops_lsst_def)
      hence "?P ia" using j(1) by metis
    } hence j: "?P ia" by metis

    have 2: "∀(k1, p) ∈ setops_lsst A ∪ set (List.insert (ia,t,s) D).
             ∀(k2, q) ∈ setops_lsst A ∪ set (List.insert (ia,t,s) D).
               p = q ⟶ k1 = k2"
      using Cons.prems(2) a Insert by (auto simp add: setops_lsst_def)

    show ?thesis using Cons.IH[OF _ 2] j(1) B Insert a by auto
  next
    case (Delete t s)
    define c where "c ≡ (λ(i::'lbl strand_label) Di.
      map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩st)) Di@
      map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩st)) [d←D. d ∉ set Di])"

    define d where "d ≡ (λDi.
      map (λd. ⟨check: (pair (t,s)) ≐ (pair d)⟩st) Di@
      map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩st) [d←unlabel D. d ∉ set Di])"

    obtain B' Di where B':
        "B = d Di@B'" "Di ∈ set (subseqs (unlabel D))"
        "B' ∈ set (tr (unlabel A) [d←unlabel D. d ∉ set Di])"
      using Cons.prems(1) a Delete unfolding d_def by auto

    obtain Di' where Di': "Di' ∈ set (subseqs D)" "unlabel Di' = Di"
      using unlabel_subseqsD[OF B'(2)] by moura

    have 2: "∀(j, p) ∈ setops_lsst A ∪ set [d←D. d ∉ set Di'].
             ∀(k, q) ∈ setops_lsst A ∪ set [d←D. d ∉ set Di'].
               p = q ⟶ j = k"
      using 1 subseqs_subset[OF Di'(1)]
            filter_is_subset[of "λd. d ∉ set Di'"]
      by blast

    have "set Di' ⊆ set D" by (rule subseqs_subset[OF Di'(1)])
    hence "∀(j, p)∈set D ∪ set Di'. ∀(k, q)∈set D ∪ set Di'. p = q ⟶ j = k"
      using Cons.prems(2) by blast
    hence 3: "[d←unlabel D. d ∉ set Di] = unlabel [d←D. d ∉ set Di']"
      using Di'(2) unlabel_filter_eq[of D "set Di'"] unfolding unlabel_def by auto
```

```
      obtain C where C: "C ∈ set (tr'_pc A [d←D. d ∉ set Di'])" "B' = unlabel C"
        using 3 Cons.IH[OF _ 2] B'(3) by auto
      hence 4: "c ia Di'@C ∈ set (tr'_pc (a#A) D)" using Di'(1) a Delete unfolding c_def by auto

      have "unlabel (c ia Di') = d Di" using Di' 3 unfolding c_def d_def unlabel_def by auto
      hence 5: "B = unlabel (c ia Di'@C)" using B'(1) C(2) unlabel_append[of "c ia Di'" C] by simp

      show ?thesis using 4 5 by blast
    next
      case (InSet ac t s)
      then obtain B' d where B':
          "B = ⟨ac: (pair (t,s)) ≐ (pair d)⟩_st#B'"
          "B' ∈ set (tr (unlabel A) (unlabel D))"
          "d ∈ set (unlabel D)"
        using Cons.prems(1) a by auto
      thus ?thesis using Cons.IH[OF _ 1] a InSet unfolding unlabel_def by auto
    next
      case (NegChecks X F F')
      then obtain B' where B':
          "B = map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' (unlabel D))@B'"
          "B' ∈ set (tr (unlabel A) (unlabel D))"
        using Cons.prems(1) a by auto
      thus ?thesis using Cons.IH[OF _ 1] a NegChecks unfolding unlabel_def by auto
    qed
  qed simp
qed
```

Part 2

```
private lemma tr_par_iff_tr'_par:
  assumes "∀(i,p) ∈ setops_lsst A ∪ set D. ∀(j,q) ∈ setops_lsst A ∪ set D.
            (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j"
    (is "?R3 A D")
  and "∀(l,t,s) ∈ set D. (fv t ∪ fv s) ∩ bvars_sst (unlabel A) = {}" (is "?R4 A D")
  and "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}" (is "?R5 A D")
  shows "(∃B ∈ set (tr_pc A D). ⟦M; unlabel B⟧_d ℐ) ⟷ (∃C ∈ set (tr'_pc A D). ⟦M; unlabel C⟧_d ℐ)"
    (is "?P ⟷ ?Q")
proof
  { fix B assume "B ∈ set (tr_pc A D)" "⟦M; unlabel B⟧_d ℐ"
    hence ?Q using assms
    proof (induction A D arbitrary: B M rule: tr_pc.induct)
      case (1 D) thus ?case by simp
    next
      case (2 i t S D)
      note prems = "2.prems"
      note IH = "2.IH"

      obtain B' where B': "B = (i,send⟨t⟩_st)#B'" "B' ∈ set (tr_pc S D)"
        using prems(1) by moura

      have 1: "⟦M; unlabel B'⟧_d ℐ" using prems(2) B'(1) by simp
      have 4: "?R3 S D" using prems(3) by (auto simp add: setops_lsst_def)
      have 5: "?R4 S D" using prems(4) by force
      have 6: "?R5 S D" using prems(5) by force

      have 7: "M ⊢ t · ℐ" using prems(2) B'(1) by simp

      obtain C where C: "C ∈ set (tr'_pc S D)" "⟦M; unlabel C⟧_d ℐ"
        using IH[OF B'(2) 1 4 5 6] by moura
      hence "((i,send⟨t⟩_st)#C) ∈ set (tr'_pc ((i,Send t)#S) D)" "⟦M; unlabel ((i,send⟨t⟩_st)#C)⟧_d ℐ"
        using 7 by auto
      thus ?case by metis
    next
```

```
    case (3 i t S D)
    note prems = "3.prems"
    note IH = "3.IH"

    obtain B' where B': "B = (i,receive⟨t⟩st)#B'" "B' ∈ set (trpc S D)" using prems(1) by moura

    have 1: "⟦insert (t · I) M; unlabel B'⟧d I " using prems(2) B'(1) by simp
    have 4: "?R3 S D" using prems(3) by (auto simp add: setopslsst_def)
    have 5: "?R4 S D" using prems(4) by force
    have 6: "?R5 S D" using prems(5) by force

    obtain C where C: "C ∈ set (tr'pc S D)" "⟦insert (t · I) M; unlabel C⟧d I"
      using IH[OF B'(2) 1 4 5 6] by moura
    hence "((i,receive⟨t⟩st)#C) ∈ set (tr'pc ((i,receive⟨t⟩)#S) D)"
          "⟦insert (t · I) M; unlabel ((i,receive⟨t⟩st)#C)⟧d I"
      by auto
    thus ?case by auto
  next
    case (4 i ac t t' S D)
    note prems = "4.prems"
    note IH = "4.IH"

    obtain B' where B': "B = (i,⟨ac: t ≐ t'⟩st)#B'" "B' ∈ set (trpc S D)"
      using prems(1) by moura

    have 1: "⟦M; unlabel B'⟧d I " using prems(2) B'(1) by simp
    have 4: "?R3 S D" using prems(3) by (auto simp add: setopslsst_def)
    have 5: "?R4 S D" using prems(4) by force
    have 6: "?R5 S D" using prems(5) by force

    have 7: "t · I = t' · I" using prems(2) B'(1) by simp

    obtain C where C: "C ∈ set (tr'pc S D)" "⟦M; unlabel C⟧d I"
      using IH[OF B'(2) 1 4 5 6] by moura
    hence "((i,⟨ac: t ≐ t'⟩st)#C) ∈ set (tr'pc ((i,Equality ac t t')#S) D)"
          "⟦M; unlabel ((i,⟨ac: t ≐ t'⟩st)#C)⟧d I"
      using 7 by auto
    thus ?case by metis
  next
    case (5 i t s S D)
    note prems = "5.prems"
    note IH = "5.IH"

    have B: "B ∈ set (trpc S (List.insert (i,t,s) D))" using prems(1) by simp

    have 1: "⟦M; unlabel B⟧d I " using prems(2) B(1) by simp
    have 4: "?R3 S (List.insert (i,t,s) D)" using prems(3) by (auto simp add: setopslsst_def)
    have 5: "?R4 S (List.insert (i,t,s) D)" using prems(4,5) by force
    have 6: "?R5 S D" using prems(5) by force

    show ?case using IH[OF B(1) 1 4 5 6] by simp
  next
    case (6 i t s S D)
    note prems = "6.prems"
    note IH = "6.IH"

    let ?cl1 = "λDi. map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩st)) Di"
    let ?cu1 = "λDi. map (λd. ⟨check: (pair (t,s)) ≐ (pair (snd d))⟩st) Di"
    let ?cl2 = "λDi. map (λd. (i,∀ []⟨∨≠: [(pair (t,s), pair (snd d))]⟩st)) [d←dbproj i D. d∉set
Di]"
    let ?cu2 = "λDi. map (λd. ∀ []⟨∨≠: [(pair (t,s), pair (snd d))]⟩st) [d←dbproj i D. d∉set Di]"

    let ?dl1 = "λDi. map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩st)) Di"
```

```
let ?du1 = "λDi. map (λd. ⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st) Di"
let ?dl2 = "λDi. map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) [d←D. d∉set Di]"
let ?du2 = "λDi. map (λd. ∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←D. d∉set Di]"

define c where c: "c = (λDi. ?cl1 Di@?cl2 Di)"
define d where d: "d = (λDi. ?dl1 Di@?dl2 Di)"

obtain B' Di where B':
    "Di ∈ set (subseqs (dbproj i D))" "B = c Di@B'" "B' ∈ set (tr_pc S [d←D. d ∉ set Di])"
  using prems(1) c by moura

have 0: "ik_st (unlabel (c Di)) = {}" "ik_st (unlabel (d Di)) = {}"
        "unlabel (?cl1 Di) = ?cu1 Di" "unlabel (?cl2 Di) = ?cu2 Di"
        "unlabel (?dl1 Di) = ?du1 Di" "unlabel (?dl2 Di) = ?du2 Di"
  unfolding c d unlabel_def by force+

have 1: "⟦M; unlabel B'⟧_d ℐ " using prems(2) B'(2) 0(1) unfolding unlabel_def by auto

{ fix j p k q
  assume "(j, p) ∈ setops_lsst S ∪ set [d←D. d ∉ set Di]"
         "(k, q) ∈ setops_lsst S ∪ set [d←D. d ∉ set Di]"
  hence "(j, p) ∈ setops_lsst ((i, delete⟨t,s⟩)#S) ∪ set D"
        "(k, q) ∈ setops_lsst ((i, delete⟨t,s⟩)#S) ∪ set D"
    using dbproj_subseq_subset[OF B'(1)] by (auto simp add: setops_lsst_def)
  hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
} hence 4: "?R3 S [d←D. d ∉ set Di]" by blast

have 5: "?R4 S (filter (λd. d ∉ set Di) D)" using prems(4) by force
have 6: "?R5 S D" using prems(5) by force

obtain C where C: "C ∈ set (tr'_pc S [d←D . d ∉ set Di])" "⟦M; unlabel C⟧_d ℐ"
  using IH[OF B'(1,3) 1 4 5 6] by moura

have 7: "⟦M; unlabel (c Di)⟧_d ℐ" "⟦M; unlabel B'⟧_d ℐ"
  using prems(2) B'(2) 0(1) strand_sem_split(3,4)[of M "unlabel (c Di)" "unlabel B'"]
  unfolding c unlabel_def by auto

have "⟦M; unlabel (?cl2 Di)⟧_d ℐ" using 7(1) 0(1) unfolding c unlabel_def by auto
hence "⟦M; ?cu2 Di⟧_d ℐ" by (metis 0(4))
moreover {
  fix j p k q
  assume "(j, p) ∈ {(i, t, s)} ∪ set D ∪ set Di"
         "(k, q) ∈ {(i, t, s)} ∪ set D ∪ set Di"
  hence "(j, p) ∈ setops_lsst ((i, delete⟨t,s⟩)#S) ∪ set D"
        "(k, q) ∈ setops_lsst ((i, delete⟨t,s⟩)#S) ∪ set D"
    using dbproj_subseq_subset[OF B'(1)] by (auto simp add: setops_lsst_def)
  hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
} hence "∀(j, p) ∈ {(i, t, s)} ∪ set D ∪ set Di.
         ∀(k, q) ∈ {(i, t, s)} ∪ set D ∪ set Di.
          (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k"
  by blast
ultimately have "⟦M; ?du2 Di⟧_d ℐ" using labeled_sat_ineq_lift by simp
hence "⟦M; unlabel (?dl2 Di)⟧_d ℐ" by (metis 0(6))
moreover have "⟦M; unlabel (?cl1 Di)⟧_d ℐ" using 7(1) unfolding c unlabel_def by auto
hence "⟦M; unlabel (?dl1 Di)⟧_d ℐ" by (metis 0(3,5))
ultimately have "⟦M; unlabel (d Di)⟧_d ℐ" using 0(2) unfolding c d unlabel_def by force
hence 8: "⟦M; unlabel (d Di@C)⟧_d ℐ" using 0(2) C(2) unfolding unlabel_def by auto

have 9: "d Di@C ∈ set (tr'_pc ((i,delete⟨t,s⟩)#S) D)"
  using C(1) dbproj_subseq_in_subseqs[OF B'(1)]
  unfolding d unlabel_def by auto

show ?case by (metis 8 9)
```

```
next
  case (7 i ac t s S D)
  note prems = "7.prems"
  note IH = "7.IH"

  obtain B' d where B':
      "B = (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st)#B'"
      "B' ∈ set (trpc S D)" "d ∈ set (dbproj i D)"
    using prems(1) by moura

  have 1: "⟦M; unlabel B'⟧d ℐ " using prems(2) B'(1) by simp

  { fix j p k q
    assume "(j,p) ∈ setopslsst S ∪ set D"
           "(k,q) ∈ setopslsst S ∪ set D"
    hence "(j,p) ∈ setopslsst ((i, InSet ac t s)#S) ∪ set D"
          "(k,q) ∈ setopslsst ((i, InSet ac t s)#S) ∪ set D"
      by (auto simp add: setopslsst_def)
    hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
  } hence 4: "?R3 S D" by blast

  have 5: "?R4 S D" using prems(4) by force
  have 6: "?R5 S D" using prems(5) by force
  have 7: "pair (t,s) · ℐ = pair (snd d) · ℐ" using prems(2) B'(1) by simp

  obtain C where C: "C ∈ set (tr'pc S D)" "⟦M; unlabel C⟧d ℐ"
    using IH[OF B'(2) 1 4 5 6] by moura
  hence "((i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st)#C) ∈ set (tr'pc ((i,InSet ac t s)#S) D)"
        "⟦M; unlabel ((i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩st)#C)⟧d ℐ"
    using 7 B'(3) by auto
  thus ?case by metis
next
  case (8 i X F F' S D)
  note prems = "8.prems"
  note IH = "8.IH"

  let ?cl = "map (λG. (i,∀X⟨∨≠: (F@G)⟩st)) (trpairs F' (map snd (dbproj i D)))"
  let ?cu = "map (λG. ∀X⟨∨≠: (F@G)⟩st) (trpairs F' (map snd (dbproj i D)))"

  let ?dl = "map (λG. (i,∀X⟨∨≠: (F@G)⟩st)) (trpairs F' (map snd D))"
  let ?du = "map (λG. ∀X⟨∨≠: (F@G)⟩st) (trpairs F' (map snd D))"

  define c where c: "c = ?cl"
  define d where d: "d = ?dl"

  obtain B' where B': "B = c@B'" "B' ∈ set (trpc S D)" using prems(1) c by moura

  have 0: "ikst (unlabel c) = {}" "ikst (unlabel d) = {}"
          "unlabel ?cl = ?cu" "unlabel ?dl = ?du"
    unfolding c d unlabel_def by force+

  have "ikst (unlabel c) = {}" unfolding c unlabel_def by force
  hence 1: "⟦M; unlabel B'⟧d ℐ " using prems(2) B'(1) unfolding unlabel_def by auto

  have "setopslsst S ⊆ setopslsst ((i, NegChecks X F F')#S)" by (auto simp add: setopslsst_def)
  hence 4: "?R3 S D" using prems(3) by blast

  have 5: "?R4 S D" using prems(4) by force
  have 6: "?R5 S D" using prems(5) by force

  obtain C where C: "C ∈ set (tr'pc S D)" "⟦M; unlabel C⟧d ℐ"
    using IH[OF B'(2) 1 4 5 6] by moura
```

```
    have 7: "⟦M; unlabel c⟧d I" "⟦M; unlabel B'⟧d I"
      using prems(2) B'(1) 0(1) strand_sem_split(3,4)[of M "unlabel c" "unlabel B'"]
      unfolding c unlabel_def by auto

    have 8: "d@C ∈ set (tr'pc ((i,NegChecks X F F')#S) D)"
      using C(1) unfolding d unlabel_def by auto

    have "⟦M; unlabel ?cl⟧d I" using 7(1) unfolding c unlabel_def by auto
    hence "⟦M; ?cu⟧d I" by (metis 0(3))
    moreover {
      fix j p k q
      assume "(j, p) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D"
             "(k, q) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D"
      hence "(j, p) ∈ setopslsst ((i, NegChecks X F F')#S) ∪ set D"
            "(k, q) ∈ setopslsst ((i, NegChecks X F F')#S) ∪ set D"
        by (auto simp add: setopslsst_def)
      hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
    } hence "∀(j, p) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D.
             ∀(k, q) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D.
              (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k"
      by blast
    moreover have "fvpairs (map snd D) ∩ set X = {}"
      using prems(4) by fastforce
    ultimately have "⟦M; ?du⟧d I" using labeled_sat_ineq_dbproj_sem_equiv[of i] by simp
    hence "⟦M; unlabel ?dl⟧d I" by (metis 0(4))
    hence "⟦M; unlabel d⟧d I" using 0(2) unfolding c d unlabel_def by force
    hence 9: "⟦M; unlabel (d@C)⟧d I" using 0(2) C(2) unfolding unlabel_def by auto

    show ?case by (metis 8 9)
  qed
} thus "?P ⟹ ?Q" by metis

{ fix C assume "C ∈ set (tr'pc A D)" "⟦M; unlabel C⟧d I"
  hence ?P using assms
  proof (induction A D arbitrary: C M rule: tr'pc.induct)
    case (1 D) thus ?case by simp
  next
    case (2 i t S D)
    note prems = "2.prems"
    note IH = "2.IH"

    obtain C' where C': "C = (i,send⟨t⟩st)#C'" "C' ∈ set (tr'pc S D)"
      using prems(1) by moura

    have 1: "⟦M; unlabel C'⟧d I " using prems(2) C'(1) by simp
    have 4: "?R3 S D" using prems(3) by (auto simp add: setopslsst_def)
    have 5: "?R4 S D" using prems(4) by force
    have 6: "?R5 S D" using prems(5) by force

    have 7: "M ⊢ t · I" using prems(2) C'(1) by simp

    obtain B where B: "B ∈ set (trpc S D)" "⟦M; unlabel B⟧d I"
      using IH[OF C'(2) 1 4 5 6] by moura
    hence "((i,send⟨t⟩st)#B) ∈ set (trpc ((i,Send t)#S) D)"
          "⟦M; unlabel ((i,send⟨t⟩st)#B)⟧d I"
      using 7 by auto
    thus ?case by metis
  next
    case (3 i t S D)
    note prems = "3.prems"
    note IH = "3.IH"

    obtain C' where C': "C = (i,receive⟨t⟩st)#C'" "C' ∈ set (tr'pc S D)"
```

```
          using prems(1) by moura

        have 1: "⟦insert (t · ℐ) M; unlabel C'⟧_d ℐ " using prems(2) C'(1) by simp
        have 4: "?R3 S D" using prems(3) by (auto simp add: setops_lsst_def)
        have 5: "?R4 S D" using prems(4) by force
        have 6: "?R5 S D" using prems(5) by force

        obtain B where B: "B ∈ set (tr_pc S D)" "⟦insert (t · ℐ) M; unlabel B⟧_d ℐ"
          using IH[OF C'(2) 1 4 5 6] by moura
        hence "((i,receive⟨t⟩_st)#B) ∈ set (tr_pc ((i,receive⟨t⟩)#S) D)"
              "⟦insert (t · ℐ) M; unlabel ((i,receive⟨t⟩_st)#B)⟧_d ℐ"
          by auto
        thus ?case by auto
      next
        case (4 i ac t t' S D)
        note prems = "4.prems"
        note IH = "4.IH"

        obtain C' where C': "C = (i,⟨ac: t ≐ t'⟩_st)#C'" "C' ∈ set (tr'_pc S D)"
          using prems(1) by moura

        have 1: "⟦M; unlabel C'⟧_d ℐ " using prems(2) C'(1) by simp
        have 4: "?R3 S D" using prems(3) by (auto simp add: setops_lsst_def)
        have 5: "?R4 S D" using prems(4) by force
        have 6: "?R5 S D" using prems(5) by force

        have 7: "t · ℐ = t' · ℐ" using prems(2) C'(1) by simp

        obtain B where B: "B ∈ set (tr_pc S D)" "⟦M; unlabel B⟧_d ℐ"
          using IH[OF C'(2) 1 4 5 6] by moura
        hence "((i,⟨ac: t ≐ t'⟩_st)#B) ∈ set (tr_pc ((i,Equality ac t t')#S) D)"
              "⟦M; unlabel ((i,⟨ac: t ≐ t'⟩_st)#B)⟧_d ℐ"
          using 7 by auto
        thus ?case by metis
      next
        case (5 i t s S D)
        note prems = "5.prems"
        note IH = "5.IH"

        have C: "C ∈ set (tr'_pc S (List.insert (i,t,s) D))" using prems(1) by simp

        have 1: "⟦M; unlabel C⟧_d ℐ " using prems(2) C(1) by simp
        have 4: "?R3 S (List.insert (i,t,s) D)" using prems(3) by (auto simp add: setops_lsst_def)
        have 5: "?R4 S (List.insert (i,t,s) D)" using prems(4,5) by force
        have 6: "?R5 S (List.insert (i,t,s) D)" using prems(5) by force

        show ?case using IH[OF C(1) 1 4 5 6] by simp
      next
        case (6 i t s S D)
        note prems = "6.prems"
        note IH = "6.IH"

        let ?dl1 = "λDi. map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di"
        let ?du1 = "λDi. map (λd. ⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st) Di"
        let ?dl2 = "λDi. map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) [d←dbproj i D. d∉set
Di]"
        let ?du2 = "λDi. map (λd. ∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←dbproj i D. d∉set Di]"

        let ?cl1 = "λDi. map (λd. (i,⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st)) Di"
        let ?cu1 = "λDi. map (λd. ⟨check: (pair (t,s)) ≐ (pair (snd d))⟩_st) Di"
        let ?cl2 = "λDi. map (λd. (i,∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st)) [d←D. d∉set Di]"
        let ?cu2 = "λDi. map (λd. ∀[]⟨∨≠: [(pair (t,s), pair (snd d))]⟩_st) [d←D. d∉set Di]"
```

```
  define c where c: "c = (λDi. ?cl1 Di@?cl2 Di)"
  define d where d: "d = (λDi. ?dl1 Di@?dl2 Di)"

  obtain C' Di where C':
      "Di ∈ set (subseqs D)" "C = c Di@C'" "C' ∈ set (tr'pc S [d←D. d ∉ set Di])"
    using prems(1) c by moura

  have 0: "ikst (unlabel (c Di)) = {}" "ikst (unlabel (d Di)) = {}"
          "unlabel (?cl1 Di) = ?cu1 Di" "unlabel (?cl2 Di) = ?cu2 Di"
          "unlabel (?dl1 Di) = ?du1 Di" "unlabel (?dl2 Di) = ?du2 Di"
    unfolding c d unlabel_def by force+

  have 1: "⟦M; unlabel C'⟧d I " using prems(2) C'(2) 0(1) unfolding unlabel_def by auto

  { fix j p k q
    assume "(j, p) ∈ setopslsst S ∪ set [d←D. d ∉ set Di]"
           "(k, q) ∈ setopslsst S ∪ set [d←D. d ∉ set Di]"
    hence "(j, p) ∈ setopslsst ((i, delete⟨t,s⟩)#S) ∪ set D"
          "(k, q) ∈ setopslsst ((i, delete⟨t,s⟩)#S) ∪ set D"
      by (auto simp add: setopslsst_def)
    hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
  } hence 4: "?R3 S [d←D. d ∉ set Di]" by blast

  have 5: "?R4 S (filter (λd. d ∉ set Di) D)" using prems(4) by force
  have 6: "?R5 S D" using prems(5) by force

  obtain B where B: "B ∈ set (trpc S [d←D. d ∉ set Di])" "⟦M; unlabel B⟧d I"
    using IH[OF C'(1,3) 1 4 5 6] by moura

  have 7: "⟦M; unlabel (c Di)⟧d I" "⟦M; unlabel C'⟧d I"
    using prems(2) C'(2) 0(1) strand_sem_split(3,4)[of M "unlabel (c Di)" "unlabel C'"]
    unfolding c unlabel_def by auto

  { fix j p k q
    assume "(j, p) ∈ {(i, t, s)} ∪ set D"
           "(k, q) ∈ {(i, t, s)} ∪ set D"
    hence "(j, p) ∈ setopslsst ((i, delete⟨t,s⟩)#S) ∪ set D"
          "(k, q) ∈ setopslsst ((i, delete⟨t,s⟩)#S) ∪ set D"
      by (auto simp add: setopslsst_def)
    hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
  } hence "∀(j, p) ∈ {(i, t, s)} ∪ set D.
          ∀(k, q) ∈ {(i, t, s)} ∪ set D.
            (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k"
    by blast
  moreover have "⟦M; unlabel (?cl1 Di)⟧d I" using 7(1) unfolding c unlabel_append by auto
  hence "⟦M; ?cu1 Di⟧d I" by (metis 0(3))
  ultimately have *: "Di ∈ set (subseqs (dbproj i D))"
    using labeled_sat_eqs_subseqs[OF C'(1)] by simp
  hence 8: "d Di@B ∈ set (trpc ((i,delete⟨t,s⟩)#S) D)"
    using B(1) unfolding d unlabel_def by auto

  have "⟦M; unlabel (?cl2 Di)⟧d I" using 7(1) 0(1) unfolding c unlabel_def by auto
  hence "⟦M; ?cu2 Di⟧d I" by (metis 0(4))
  hence "⟦M; ?du2 Di⟧d I" by (metis labeled_sat_ineq_dbproj)
  hence "⟦M; unlabel (?dl2 Di)⟧d I" by (metis 0(6))
  moreover have "⟦M; unlabel (?cl1 Di)⟧d I" using 7(1) unfolding c unlabel_def by auto
  hence "⟦M; unlabel (?dl1 Di)⟧d I" by (metis 0(3,5))
  ultimately have "⟦M; unlabel (d Di)⟧d I" using 0(2) unfolding c d unlabel_def by force
  hence 9: "⟦M; unlabel (d Di@B)⟧d I" using 0(2) B(2) unfolding unlabel_def by auto

  show ?case by (metis 8 9)
next
  case (7 i ac t s S D)
```

```
note prems = "7.prems"
note IH = "7.IH"

obtain C' d where C':
    "C = (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#C'"
    "C' ∈ set (tr'_pc S D)" "d ∈ set D"
  using prems(1) by moura

have 1: "⟦M; unlabel C'⟧_d ℐ " using prems(2) C'(1) by simp

{ fix j p k q
  assume "(j,p) ∈ setops_lsst S ∪ set D"
         "(k,q) ∈ setops_lsst S ∪ set D"
  hence "(j,p) ∈ setops_lsst ((i, InSet ac t s)#S) ∪ set D"
        "(k,q) ∈ setops_lsst ((i, InSet ac t s)#S) ∪ set D"
    by (auto simp add: setops_lsst_def)
  hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
} hence 4: "?R3 S D" by blast

have 5: "?R4 S D" using prems(4) by force
have 6: "?R5 S D" using prems(5) by force

obtain B where B: "B ∈ set (tr_pc S D)" "⟦M; unlabel B⟧_d ℐ"
  using IH[OF C'(2) 1 4 5 6] by moura

have 7: "pair (t,s) · ℐ = pair (snd d) · ℐ" using prems(2) C'(1) by simp

have "(i,t,s) ∈ setops_lsst ((i, InSet ac t s)#S) ∪ set D"
     "(fst d, snd d) ∈ setops_lsst ((i, InSet ac t s)#S) ∪ set D"
  using C'(3) by (auto simp add: setops_lsst_def)
hence "∃δ. Unifier δ (pair (t,s)) (pair (snd d)) ⟹ i = fst d"
  using prems(3) by blast
hence "fst d = i" using 7 by auto
hence 8: "d ∈ set (dbproj i D)" using C'(3) by auto

have 9: "((i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#B) ∈ set (tr_pc ((i,InSet ac t s)#S) D)"
  using B 8 by auto
have 10: "⟦M; unlabel ((i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#B)⟧_d ℐ"
  using B 7 8 by auto

show ?case by (metis 9 10)
next
case (8 i X F F' S D)
note prems = "8.prems"
note IH = "8.IH"

let ?dl = "map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd (dbproj i D)))"
let ?du = "map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' (map snd (dbproj i D)))"

let ?cl = "map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd D))"
let ?cu = "map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' (map snd D))"

define c where c: "c = ?cl"
define d where d: "d = ?dl"

obtain C' where C': "C = c@C'" "C' ∈ set (tr'_pc S D)" using prems(1) c by moura

have 0: "ik_st (unlabel c) = {}" "ik_st (unlabel d) = {}"
        "unlabel ?cl = ?cu" "unlabel ?dl = ?du"
  unfolding c d unlabel_def by force+

have "ik_st (unlabel c) = {}" unfolding c unlabel_def by force
hence 1: "⟦M; unlabel C'⟧_d ℐ " using prems(2) C'(1) unfolding unlabel_def by auto
```

```
      have "setops_lsst S ⊆ setops_lsst ((i, NegChecks X F F')#S)" by (auto simp add: setops_lsst_def)
      hence 4: "?R3 S D" using prems(3) by blast

      have 5: "?R4 S D" using prems(4) by force
      have 6: "?R5 S D" using prems(5) by force

      obtain B where B: "B ∈ set (tr_pc S D)" "⟦M; unlabel B⟧_d ℐ"
        using IH[OF C'(2) 1 4 5 6] by moura

      have 7: "⟦M; unlabel c⟧_d ℐ" "⟦M; unlabel C'⟧_d ℐ"
        using prems(2) C'(1) 0(1) strand_sem_split(3,4)[of M "unlabel c" "unlabel C'"]
        unfolding c unlabel_def by auto

      have 8: "d@B ∈ set (tr_pc ((i,NegChecks X F F')#S) D)"
        using B(1) unfolding d unlabel_def by auto

      have "⟦M; unlabel ?cl⟧_d ℐ" using 7(1) unfolding c unlabel_def by auto
      hence "⟦M; ?cu⟧_d ℐ" by (metis 0(3))
      moreover {
        fix j p k q
        assume "(j, p) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D"
               "(k, q) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D"
        hence "(j, p) ∈ setops_lsst ((i, NegChecks X F F')#S) ∪ set D"
              "(k, q) ∈ setops_lsst ((i, NegChecks X F F')#S) ∪ set D"
          by (auto simp add: setops_lsst_def)
        hence "(∃δ. Unifier δ (pair p) (pair q)) ⟹ j = k" using prems(3) by blast
      } hence "∀(j, p) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D.
              ∀(k, q) ∈ ((λ(t,s). (i,t,s)) ` set F') ∪ set D.
                (∃δ. Unifier δ (pair p) (pair q)) ⟶ j = k"
        by blast
      moreover have "fv_pairs (map snd D) ∩ set X = {}"
        using prems(4) by fastforce
      ultimately have "⟦M; ?du⟧_d ℐ" using labeled_sat_ineq_dbproj_sem_equiv[of i] by simp
      hence "⟦M; unlabel ?dl⟧_d ℐ" by (metis 0(4))
      hence "⟦M; unlabel d⟧_d ℐ" using 0(2) unfolding c d unlabel_def by force
      hence 9: "⟦M; unlabel (d@B)⟧_d ℐ" using 0(2) B(2) unfolding unlabel_def by auto

      show ?case by (metis 8 9)
    qed
  } thus "?Q ⟹ ?P" by metis
qed
```

Part 3

```
private lemma tr'_par_sem_equiv:
  assumes "∀(l,t,s) ∈ set D. (fv t ∪ fv s) ∩ bvars_sst (unlabel A) = {}"
  and "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}" "ground M"
  and "∀(i,p) ∈ setops_lsst A ∪ set D. ∀(j,q) ∈ setops_lsst A ∪ set D.
        (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j" (is "?R A D")
  and ℐ: "interpretation_subst ℐ"
  shows "⟦M; set (unlabel D) ·_pset ℐ; unlabel A⟧_s ℐ ⟷ (∃B ∈ set (tr'_pc A D). ⟦M; unlabel B⟧_d ℐ)"
        (is "?P ⟷ ?Q")
proof -
  have 1: "∀(t,s) ∈ set (unlabel D). (fv t ∪ fv s) ∩ bvars_sst (unlabel A) = {}"
    using assms(1) unfolding unlabel_def by force

  have 2: "∀(i,p) ∈ setops_lsst A ∪ set D. ∀(j,q) ∈ setops_lsst A ∪ set D. p = q ⟶ i = j"
    using assms(4) subst_apply_term_empty by blast

  show ?thesis by (metis tr_sem_equiv'[OF 1 assms(2,3) ℐ] tr'_par_iff_unlabel_tr[OF 2])
qed
```

Part 4

```
lemma tr_par_sem_equiv:
  assumes "∀(l,t,s) ∈ set D. (fv t ∪ fv s) ∩ bvars_sst (unlabel A) = {}"
  and "fv_sst (unlabel A) ∩ bvars_sst (unlabel A) = {}" "ground M"
  and "∀(i,p) ∈ setops_lsst A ∪ set D. ∀(j,q) ∈ setops_lsst A ∪ set D.
        (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j"
  and I: "interpretation_subst I"
  shows "⟦M; set (unlabel D) ·_pset I; unlabel A⟧_s I ⟷ (∃B ∈ set (tr_pc A D). ⟦M; unlabel B⟧_d I)"
  (is "?P ⟷ ?Q")
using tr_par_iff_tr'_par[OF assms(4,1,2), of M I] tr'_par_sem_equiv[OF assms] by metis

end
```

6.2.6 Theorem: The Stateful Compositionality Result, on the Constraint Level

```
theorem par_comp_constr_stateful:
  assumes 𝒜: "par_comp_lsst 𝒜 Sec" "typing_cond_sst (unlabel 𝒜)"
  and ℐ: "ℐ ⊨_s unlabel 𝒜" "interpretation_subst ℐ"
  shows "∃ℐ_τ. interpretation_subst ℐ_τ ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ) ∧ (ℐ_τ ⊨_s unlabel 𝒜) ∧
            ((∀n. ℐ_τ ⊨_s proj_unl n 𝒜) ∨ (∃𝒜'. prefix 𝒜' 𝒜 ∧ (𝒜' leaks Sec under ℐ_τ)))"
proof -
  let ?P = "λn A D.
        ∀(i, p) ∈ setops_lsst (proj n A) ∪ set D.
        ∀(j, q) ∈ setops_lsst (proj n A) ∪ set D.
           (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j"

  have 1: "∀(l, t, t')∈set []. (fv t ∪ fv t') ∩ bvars_sst (unlabel 𝒜) = {}"
          "fv_sst (unlabel 𝒜) ∩ bvars_sst (unlabel 𝒜) = {}" "ground {}"
    using 𝒜(2) unfolding typing_cond_sst_def by simp_all

  have 2: "⋀n. ∀(l, t, t')∈set []. (fv t ∪ fv t') ∩ bvars_sst (proj_unl n 𝒜) = {}"
          "⋀n. fv_sst (proj_unl n 𝒜) ∩ bvars_sst (proj_unl n 𝒜) = {}"
    using 1(1,2) sst_vars_proj_subset[of _ 𝒜] by fast+

  have 3: "⋀n. par_comp_lsst (proj n 𝒜) Sec"
    using par_comp_lsst_proj[OF 𝒜(1)] by metis

  have 4:
      "⟦{}; set (unlabel []) ·_pset ℐ'; unlabel 𝒜⟧_s ℐ' ⟷
        (∃B∈set (tr_pc 𝒜 []). ⟦{}; unlabel B⟧_d ℐ')"
    when ℐ': "interpretation_subst ℐ'" for ℐ'
    using tr_par_sem_equiv[OF 1 _ ℐ'] 𝒜(1)
    unfolding par_comp_lsst_def constr_sem_d_def by auto

  obtain 𝒜' where 𝒜': "𝒜' ∈ set (tr_pc 𝒜 [])" "ℐ ⊨ ⟨unlabel 𝒜'⟩"
    using 4[OF ℐ(2)] ℐ(1) unfolding constr_sem_d_def by moura

  obtain ℐ_τ where ℐ_τ:
      "interpretation_subst ℐ_τ" "wt_subst ℐ_τ" "wf_trms (subst_range ℐ_τ)" "ℐ_τ ⊨ ⟨unlabel 𝒜'⟩"
      "(∀n. (ℐ_τ ⊨ ⟨proj_unl n 𝒜'⟩)) ∨ (∃𝒜''. prefix 𝒜'' 𝒜' ∧ (strand_leaks_lst 𝒜'' Sec ℐ_τ))"
    using par_comp_constr[OF tr_par_preserves_par_comp[OF 𝒜(1) 𝒜'(1)]
                             tr_par_preserves_typing_cond[OF 𝒜 𝒜'(1)]
                             𝒜'(2) ℐ(2)]
    by moura

  have ℐ_τ': "ℐ_τ ⊨_s unlabel 𝒜" using 4[OF ℐ_τ(1)] 𝒜'(1) ℐ_τ(4) unfolding constr_sem_d_def by auto

  show ?thesis
  proof (cases "∀n. (ℐ_τ ⊨ ⟨proj_unl n 𝒜'⟩)")
    case True
    { fix n assume "ℐ_τ ⊨ ⟨proj_unl n 𝒜'⟩"
      hence "⟦{}; {}; unlabel (proj n 𝒜)⟧_s ℐ_τ"
```

```
      using tr_par_proj[OF 𝒜'(1), of n]
            tr_par_sem_equiv[OF 2(1,2) 1(3) _ ℐτ(1), of n] 3(1)
      unfolding par_complsst_def proj_def constr_sem_d_def by force
  } thus ?thesis using True ℐτ(1,2,3) ℐτ' by metis
next
  case False
  then obtain 𝒜''::"('fun,'var,'lbl) labeled_strand" where 𝒜'':
      "prefix 𝒜'' 𝒜'" "strand_leakslst 𝒜'' Sec ℐτ"
    using ℐτ by blast
  moreover {
    fix t l assume *: "⟦{}; unlabel (proj l 𝒜'')@[send⟨t⟩st]⟧d ℐτ"
    have "ℐτ ⊨ ⟨unlabel (proj l 𝒜'')⟩" "ikst (unlabel (proj l 𝒜'')) ·set ℐτ ⊢ t · ℐτ"
      using strand_sem_split(3,4)[OF *] unfolding constr_sem_d_def by auto
  } ultimately have "∃t ∈ Sec - declassifiedlst 𝒜'' ℐτ. ∃l.
          (ℐτ ⊨ ⟨unlabel (proj l 𝒜'')⟩) ∧ ikst (unlabel (proj l 𝒜'')) ·set ℐτ ⊢ t · ℐτ"
    unfolding strand_leakslst_def constr_sem_d_def by metis
  then obtain s m where sm:
      "s ∈ Sec - declassifiedlst 𝒜'' ℐτ"
      "ℐτ ⊨ ⟨unlabel (proj m 𝒜'')⟩"
      "ikst (unlabel (proj m 𝒜'')) ·set ℐτ ⊢ s · ℐτ"
    by moura
```

— We now need to show that there is some prefix B of 𝒜'' that also leaks and where B ∈ set (tr C D) for some prefix C of 𝒜

```
  obtain B::"('fun,'var,'lbl) labeled_strand"
      and C::"('fun,'var,'lbl) labeled_stateful_strand"
    where BC:
      "prefix B 𝒜'" "prefix C 𝒜" "B ∈ set (trpc C [])"
      "ikst (unlabel (proj m B)) ·set ℐτ ⊢ s · ℐτ"
      "prefix B 𝒜''"
    using tr_leaking_prefix_exists[OF 𝒜'(1) 𝒜''(1) sm(3)] prefix_order.order_trans[OF _ 𝒜''(1)]
    by auto
  have "⟦{}; unlabel (proj m B)⟧d ℐτ"
    using sm(2) BC(5) unfolding prefix_def unlabel_def proj_def constr_sem_d_def by auto
  hence BC': "ℐτ ⊨ ⟨proj_unl m B@[send⟨s⟩st]⟩"
    using BC(4) unfolding constr_sem_d_def by auto
  have BC'': "s ∈ Sec - declassifiedlst B ℐτ"
    using BC(5) sm(1) unfolding prefix_def declassifiedlst_def by auto
  have 5: "par_complsst (proj n C) Sec" for n
    using 𝒜(1) BC(2) par_complsst_split(1)[THEN par_complsst_proj]
    unfolding prefix_def by auto
  have "fvsst (unlabel 𝒜) ∩ bvarssst (unlabel 𝒜) = {}"
       "fvsst (unlabel C) ⊆ fvsst (unlabel 𝒜)"
       "bvarssst (unlabel C) ⊆ bvarssst (unlabel 𝒜)"
    using 𝒜(2) BC(2) sst_vars_append_subset(1,2)[of "unlabel C"]
    unfolding typing_condsst_def prefix_def unlabel_def by auto
  hence "fvsst (proj_unl n C) ∩ bvarssst (proj_unl n C) = {}" for n
    using sst_vars_proj_subset[of _ C] sst_vars_proj_subset[of _ 𝒜]
    by blast
  hence 6:
      "∀(l, t, t')∈set []. (fv t ∪ fv t') ∩ bvarssst (proj_unl n C) = {}"
      "fvsst (proj_unl n C) ∩ bvarssst (proj_unl n C) = {}"
      "ground {}"
    for n
    using 2 by auto
  have 7: "?P n C []" for n using 5 unfolding par_complsst_def by simp
  have "s · ℐτ = s" using ℐτ(1) BC'' 𝒜(1) unfolding par_complsst_def by auto
  hence "∃n. (ℐτ ⊨s proj_unl n C) ∧ iksst (proj_unl n C) ·set ℐτ ⊢ s · ℐτ"
    using tr_par_proj[OF BC(3), of m] BC'(1)
          tr_par_sem_equiv[OF 6 7 ℐτ(1), of m]
          tr_par_deduct_iff[OF tr_par_proj(1)[OF BC(3)], of ℐτ m s]
    unfolding proj_def constr_sem_d_def by auto
  hence "∃n. ℐτ ⊨s (proj_unl n C@[Send s])" using strand_sem_append_stateful by simp
```

    moreover have "s ∈ Sec - declassified$_{lsst}$ C $\mathcal{I}_\tau$" **by** (metis tr_par_declassified_eq BC(3) BC'')
    ultimately show ?thesis **using** $\mathcal{I}_\tau$(1,2,3) $\mathcal{I}_\tau$' BC(2) **unfolding** strand_leaks$_{lsst}$_def **by** metis
  qed
qed

6.2.7 Theorem: The Stateful Compositionality Result, on the Protocol Level

abbreviation wf$_{lsst}$ **where**
  "wf$_{lsst}$ V $\mathcal{A}$ ≡ wf'$_{sst}$ V (unlabel $\mathcal{A}$)"

We state our result on the level of protocol traces (i.e., the constraints reachable in a symbolic execution of the actual protocol). Hence, we do not need to convert protocol strands to intruder constraints in the following well-formedness definitions.

definition wf$_{lssts}$::"('fun,'var,'lbl) labeled_stateful_strand set ⇒ bool" **where**
  "wf$_{lssts}$ $\mathcal{S}$ ≡ (∀$\mathcal{A}$ ∈ $\mathcal{S}$. wf$_{lsst}$ {} $\mathcal{A}$) ∧ (∀$\mathcal{A}$ ∈ $\mathcal{S}$. ∀$\mathcal{A}$' ∈ $\mathcal{S}$. fv$_{lsst}$ $\mathcal{A}$ ∩ bvars$_{lsst}$ $\mathcal{A}$' = {})"

definition wf$_{lssts}$'::
  "('fun,'var,'lbl) labeled_stateful_strand set ⇒ ('fun,'var,'lbl) labeled_stateful_strand ⇒ bool"
where
  "wf$_{lssts}$' $\mathcal{S}$ $\mathcal{A}$ ≡ (∀$\mathcal{A}$' ∈ $\mathcal{S}$. wf'$_{sst}$ (wfrestrictedvars$_{lsst}$ $\mathcal{A}$) (unlabel $\mathcal{A}$')) ∧
                (∀$\mathcal{A}$' ∈ $\mathcal{S}$. ∀$\mathcal{A}$'' ∈ $\mathcal{S}$. fv$_{lsst}$ $\mathcal{A}$' ∩ bvars$_{lsst}$ $\mathcal{A}$'' = {}) ∧
                (∀$\mathcal{A}$' ∈ $\mathcal{S}$. fv$_{lsst}$ $\mathcal{A}$' ∩ bvars$_{lsst}$ $\mathcal{A}$ = {}) ∧
                (∀$\mathcal{A}$' ∈ $\mathcal{S}$. fv$_{lsst}$ $\mathcal{A}$ ∩ bvars$_{lsst}$ $\mathcal{A}$' = {})"

definition typing_cond_prot_stateful **where**
  "typing_cond_prot_stateful $\mathcal{P}$ ≡
    wf$_{lssts}$ $\mathcal{P}$ ∧
    tfr$_{set}$ (⋃(trms$_{lsst}$ ` $\mathcal{P}$) ∪ pair ` ⋃(setops$_{sst}$ ` unlabel ` $\mathcal{P}$)) ∧
    wf$_{trms}$ (⋃(trms$_{lsst}$ ` $\mathcal{P}$)) ∧
    (∀S ∈ $\mathcal{P}$. list_all tfr$_{sstp}$ (unlabel S))"

definition par_comp_prot_stateful **where**
  "par_comp_prot_stateful $\mathcal{P}$ Sec ≡
    (∀l1 l2. l1 ≠ l2 ⟶
      GSMP_disjoint (⋃$\mathcal{A}$ ∈ $\mathcal{P}$. trms$_{sst}$ (proj_unl l1 $\mathcal{A}$) ∪ pair ` setops$_{sst}$ (proj_unl l1 $\mathcal{A}$))
                    (⋃$\mathcal{A}$ ∈ $\mathcal{P}$. trms$_{sst}$ (proj_unl l2 $\mathcal{A}$) ∪ pair ` setops$_{sst}$ (proj_unl l2 $\mathcal{A}$)) Sec) ∧
    ground Sec ∧ (∀s ∈ Sec. ∀s' ∈ subterms s. {} ⊢$_c$ s' ∨ s' ∈ Sec) ∧
    (∀(i,p) ∈ ⋃$\mathcal{A}$ ∈ $\mathcal{P}$. setops$_{lsst}$ $\mathcal{A}$. ∀(j,q) ∈ ⋃$\mathcal{A}$ ∈ $\mathcal{P}$. setops$_{lsst}$ $\mathcal{A}$.
      (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j) ∧
    typing_cond_prot_stateful $\mathcal{P}$"

definition component_secure_prot_stateful **where**
  "component_secure_prot_stateful n P Sec attack ≡
    (∀$\mathcal{A}$ ∈ P. suffix [(ln n, Send (Fun attack []))] $\mathcal{A}$ ⟶
     (∀$\mathcal{I}_\tau$. (interpretation$_{subst}$ $\mathcal{I}_\tau$ ∧ wt$_{subst}$ $\mathcal{I}_\tau$ ∧ wf$_{trms}$ (subst_range $\mathcal{I}_\tau$)) ⟶
          ¬($\mathcal{I}_\tau$ ⊨$_s$ (proj_unl n $\mathcal{A}$)) ∧
          (∀$\mathcal{A}$'. prefix $\mathcal{A}$' $\mathcal{A}$ ⟶
                  (∀t ∈ Sec-declassified$_{lsst}$ $\mathcal{A}$' $\mathcal{I}_\tau$. ¬($\mathcal{I}_\tau$ ⊨$_s$ (proj_unl n $\mathcal{A}$'@[Send t]))))))"

definition component_leaks_stateful **where**
  "component_leaks_stateful n $\mathcal{A}$ Sec ≡
    (∃$\mathcal{A}$' $\mathcal{I}_\tau$. interpretation$_{subst}$ $\mathcal{I}_\tau$ ∧ wt$_{subst}$ $\mathcal{I}_\tau$ ∧ wf$_{trms}$ (subst_range $\mathcal{I}_\tau$) ∧ prefix $\mathcal{A}$' $\mathcal{A}$ ∧
          (∃t ∈ Sec - declassified$_{lsst}$ $\mathcal{A}$' $\mathcal{I}_\tau$. ($\mathcal{I}_\tau$ ⊨$_s$ (proj_unl n $\mathcal{A}$'@[Send t]))))"

definition unsat_stateful **where**
  "unsat_stateful $\mathcal{A}$ ≡ (∀$\mathcal{I}$. interpretation$_{subst}$ $\mathcal{I}$ ⟶ ¬($\mathcal{I}$ ⊨$_s$ unlabel $\mathcal{A}$))"

lemma wf$_{lssts}$_eqs_wf$_{lssts}$'[simp]: "wf$_{lssts}$ S = wf$_{lssts}$' S []"
unfolding wf$_{lssts}$_def wf$_{lssts}$'_def unlabel_def wfrestrictedvars$_{sst}$_def **by** simp

lemma par_comp_prot_impl_par_comp_stateful:
  assumes "par_comp_prot_stateful $\mathcal{P}$ Sec" "$\mathcal{A}$ ∈ $\mathcal{P}$"
  shows "par_comp$_{lsst}$ $\mathcal{A}$ Sec"

```
proof -
  have *:
      "∀l1 l2. l1 ≠ l2 ⟶
          GSMP_disjoint (⋃𝒜 ∈ 𝒫. trms_sst (proj_unl l1 𝒜) ∪ pair ` setops_sst (proj_unl l1 𝒜))
                        (⋃𝒜 ∈ 𝒫. trms_sst (proj_unl l2 𝒜) ∪ pair ` setops_sst (proj_unl l2 𝒜)) Sec"
    using assms(1) unfolding par_comp_prot_stateful_def by argo
  { fix l1 l2::'lbl assume **: "l1 ≠ l2"
    hence ***:
        "GSMP_disjoint (⋃𝒜 ∈ 𝒫. trms_sst (proj_unl l1 𝒜) ∪ pair ` setops_sst (proj_unl l1 𝒜))
                       (⋃𝒜 ∈ 𝒫. trms_sst (proj_unl l2 𝒜) ∪ pair ` setops_sst (proj_unl l2 𝒜)) Sec"
      using * by auto
    have "GSMP_disjoint (trms_sst (proj_unl l1 𝒜) ∪ pair ` setops_sst (proj_unl l1 𝒜))
                        (trms_sst (proj_unl l2 𝒜) ∪ pair ` setops_sst (proj_unl l2 𝒜)) Sec"
      using GSMP_disjoint_subset[OF ***] assms(2) by auto
  } hence "∀l1 l2. l1 ≠ l2 ⟶
             GSMP_disjoint (trms_sst (proj_unl l1 𝒜) ∪ pair ` setops_sst (proj_unl l1 𝒜))
                           (trms_sst (proj_unl l2 𝒜) ∪ pair ` setops_sst (proj_unl l2 𝒜)) Sec"
    by metis
  moreover have "∀(i,p) ∈ setops_lsst 𝒜. ∀(j,q) ∈ setops_lsst 𝒜.
                  (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j"
    using assms(1,2) unfolding par_comp_prot_stateful_def by blast
  ultimately show ?thesis
    using assms
    unfolding par_comp_prot_stateful_def par_comp_lsst_def
    by fast
qed

lemma typing_cond_prot_impl_typing_cond_stateful:
  assumes "typing_cond_prot_stateful 𝒫" "𝒜 ∈ 𝒫"
  shows "typing_cond_sst (unlabel 𝒜)"
proof -
  have 1: "wf'_sst {} (unlabel 𝒜)" "fv_lsst 𝒜 ∩ bvars_lsst 𝒜 = {}"
    using assms unfolding typing_cond_prot_stateful_def wf_lssts_def by auto

  have "tfr_set (⋃(trms_lsst ` 𝒫) ∪ pair ` ⋃(setops_sst ` unlabel ` 𝒫))"
       "wf_trms (⋃(trms_lsst ` 𝒫))"
       "trms_lsst 𝒜 ⊆ ⋃(trms_lsst ` 𝒫)"
       "SMP (trms_lsst 𝒜 ∪ pair ` setops_sst (unlabel 𝒜)) - Var`𝒱 ⊆
        SMP (⋃(trms_lsst ` 𝒫) ∪ pair ` ⋃(setops_sst ` unlabel ` 𝒫)) - Var`𝒱"
    using assms SMP_mono[of "trms_lsst 𝒜 ∪ pair ` setops_sst (unlabel 𝒜)"
                            "⋃(trms_lsst ` 𝒫) ∪ pair ` ⋃(setops_sst ` unlabel ` 𝒫)"]
    unfolding typing_cond_prot_stateful_def
    by (metis, metis, auto)
  hence 2: "tfr_set (trms_lsst 𝒜 ∪ pair ` setops_sst (unlabel 𝒜))" and 3: "wf_trms (trms_lsst 𝒜)"
    unfolding tfr_set_def by (meson subsetD)+

  have 4: "list_all tfr_sstp (unlabel 𝒜)" using assms unfolding typing_cond_prot_stateful_def by auto

  show ?thesis using 1 2 3 4 unfolding typing_cond_sst_def tfr_sst_def by blast
qed

theorem par_comp_constr_prot_stateful:
  assumes P: "P = composed_prot Pi" "par_comp_prot_stateful P Sec" "∀n. component_prot n (Pi n)"
  and left_secure: "component_secure_prot_stateful n (Pi n) Sec attack"
  shows "∀𝒜 ∈ P. suffix [(ln n, Send (Fun attack []))] 𝒜 ⟶
                 unsat_stateful 𝒜 ∨ (∃m. n ≠ m ∧ component_leaks_stateful m 𝒜 Sec)"
proof -
  { fix 𝒜 𝒜' assume 𝒜: "𝒜 = 𝒜'@[(ln n, Send (Fun attack []))]" "𝒜 ∈ P"
    let ?P = "∃𝒜' ℐ_τ. interpretation_subst ℐ_τ ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ) ∧ prefix 𝒜' 𝒜
∧
                 (∃t ∈ Sec-declassified_lsst 𝒜' ℐ_τ. ∃m. n ≠ m ∧ (ℐ_τ ⊨_s (proj_unl m 𝒜'@[Send
t])))"
    have tcp: "typing_cond_prot_stateful P" using P(2) unfolding par_comp_prot_stateful_def by simp
```

```
    have par_comp: "par_comp_lsst 𝒜 Sec" "typing_cond_sst (unlabel 𝒜)"
      using par_comp_prot_impl_par_comp_stateful[OF P(2) 𝒜(2)]
            typing_cond_prot_impl_typing_cond_stateful[OF tcp 𝒜(2)]
      by metis+

    have "unlabel (proj n 𝒜) = proj_unl n 𝒜" "proj_unl n 𝒜 = proj_unl n (proj n 𝒜)"
         "⋀A. A ∈ Pi n ⟹ proj n A = A"
         "proj n 𝒜 = (proj n 𝒜')@[(ln n, Send (Fun attack []))]"
      using P(1,3) 𝒜 by (auto simp add: proj_def unlabel_def component_prot_def composed_prot_def)
    moreover have "proj n 𝒜 ∈ Pi n"
      using P(1) 𝒜 unfolding composed_prot_def by blast
    moreover {
      fix A assume "prefix A 𝒜"
      hence *: "prefix (proj n A) (proj n 𝒜)" unfolding proj_def prefix_def by force
      hence "proj_unl n A = proj_unl n (proj n A)"
            "∀I. declassified_lsst A I = declassified_lsst (proj n A) I"
        unfolding proj_def declassified_lsst_def by auto
      hence "∃B. prefix B (proj n 𝒜) ∧ proj_unl n A = proj_unl n B ∧
                (∀I. declassified_lsst A I = declassified_lsst B I)"
        using * by metis
    }
    ultimately have *:
        "∀ℐ_τ. interpretation_subst ℐ_τ ∧ wt_subst ℐ_τ ∧ wf_trms (subst_range ℐ_τ) ⟶
                ¬(ℐ_τ ⊨_s (proj_unl n 𝒜)) ∧ (∀𝒜'. prefix 𝒜' 𝒜 ⟶
                     (∀t ∈ Sec - declassified_lsst 𝒜' ℐ_τ. ¬(ℐ_τ ⊨_s (proj_unl n 𝒜'@[Send t]))))"
      using left_secure
      unfolding component_secure_prot_stateful_def composed_prot_def suffix_def
      by metis
    { fix ℐ assume ℐ: "interpretation_subst ℐ" "ℐ ⊨_s unlabel 𝒜"
      obtain ℐ_τ where ℐ_τ:
          "interpretation_subst ℐ_τ" "wt_subst ℐ_τ" "wf_trms (subst_range ℐ_τ)"
          "∃𝒜'. prefix 𝒜' 𝒜 ∧ (𝒜' leaks Sec under ℐ_τ)"
        using par_comp_constr_stateful[OF par_comp ℐ(2,1)] * by moura
      hence "∃𝒜'. prefix 𝒜' 𝒜 ∧ (∃t ∈ Sec - declassified_lsst 𝒜' ℐ_τ. ∃m.
                 n ≠ m ∧ (ℐ_τ ⊨_s (proj_unl m 𝒜'@[Send t])))"
        using ℐ_τ(4) * unfolding strand_leaks_lsst_def by metis
      hence ?P using ℐ_τ(1,2,3) by auto
    } hence "unsat_stateful 𝒜 ∨ (∃m. n ≠ m ∧ component_leaks_stateful m 𝒜 Sec)"
      by (metis unsat_stateful_def component_leaks_stateful_def)
  } thus ?thesis unfolding suffix_def by metis
qed

end
```

6.2.8 Automated Compositionality Conditions

```
definition comp_GSMP_disjoint where
  "comp_GSMP_disjoint public arity Ana Γ A' B' A B C ≡
    let Bδ = B ·_list var_rename (max_var_set (fv_set (set A)))
    in has_all_wt_instances_of Γ (set A') (set A) ∧
       has_all_wt_instances_of Γ (set B') (set Bδ) ∧
       finite_SMP_representation arity Ana Γ A ∧
       finite_SMP_representation arity Ana Γ Bδ ∧
       (∀t ∈ set A. ∀s ∈ set Bδ. Γ t = Γ s ∧ mgu t s ≠ None ⟶
         (intruder_synth' public arity {} t ∧ intruder_synth' public arity {} s) ∨
         (∃u ∈ set C. is_wt_instance_of_cond Γ t u) ∧ (∃u ∈ set C. is_wt_instance_of_cond Γ s u))"

definition comp_par_comp_lsst where
  "comp_par_comp_lsst public arity Ana Γ pair_fun A M C ≡
  let L = remdups (map (the_LabelN ∘ fst) (filter (Not ∘ is_LabelS) A));
      MP0 = λB. remdups (trms_list_sst B@map (pair' pair_fun) (setops_list_sst B));
      pr = λl. MP0 (proj_unl l A)
  in length L > 1 ∧
```

```
     list_all (wf_trm' arity) (MP0 (unlabel A)) ∧
     list_all (wf_trm' arity) C ∧
     has_all_wt_instances_of Γ (subterms_set (set C)) (set C) ∧
     is_TComp_var_instance_closed Γ C ∧
     (∀i ∈ set L. ∀j ∈ set L. i ≠ j ⟶
        comp_GSMP_disjoint public arity Ana Γ (pr i) (pr j) (M i) (M j) C) ∧
     (∀(i,p) ∈ setops_lsst A. ∀(j,q) ∈ setops_lsst A. i ≠ j ⟶
        (let s = pair' pair_fun p; t = pair' pair_fun q
         in mgu s (t · var_rename (max_var s)) = None))"

locale labeled_stateful_typed_model' =
  stateful_typed_model' arity public Ana Γ Pair
+ labeled_typed_model' arity public Ana Γ label_witness1 label_witness2
  for arity::"'fun ⇒ nat"
  and public::"'fun ⇒ bool"
  and Ana::"('fun,(('fun,'atom::finite) term_type × nat)) term
          ⇒ (('fun,(('fun,'atom) term_type × nat)) term list
             × ('fun,(('fun,'atom) term_type × nat)) term list)"
  and Γ::"('fun,(('fun,'atom) term_type × nat)) term ⇒ ('fun,'atom) term_type"
  and Pair::"'fun"
  and label_witness1::"'lbl"
  and label_witness2::"'lbl"
begin

sublocale labeled_stateful_typed_model
by unfold_locales

lemma GSMP_disjoint_if_comp_GSMP_disjoint:
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
  assumes AB'_wf: "list_all (wf_trm' arity) A'" "list_all (wf_trm' arity) B'"
    and C_wf: "list_all (wf_trm' arity) C"
    and AB'_disj: "comp_GSMP_disjoint public arity Ana Γ A' B' A B C"
  shows "GSMP_disjoint (set A') (set B') ((f (set C)) - {m. {} ⊢_c m})"
using GSMP_disjointI[of A' B' A B] AB'_wf AB'_disj C_wf
unfolding comp_GSMP_disjoint_def f_def wf_trm_code list_all_iff Let_def by fast

lemma par_comp_lsst_if_comp_par_comp_lsst:
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
  assumes A: "comp_par_comp_lsst public arity Ana Γ Pair A M C"
  shows "par_comp_lsst A ((f (set C)) - {m. {} ⊢_c m})"
proof (unfold par_comp_lsst_def; intro conjI)
  let ?Sec = "(f (set C)) - {m. {} ⊢_c m}"
  let ?L = "remdups (map (the_LabelN ∘ fst) (filter (Not ∘ is_LabelS) A))"
  let ?N1 = "λB. remdups (trms_list_sst B@map (pair' Pair) (setops_list_sst B))"
  let ?N2 = "λB. trms_sst B ∪ pair ` setops_sst B"
  let ?pr = "λl. ?N1 (proj_unl l A)"
  let ?α = "λp. var_rename (max_var (pair p))"

  have 0:
      "length ?L > 1"
      "list_all (wf_trm' arity) (?N1 (unlabel A))"
      "list_all (wf_trm' arity) C"
      "has_all_wt_instances_of Γ (subterms_set (set C)) (set C)"
      "is_TComp_var_instance_closed Γ C"
      "∀i ∈ set ?L. ∀j ∈ set ?L. i ≠ j ⟶
        comp_GSMP_disjoint public arity Ana Γ (?pr i) (?pr j) (M i) (M j) C"
      "∀(i,p) ∈ setops_lsst A. ∀(j,q) ∈ setops_lsst A. i ≠ j ⟶ mgu (pair p) (pair q · ?α p) = None"
    using A unfolding comp_par_comp_lsst_def pair_code by meson+

  have L_in_iff: "l ∈ set ?L ⟷ (∃a ∈ set A. is_LabelN l a)" for l by force

  have A_wf_trms: "wf_trms (trms_lsst A ∪ pair ` setops_sst (unlabel A))"
    using 0(2)
```

```
    unfolding pair_code wf_trm_code list_all_iff trms_list_sst_is_trms_sst setops_list_sst_is_setops_sst
    by auto
  hence A_proj_wf_trms: "wf_trms (trms_lsst (proj l A) ∪ pair ‘ setops_sst (proj_unl l A))" for l
    using trms_sst_proj_subset(1)[of l A] setops_sst_proj_subset(1)[of l A] by blast
  hence A_proj_wf_trms': "list_all (wf_trm' arity) (?N1 (proj_unl l A))" for l
    unfolding pair_code wf_trm_code list_all_iff trms_list_sst_is_trms_sst setops_list_sst_is_setops_sst
    by auto

  note C_wf_trms = 0(3)[unfolded list_all_iff wf_trm_code[symmetric]]

  note 1 = has_all_wt_instances_ofD'[OF wf_trms_subterms[OF C_wf_trms] C_wf_trms 0(4)]

  have 2: "GSMP (?N2 (proj_unl l A)) ⊆ GSMP (?N2 (proj_unl l' A))" when "l ∉ set ?L" for l l'
    using that L_in_iff GSMP_mono[of "?N2 (proj_unl l A)" "?N2 (proj_unl l' A)"]
          trms_sst_unlabel_subset_if_no_label[of l A]
          setops_sst_unlabel_subset_if_no_label[of l A]
    unfolding list_ex_iff by fast

  have 3: "GSMP_disjoint (?N2 (proj_unl l1 A)) (?N2 (proj_unl l2 A)) ?Sec"
    when "l1 ∈ set ?L" "l2 ∈ set ?L" "l1 ≠ l2" for l1 l2
  proof -
    have "GSMP_disjoint (set (?N1 (proj_unl l1 A))) (set (?N1 (proj_unl l2 A))) ?Sec"
      using 0(6) that
            GSMP_disjoint_if_comp_GSMP_disjoint[
              OF A_proj_wf_trms'[of l1] A_proj_wf_trms'[of l2] 0(3),
              of "M l1" "M l2"]
      unfolding f_def by blast
    thus ?thesis
      unfolding pair_code trms_list_sst_is_trms_sst setops_list_sst_is_setops_sst
      by simp
  qed

  obtain a1 a2 where a: "a1 ∈ set ?L" "a2 ∈ set ?L" "a1 ≠ a2"
    using remdups_ex2[OF 0(1)] by moura

  show "ground ?Sec" unfolding f_def by fastforce

  { fix i p j q
    assume p: "(i,p) ∈ setops_lsst A" and q: "(j,q) ∈ setops_lsst A"
      and pq: "∃δ. Unifier δ (pair p) (pair q)"

    have "∃δ. Unifier δ (pair p) (pair q · ?α p)"
      using pq vars_term_disjoint_imp_unifier[OF var_rename_fv_disjoint[of "pair p"], of _ "pair q"]
      by (metis (no_types, lifting) subst_subst_compose var_rename_inv_comp)
    hence "i = j" using 0(7) mgu_None_is_subst_neq[of "pair p" "pair q · ?α p"] p q by fast
  } thus "∀(i,p) ∈ setops_lsst A. ∀(j,q) ∈ setops_lsst A. (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j"
    by blast

  show "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (?N2 (proj_unl l1 A)) (?N2 (proj_unl l2 A)) ?Sec"
    using 2 3 3[OF a] unfolding GSMP_disjoint_def by blast

  show "∀s ∈ ?Sec. ∀s' ∈ subterms s. {} ⊢_c s' ∨ s' ∈ ?Sec"
  proof (intro ballI)
    fix s s'
    assume s: "s ∈ ?Sec" and s': "s' ⊑ s"
    then obtain t δ where t: "t ∈ set C" "s = t · δ" "fv s = {}" "¬{} ⊢_c s"
        and δ: "wt_subst δ" "wf_trms (subst_range δ)"
      unfolding f_def by blast

    obtain m ϑ where m: "m ∈ set C" "s' = m · ϑ" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
      using TComp_var_and_subterm_instance_closed_has_subterms_instances[
              OF 0(5,4) C_wf_trms in_subterms_Union[OF t(1)] s'[unfolded t(2)] δ]
      by blast
```

```
    thus "{} ⊢_c s' ∨ s' ∈ ?Sec"
      using ground_subterm[OF t(3) s']
      unfolding f_def by blast
  qed
qed

lemma par_comp_lsst_if_comp_par_comp_lsst':
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
  assumes a: "comp_par_comp_lsst public arity Ana Γ Pair A M C"
    and B: "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·_lsstp δ ∧ wt_subst δ ∧ wf_trms (subst_range δ)"
      (is "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·_lsstp δ ∧ ?D δ")
  shows "par_comp_lsst B ((f (set C)) - {m. {} ⊢_c m})"
proof (unfold par_comp_lsst_def; intro conjI)
  define N1 where "N1 ≡ λB::('fun, ('fun,'atom) term_type × nat) stateful_strand.
    remdups (trms_list_sst B@map (pair' Pair) (setops_list_sst B))"

  define N2 where "N2 ≡ λB::('fun, ('fun,'atom) term_type × nat) stateful_strand.
    trms_sst B ∪ pair ` setops_sst B"

  define L where "L ≡ λA::('fun, ('fun,'atom) term_type × nat, 'lbl) labeled_stateful_strand.
    remdups (map (the_LabelN ∘ fst) (filter (Not ∘ is_LabelS) A))"

  define α where "α ≡ λp. var_rename (max_var (pair p::('fun, ('fun,'atom) term_type × nat) term))
    ::('fun, ('fun,'atom) term_type × nat) subst"

  let ?Sec = "(f (set C)) - {m. {} ⊢_c m}"

  have 0:
      "length (L A) > 1"
      "list_all (wf_trm' arity) (N1 (unlabel A))"
      "list_all (wf_trm' arity) C"
      "has_all_wt_instances_of Γ (subterms_set (set C)) (set C)"
      "is_TComp_var_instance_closed Γ C"
      "∀i ∈ set (L A). ∀j ∈ set (L A). i ≠ j ⟶
        comp_GSMP_disjoint public arity Ana Γ (N1 (proj_unl i A)) (N1 (proj_unl j A)) (M i) (M j) C"
      "∀(i,p) ∈ setops_lsst A. ∀(j,q) ∈ setops_lsst A. i ≠ j ⟶ mgu (pair p) (pair q · α p) = None"
    using a unfolding comp_par_comp_lsst_def pair_code L_def N1_def α_def by meson+

  note 1 = trms_sst_proj_subset(1) setops_sst_proj_subset(1)

  have N1_iff_N2: "set (N1 A) = N2 A" for A
    unfolding pair_code trms_list_sst_is_trms_sst setops_list_sst_is_setops_sst N1_def N2_def by simp

  have N2_proj_subset: "N2 (proj_unl l A) ⊆ N2 (unlabel A)"
    for l::'lbl and A::"('fun, ('fun,'atom) term_type × nat, 'lbl) labeled_stateful_strand"
    using 1(1)[of l A] image_mono[OF 1(2)[of l A], of pair] unfolding N2_def by blast

  have L_in_iff: "l ∈ set (L A) ⟷ (∃a ∈ set A. is_LabelN l a)" for l A
    unfolding L_def by force

  have L_B_subset_A: "l ∈ set (L A)" when l: "l ∈ set (L B)" for l
    using L_in_iff[of l B] L_in_iff[of l A] B l by fastforce

  note B_setops = setops_lsst_wt_instance_ex[OF B]

  have B_proj: "∀b ∈ set (proj l B). ∃a ∈ set (proj l A). ∃δ. b = a ·_lsstp δ ∧ ?D δ" for l
    using proj_instance_ex[OF B] by fast

  have B': "∀t ∈ N2 (unlabel B). ∃s ∈ N2 (unlabel A). ∃δ. t = s · δ ∧ ?D δ"
    using trms_sst_setops_sst_wt_instance_ex[OF B] unfolding N2_def by blast

  have B'_proj: "∀t ∈ N2 (proj_unl l B). ∃s ∈ N2 (proj_unl l A). ∃δ. t = s · δ ∧ ?D δ" for l
    using trms_sst_setops_sst_wt_instance_ex[OF B_proj] unfolding N2_def by presburger
```

```
have A_wf_trms: "wf_trms (N2 (unlabel A))"
  using N1_iff_N2[of "unlabel A"] 0(2) unfolding wf_trm_code list_all_iff by auto
hence A_proj_wf_trms: "wf_trms (N2 (proj_unl l A))" for l
  using 1[of l] unfolding N2_def by blast
hence A_proj_wf_trms': "list_all (wf_trm' arity) (N1 (proj_unl l A))" for l
  using N1_iff_N2[of "proj_unl l A"] unfolding wf_trm_code list_all_iff by presburger

note C_wf_trms = 0(3)[unfolded list_all_iff wf_trm_code[symmetric]]

have 2: "GSMP (N2 (proj_unl l A)) ⊆ GSMP (N2 (proj_unl l' A))"
  when "l ∉ set (L A)" for l l'
    and A::"('fun, ('fun,'atom) term_type × nat, 'lbl) labeled_stateful_strand"
  using that L_in_iff[of _ A] GSMP_mono[of "N2 (proj_unl l A)" "N2 (proj_unl l' A)"]
        trms_sst_unlabel_subset_if_no_label[of l A]
        setops_sst_unlabel_subset_if_no_label[of l A]
  unfolding list_ex_iff N2_def by fast

have 3: "GSMP (N2 (proj_unl l B)) ⊆ GSMP (N2 (proj_unl l A))" (is "?X ⊆ ?Y") for l
proof
  fix t assume "t ∈ ?X"
  hence t: "t ∈ SMP (N2 (proj_unl l B))" "fv t = {}" unfolding GSMP_def by simp_all
  have "t ∈ SMP (N2 (proj_unl l A))"
    using t(1) B'_proj[of l] SMP_wt_instances_subset[of "N2 (proj_unl l B)" "N2 (proj_unl l A)"]
    by metis
  thus "t ∈ ?Y" using t(2) unfolding GSMP_def by fast
qed

have "GSMP_disjoint (N2 (proj_unl l1 A)) (N2 (proj_unl l2 A)) ?Sec"
  when "l1 ∈ set (L A)" "l2 ∈ set (L A)" "l1 ≠ l2" for l1 l2
proof -
  have "GSMP_disjoint (set (N1 (proj_unl l1 A))) (set (N1 (proj_unl l2 A))) ?Sec"
    using 0(6) that
          GSMP_disjoint_if_comp_GSMP_disjoint[
            OF A_proj_wf_trms'[of l1] A_proj_wf_trms'[of l2] 0(3),
            of "M l1" "M l2"]
    unfolding f_def by blast
  thus ?thesis using N1_iff_N2 by simp
qed
hence 4: "GSMP_disjoint (N2 (proj_unl l1 B)) (N2 (proj_unl l2 B)) ?Sec"
  when "l1 ∈ set (L A)" "l2 ∈ set (L A)" "l1 ≠ l2" for l1 l2
  using that 3 unfolding GSMP_disjoint_def by blast

{ fix i p j q
  assume p: "(i,p) ∈ setops_lsst B" and q: "(j,q) ∈ setops_lsst B"
    and pq: "∃δ. Unifier δ (pair p) (pair q)"

  obtain p' δp where p': "(i,p') ∈ setops_lsst A" "p = p' ·_p δp" "pair p = pair p' · δp"
    using p B_setops unfolding pair_def by auto

  obtain q' δq where q': "(j,q') ∈ setops_lsst A" "q = q' ·_p δq" "pair q = pair q' · δq"
    using q B_setops unfolding pair_def by auto

  obtain ϑ where "Unifier ϑ (pair p) (pair q)" using pq by blast
  hence "∃δ. Unifier δ (pair p') (pair q' · α p')"
    using p'(3) q'(3) var_rename_inv_comp[of "pair q'"] subst_subst_compose
          vars_term_disjoint_imp_unifier[
            OF var_rename_fv_disjoint[of "pair p'"],
            of "δp ∘_s ϑ" "pair q'" "var_rename_inv (max_var_set (fv (pair p'))) ∘_s δq ∘_s ϑ"]
    unfolding α_def by fastforce
  hence "i = j"
    using mgu_None_is_subst_neq[of "pair p'" "pair q' · α p'"] p'(1) q'(1) 0(7)
    unfolding α_def by fast
```

```
  } thus "∀(i,p) ∈ setops_lsst B. ∀(j,q) ∈ setops_lsst B. (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j"
    by blast

  obtain a1 a2 where a: "a1 ∈ set (L A)" "a2 ∈ set (L A)" "a1 ≠ a2"
    using remdups_ex2[OF 0(1)[unfolded L_def]] unfolding L_def by moura

  show "∀l1 l2. l1 ≠ l2 ⟶ GSMP_disjoint (N2 (proj_unl l1 B)) (N2 (proj_unl l2 B)) ?Sec"
    using 2[of _ B] 4 4[OF a] L_B_subset_A unfolding GSMP_disjoint_def by blast

  show "ground ?Sec" unfolding f_def by fastforce

  show "∀s ∈ ?Sec. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ ?Sec"
  proof (intro ballI)
    fix s s'
    assume s: "s ∈ ?Sec" and s': "s' ⊑ s"
    then obtain t δ where t: "t ∈ set C" "s = t · δ" "fv s = {}" "¬{} ⊢c s"
        and δ: "wt_subst δ" "wf_trms (subst_range δ)"
      unfolding f_def by blast

    obtain m ϑ where m: "m ∈ set C" "s' = m · ϑ" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)"
      using TComp_var_and_subterm_instance_closed_has_subterms_instances[
              OF 0(5,4) C_wf_trms in_subterms_Union[OF t(1)] s'[unfolded t(2)] δ]
      by blast
    thus "{} ⊢c s' ∨ s' ∈ ?Sec"
      using ground_subterm[OF t(3) s']
      unfolding f_def by blast
  qed
qed

end

end
```

7 Examples

In this chapter, we present two examples illustrating our results: In section 7.1 we show that the TLS example from [2] is type-flaw resistant. In section 7.2 we show that the keyserver examples from [3, 4] are also type-flaw resistant and that the steps of the composed keyserver protocol from [4] satisfy our conditions for protocol composition.

7.1 Proving Type-Flaw Resistance of the TLS Handshake Protocol (Example_TLS)

```
theory Example_TLS
imports "../Typed_Model"
begin

declare [[code_timing]]
```

7.1.1 TLS example: Datatypes and functions setup

```
datatype ex_atom = PrivKey | SymKey | PubConst | Agent | Nonce | Bot

datatype ex_fun =
  clientHello | clientKeyExchange | clientFinished
| serverHello | serverCert | serverHelloDone
| finished | changeCipher | x509 | prfun | master | pmsForm
| sign | hash | crypt | pub | concat | privkey nat
| pubconst ex_atom nat

type_synonym ex_type = "(ex_fun, ex_atom) term_type"
type_synonym ex_var = "ex_type × nat"

instance ex_atom::finite
proof
  let ?S = "UNIV::ex_atom set"
  have "?S = {PrivKey, SymKey, PubConst, Agent, Nonce, Bot}" by (auto intro: ex_atom.exhaust)
  thus "finite ?S" by (metis finite.emptyI finite.insertI)
qed

type_synonym ex_term = "(ex_fun, ex_var) term"
type_synonym ex_terms = "(ex_fun, ex_var) terms"

primrec arity::"ex_fun ⇒ nat" where
  "arity changeCipher = 0"
| "arity clientFinished = 4"
| "arity clientHello = 5"
| "arity clientKeyExchange = 1"
| "arity concat = 5"
| "arity crypt = 2"
| "arity finished = 1"
| "arity hash = 1"
| "arity master = 3"
| "arity pmsForm = 1"
| "arity prfun = 1"
| "arity (privkey _) = 0"
| "arity pub = 1"
| "arity (pubconst _ _) = 0"
| "arity serverCert = 1"
```

```
| "arity serverHello = 5"
| "arity serverHelloDone = 0"
| "arity sign = 2"
| "arity x509 = 2"

fun public::"ex_fun ⇒ bool" where
  "public (privkey _) = False"
| "public _ = True"

fun Ana_crypt::"ex_term list ⇒ (ex_term list × ex_term list)" where
  "Ana_crypt [Fun pub [k],m] = ([k], [m])"
| "Ana_crypt _ = ([], [])"

fun Ana_sign::"ex_term list ⇒ (ex_term list × ex_term list)" where
  "Ana_sign [k,m] = ([], [m])"
| "Ana_sign _ = ([], [])"

fun Ana::"ex_term ⇒ (ex_term list × ex_term list)" where
  "Ana (Fun crypt T) = Ana_crypt T"
| "Ana (Fun finished T) = ([], T)"
| "Ana (Fun master T) = ([], T)"
| "Ana (Fun pmsForm T) = ([], T)"
| "Ana (Fun serverCert T) = ([], T)"
| "Ana (Fun serverHello T) = ([], T)"
| "Ana (Fun sign T) = Ana_sign T"
| "Ana (Fun x509 T) = ([], T)"
| "Ana _ = ([], [])"
```

7.1.2 TLS example: Locale interpretation

```
lemma assm1:
  "Ana t = (K,M) ⟹ fv_set (set K) ⊆ fv t"
  "Ana t = (K,M) ⟹ (⋀g S'. Fun g S' ⊑ t ⟹ length S' = arity g)
                ⟹ k ∈ set K ⟹ Fun f T' ⊑ k ⟹ length T' = arity f"
  "Ana t = (K,M) ⟹ K ≠ [] ∨ M ≠ [] ⟹ Ana (t · δ) = (K ·_list δ, M ·_list δ)"
by (rule Ana.cases[of "t"], auto elim!: Ana_crypt.elims Ana_sign.elims)+

lemma assm2: "Ana (Fun f T) = (K, M) ⟹ set M ⊆ set T"
by (rule Ana.cases[of "Fun f T"]) (auto elim!: Ana_crypt.elims Ana_sign.elims)

lemma assm6: "0 < arity f ⟹ public f" by (cases f) simp_all

global_interpretation im: intruder_model arity public Ana
  defines wf_trm = "im.wf_trm"
    and wf_trms = "im.wf_trms"
by unfold_locales (metis assm1(1), metis assm1(2), rule Ana.simps, metis assm2, metis assm1(3))
```

7.1.3 TLS Example: Typing function

```
definition Γ_v::"ex_var ⇒ ex_type" where
  "Γ_v v = (if (∀t ∈ subterms (fst v). case t of
              (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
            | _ ⇒ True)
         then fst v else TAtom Bot)"

fun Γ::"ex_term ⇒ ex_type" where
  "Γ (Var v) = Γ_v v"
| "Γ (Fun (privkey _) _) = TAtom PrivKey"
| "Γ (Fun changeCipher _) = TAtom PubConst"
| "Γ (Fun serverHelloDone _) = TAtom PubConst"
| "Γ (Fun (pubconst τ _) _) = TAtom τ"
| "Γ (Fun f T) = TComp f (map Γ T)"
```

7.1.4 TLS Example: Locale interpretation (typed model)

```
lemma assm7: "arity c = 0 ⟹ ∃a. ∀X. Γ (Fun c X) = TAtom a" by (cases c) simp_all

lemma assm8: "0 < arity f ⟹ Γ (Fun f X) = TComp f (map Γ X)" by (cases f) simp_all

lemma assm9: "infinite {c. Γ (Fun c []) = TAtom a ∧ public c}"
proof -
  let ?T = "(range (pubconst a))::ex_fun set"
  have *:
      "⋀x y::nat. x ∈ UNIV ⟹ y ∈ UNIV ⟹ (pubconst a x = pubconst a y) = (x = y)"
      "⋀x::nat. x ∈ UNIV ⟹ pubconst a x ∈ ?T"
      "⋀y::ex_fun. y ∈ ?T ⟹ ∃x ∈ UNIV. y = pubconst a x"
    by auto
  have "?T ⊆ {c. Γ (Fun c []) = TAtom a ∧ public c}" by auto
  moreover have "∃f::nat ⇒ ex_fun. bij_betw f UNIV ?T"
    using bij_betwI'[OF *] by blast
  hence "infinite ?T" by (metis nat_not_finite bij_betw_finite)
  ultimately show ?thesis using infinite_super by blast
qed

lemma assm10: "TComp f T ⊑ Γ t ⟹ arity f > 0"
proof (induction rule: Γ.induct)
  case (1 x)
  hence *: "TComp f T ⊑ Γ_v x" by simp
  hence "Γ_v x ≠ TAtom Bot" unfolding Γ_v_def by force
  hence "∀t ∈ subterms (fst x). case t of
              (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
            | _ ⇒ True"
    unfolding Γ_v_def by argo
  thus ?case using * unfolding Γ_v_def by fastforce
qed auto

lemma assm11: "im.wf_trm (Γ (Var x))"
proof -
  have "im.wf_trm (Γ_v x)" unfolding Γ_v_def im.wf_trm_def by auto
  thus ?thesis by simp
qed

lemma assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))"
  apply (cases "∀t ∈ subterms τ. case t of
              (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
            | _ ⇒ True")
  by (auto simp add: Γ_v_def)

lemma Ana_const: "arity c = 0 ⟹ Ana (Fun c T) = ([],[])"
by (cases c) simp_all

lemma Ana_keys_subterm: "Ana t = (K,T) ⟹ k ∈ set K ⟹ k ⊏ t"
proof (induct t rule: Ana.induct)
  case (1 U)
  then obtain m where "U = [Fun pub [k], m]" "K = [k]" "T = [m]"
    by (auto elim!: Ana_crypt.elims Ana_sign.elims)
  thus ?case using Fun_subterm_inside_params[of k crypt U] by auto
qed (auto elim!: Ana_crypt.elims Ana_sign.elims)

global_interpretation tm: typed_model' arity public Ana Γ
by (unfold_locales, unfold wf_trm_def[symmetric],
    metis assm7, metis assm8, metis assm9, metis assm10, metis assm11, metis assm6,
    metis assm12, metis Ana_const, metis Ana_keys_subterm)
```

7.1.5 TLS example: Proving type-flaw resistance

abbreviation Γ_v*_clientHello* **where**
"Γ_v_clientHello ≡
TComp clientHello [TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce]"

abbreviation Γ_v*_serverHello* **where**
"Γ_v_serverHello ≡
TComp serverHello [TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce]"

abbreviation Γ_v*_pub* **where**
"Γ_v_pub ≡ TComp pub [TAtom PrivKey]"

abbreviation Γ_v*_x509* **where**
"Γ_v_x509 ≡ TComp x509 [TAtom Agent, Γ_v_pub]"

abbreviation Γ_v*_sign* **where**
"Γ_v_sign ≡ TComp sign [TAtom PrivKey, Γ_v_x509]"

abbreviation Γ_v*_serverCert* **where**
"Γ_v_serverCert ≡ TComp serverCert [Γ_v_sign]"

abbreviation Γ_v*_pmsForm* **where**
"Γ_v_pmsForm ≡ TComp pmsForm [TAtom SymKey]"

abbreviation Γ_v*_crypt* **where**
"Γ_v_crypt ≡ TComp crypt [Γ_v_pub, Γ_v_pmsForm]"

abbreviation Γ_v*_clientKeyExchange* **where**
"Γ_v_clientKeyExchange ≡
TComp clientKeyExchange [Γ_v_crypt]"

abbreviation Γ_v*_HSMsgs* **where**
"Γ_v_HSMsgs ≡ TComp concat [
Γ_v_clientHello,
Γ_v_serverHello,
Γ_v_serverCert,
TAtom PubConst,
Γ_v_clientKeyExchange]"

abbreviation "T_1 n ≡ Var (TAtom Nonce,n)"
abbreviation "T_2 n ≡ Var (TAtom Nonce,n)"
abbreviation "R_A n ≡ Var (TAtom Nonce,n)"
abbreviation "R_B n ≡ Var (TAtom Nonce,n)"
abbreviation "S n ≡ Var (TAtom Nonce,n)"
abbreviation "Cipher n ≡ Var (TAtom Nonce,n)"
abbreviation "Comp n ≡ Var (TAtom Nonce,n)"
abbreviation "B n ≡ Var (TAtom Agent,n)"
abbreviation "Pr_{ca} n ≡ Var (TAtom PrivKey,n)"
abbreviation "PMS n ≡ Var (TAtom SymKey,n)"
abbreviation "P_B n ≡ Var (TComp pub [TAtom PrivKey],n)"
abbreviation "HSMsgs n ≡ Var (Γ_v_HSMsgs,n)"

Defining the over-approximation set

abbreviation $clientHello_{trm}$ **where**
"$clientHello_{trm}$ ≡ Fun clientHello [T_1 0, R_A 1, S 2, Cipher 3, Comp 4]"

abbreviation $serverHello_{trm}$ **where**
"$serverHello_{trm}$ ≡ Fun serverHello [T_2 0, R_B 1, S 2, Cipher 3, Comp 4]"

abbreviation $serverCert_{trm}$ **where**

```
  "serverCert_trm ≡ Fun serverCert [Fun sign [Pr_ca 0, Fun x509 [B 1, P_B 2]]]"
```

abbreviation *serverHelloDone$_{trm}$* **where**
```
  "serverHelloDone_trm ≡ Fun serverHelloDone []"
```

abbreviation *clientKeyExchange$_{trm}$* **where**
```
  "clientKeyExchange_trm ≡ Fun clientKeyExchange [Fun crypt [P_B 0, Fun pmsForm [PMS 1]]]"
```

abbreviation *changeCipher$_{trm}$* **where**
```
  "changeCipher_trm ≡ Fun changeCipher []"
```

abbreviation *finished$_{trm}$* **where**
```
  "finished_trm ≡ Fun finished [Fun prfun [
      Fun clientFinished [
          Fun prfun [Fun master [PMS 0, R_A 1, R_B 2]],
          R_A 3, R_B 4, Fun hash [HSMsgs 5]
      ]
  ]]"
```

definition *M_{TLS}::"ex_term list"* **where**
```
  "M_TLS ≡ [
    clientHello_trm,
    serverHello_trm,
    serverCert_trm,
    serverHelloDone_trm,
    clientKeyExchange_trm,
    changeCipher_trm,
    finished_trm
]"
```

7.1.6 Theorem: The TLS handshake protocol is type-flaw resistant

theorem *"tm.tfr$_{set}$ (set M_{TLS})"*
by *(rule tm.tfr$_{set}$_if_comp_tfr$_{set}$') eval*

end

7.2 The Keyserver Example (Example_Keyserver)

theory *Example_Keyserver*
imports *"../Stateful_Compositionality"*
begin

declare *[[code_timing]]*

7.2.1 Setup

Datatypes and functions setup

datatype *ex_lbl = Label1 ("1") | Label2 ("2")*

datatype *ex_atom =*
```
  Agent | Value | Attack | PrivFunSec
| Bot
```

datatype *ex_fun =*
```
  ring | valid | revoked | events | beginauth nat | endauth nat | pubkeys | seen
| invkey | tuple | tuple' | attack nat
| sign | crypt | update | pw
| encodingsecret | pubkey nat
| pubconst ex_atom nat
```

type_synonym *ex_type = "(ex_fun, ex_atom) term_type"*

```
type_synonym ex_var = "ex_type × nat"

lemma ex_atom_UNIV:
  "(UNIV::ex_atom set) = {Agent, Value, Attack, PrivFunSec, Bot}"
by (auto intro: ex_atom.exhaust)

instance ex_atom::finite
by intro_classes (metis ex_atom_UNIV finite.emptyI finite.insertI)

lemma ex_lbl_UNIV:
  "(UNIV::ex_lbl set) = {Label1, Label2}"
by (auto intro: ex_lbl.exhaust)

type_synonym ex_term = "(ex_fun, ex_var) term"
type_synonym ex_terms = "(ex_fun, ex_var) terms"

primrec arity::"ex_fun ⇒ nat" where
  "arity ring = 2"
| "arity valid = 3"
| "arity revoked = 3"
| "arity events = 1"
| "arity (beginauth _) = 3"
| "arity (endauth _) = 3"
| "arity pubkeys = 2"
| "arity seen = 2"
| "arity invkey = 2"
| "arity tuple = 2"
| "arity tuple' = 2"
| "arity (attack _) = 0"
| "arity sign = 2"
| "arity crypt = 2"
| "arity update = 4"
| "arity pw = 2"
| "arity (pubkey _) = 0"
| "arity encodingsecret = 0"
| "arity (pubconst _ _) = 0"

fun public::"ex_fun ⇒ bool" where
  "public (pubkey _) = False"
| "public encodingsecret = False"
| "public _ = True"

fun Ana_crypt::"ex_term list ⇒ (ex_term list × ex_term list)" where
  "Ana_crypt [k,m] = ([Fun invkey [Fun encodingsecret [], k]], [m])"
| "Ana_crypt _ = ([], [])"

fun Ana_sign::"ex_term list ⇒ (ex_term list × ex_term list)" where
  "Ana_sign [k,m] = ([], [m])"
| "Ana_sign _ = ([], [])"

fun Ana::"ex_term ⇒ (ex_term list × ex_term list)" where
  "Ana (Fun tuple T) = ([], T)"
| "Ana (Fun tuple' T) = ([], T)"
| "Ana (Fun sign T) = Ana_sign T"
| "Ana (Fun crypt T) = Ana_crypt T"
| "Ana _ = ([], [])"
```

Keyserver example: Locale interpretation

```
lemma assm1:
  "Ana t = (K,M) ⟹ fv_set (set K) ⊆ fv t"
  "Ana t = (K,M) ⟹ (⋀g S'. Fun g S' ⊑ t ⟹ length S' = arity g)
              ⟹ k ∈ set K ⟹ Fun f T' ⊑ k ⟹ length T' = arity f"
```

```
  "Ana t = (K,M) ⟹ K ≠ [] ∨ M ≠ [] ⟹ Ana (t · δ) = (K ·list δ, M ·list δ)"
by (rule Ana.cases[of "t"], auto elim!: Ana_crypt.elims Ana_sign.elims)+

lemma assm2: "Ana (Fun f T) = (K, M) ⟹ set M ⊆ set T"
by (rule Ana.cases[of "Fun f T"]) (auto elim!: Ana_crypt.elims Ana_sign.elims)

lemma assm6: "0 < arity f ⟹ public f" by (cases f) simp_all

global_interpretation im: intruder_model arity public Ana
  defines wf_trm = "im.wf_trm"
by unfold_locales (metis assm1(1), metis assm1(2),rule Ana.simps, metis assm2, metis assm1(3))

type_synonym ex_strand_step = "(ex_fun,ex_var) strand_step"
type_synonym ex_strand = "(ex_fun,ex_var) strand"
```

Typing function

```
definition Γv::"ex_var ⇒ ex_type" where
  "Γv v = (if (∀t ∈ subterms (fst v). case t of
              (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
            | _ ⇒ True)
          then fst v else TAtom Bot)"

fun Γ::"ex_term ⇒ ex_type" where
  "Γ (Var v) = Γv v"
| "Γ (Fun (attack _) _) = TAtom Attack"
| "Γ (Fun (pubkey _) _) = TAtom Value"
| "Γ (Fun encodingsecret _) = TAtom PrivFunSec"
| "Γ (Fun (pubconst τ _) _) = TAtom τ"
| "Γ (Fun f T) = TComp f (map Γ T)"
```

Locale interpretation: typed model

```
lemma assm7: "arity c = 0 ⟹ ∃a. ∀X. Γ (Fun c X) = TAtom a" by (cases c) simp_all

lemma assm8: "0 < arity f ⟹ Γ (Fun f X) = TComp f (map Γ X)" by (cases f) simp_all

lemma assm9: "infinite {c. Γ (Fun c []) = TAtom a ∧ public c}"
proof -
  let ?T = "(range (pubconst a))::ex_fun set"
  have *:
      "⋀x y::nat. x ∈ UNIV ⟹ y ∈ UNIV ⟹ (pubconst a x = pubconst a y) = (x = y)"
      "⋀x::nat. x ∈ UNIV ⟹ pubconst a x ∈ ?T"
      "⋀y::ex_fun. y ∈ ?T ⟹ ∃x ∈ UNIV. y = pubconst a x"
    by auto
  have "?T ⊆ {c. Γ (Fun c []) = TAtom a ∧ public c}" by auto
  moreover have "∃f::nat ⇒ ex_fun. bij_betw f UNIV ?T"
    using bij_betwI'[OF *] by blast
  hence "infinite ?T" by (metis nat_not_finite bij_betw_finite)
  ultimately show ?thesis using infinite_super by blast
qed

lemma assm10: "TComp f T ⊑ Γ t ⟹ arity f > 0"
proof (induction rule: Γ.induct)
  case (1 x)
  hence *: "TComp f T ⊑ Γv x" by simp
  hence "Γv x ≠ TAtom Bot" unfolding Γv_def by force
  hence "∀t ∈ subterms (fst x). case t of
           (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
         | _ ⇒ True"
    unfolding Γv_def by argo
  thus ?case using * unfolding Γv_def by fastforce
qed auto
```

```
lemma assm11: "im.wf_trm (Γ (Var x))"
proof -
  have "im.wf_trm (Γ_v x)" unfolding Γ_v_def im.wf_trm_def by auto
  thus ?thesis by simp
qed

lemma assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))"
apply (cases "∀t ∈ subterms τ. case t of
                (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
              | _ ⇒ True")
by (auto simp add: Γ_v_def)

lemma Ana_const: "arity c = 0 ⟹ Ana (Fun c T) = ([], [])"
by (cases c) simp_all

lemma Ana_subst': "Ana (Fun f T) = (K,M) ⟹ Ana (Fun f T · δ) = (K ·_list δ,M ·_list δ)"
by (cases f) (auto elim!: Ana_crypt.elims Ana_sign.elims)

global_interpretation tm: typed_model' arity public Ana Γ
by (unfold_locales, unfold wf_trm_def[symmetric])
   (metis assm7, metis assm8, metis assm9, metis assm10, metis assm11, metis assm6,
    metis assm12, metis Ana_const, metis Ana_subst')
```

Locale interpretation: labeled stateful typed model

```
global_interpretation stm: labeled_stateful_typed_model' arity public Ana Γ tuple 1 2
by standard (rule arity.simps, metis Ana_subst', metis assm12, metis Ana_const, simp)

type_synonym ex_stateful_strand_step = "(ex_fun,ex_var) stateful_strand_step"
type_synonym ex_stateful_strand = "(ex_fun,ex_var) stateful_strand"

type_synonym ex_labeled_stateful_strand_step =
  "(ex_fun,ex_var,ex_lbl) labeled_stateful_strand_step"

type_synonym ex_labeled_stateful_strand =
  "(ex_fun,ex_var,ex_lbl) labeled_stateful_strand"
```

7.2.2 Theorem: Type-flaw resistance of the keyserver example from the CSF18 paper

```
abbreviation "PK n ≡ Var (TAtom Value,n)"
abbreviation "A n ≡ Var (TAtom Agent,n)"
abbreviation "X n ≡ (TAtom Agent,n)"

abbreviation "ringset t ≡ Fun ring [Fun encodingsecret [], t]"
abbreviation "validset t t' ≡ Fun valid [Fun encodingsecret [], t, t']"
abbreviation "revokedset t t' ≡ Fun revoked [Fun encodingsecret [], t, t']"
abbreviation "eventsset ≡ Fun events [Fun encodingsecret []]"


abbreviation S_ks::"(ex_fun,ex_var) stateful_strand_step list" where
  "S_ks ≡ [
    insert⟨Fun (attack 0) [], eventsset⟩,
    delete⟨PK 0, validset (A 0) (A 0)⟩,
    ∀(TAtom Agent,0)⟨PK 0 not in revokedset (A 0) (A 0)⟩,
    ∀(TAtom Agent,0)⟨PK 0 not in validset (A 0) (A 0)⟩,
    insert⟨PK 0, validset (A 0) (A 0)⟩,
    insert⟨PK 0, ringset (A 0)⟩,
    insert⟨PK 0, revokedset (A 0) (A 0)⟩,
    select⟨PK 0, validset (A 0) (A 0)⟩,
    select⟨PK 0, ringset (A 0)⟩,
    receive⟨Fun invkey [Fun encodingsecret [], PK 0]⟩,
    receive⟨Fun sign [Fun invkey [Fun encodingsecret [], PK 0], Fun tuple' [A 0, PK 0]]⟩,
```

```
    send⟨Fun invkey [Fun encodingsecret [], PK 0]⟩,
    send⟨Fun sign [Fun invkey [Fun encodingsecret [], PK 0], Fun tuple' [A 0, PK 0]]⟩
]"

theorem "stm.tfr_sst S_ks"
proof -
  let ?M = "concat (map subterms_list (trms_list_sst S_ks@map (pair' tuple) (setops_list_sst S_ks)))"
  have "comp_tfr_sst arity Ana Γ tuple ?M S_ks" by eval
  thus ?thesis by (rule stm.tfr_sst_if_comp_tfr_sst)
qed
```

7.2.3 Theorem: Type-flaw resistance of the keyserver examples from the ESORICS18 paper

```
abbreviation "signmsg t t' ≡ Fun sign [t, t']"
abbreviation "cryptmsg t t' ≡ Fun crypt [t, t']"
abbreviation "invkeymsg t ≡ Fun invkey [Fun encodingsecret [], t]"
abbreviation "updatemsg a b c d ≡ Fun update [a,b,c,d]"
abbreviation "pwmsg t t' ≡ Fun pw [t, t']"

abbreviation "beginauthset n t t' ≡ Fun (beginauth n) [Fun encodingsecret [], t, t']"
abbreviation "endauthset n t t' ≡ Fun (endauth n) [Fun encodingsecret [], t, t']"
abbreviation "pubkeysset t ≡ Fun pubkeys [Fun encodingsecret [], t]"
abbreviation "seenset t ≡ Fun seen [Fun encodingsecret [], t]"

declare [[coercion "Var::ex_var ⇒ ex_term"]]
declare [[coercion_enabled]]


definition S'_ks::"ex_labeled_stateful_strand_step list" where
  "S'_ks ≡ [
constraint steps from the first protocol (duplicate steps are ignored)

    rule R^1_1
    ⟨1, send⟨invkeymsg (PK 0)⟩⟩,
    ⟨⋆, ⟨PK 0 in validset (A 0) (A 1)⟩⟩,
    ⟨1, receive⟨Fun (attack 0) []⟩⟩,

    rule R^1_2
    ⟨1, send⟨signmsg (invkeymsg (PK 0)) (Fun tuple' [A 0, PK 0])⟩⟩,
    ⟨⋆, ⟨PK 0 in validset (A 0) (A 1)⟩⟩,
    ⟨⋆, ∀X 0, X 1⟨PK 0 not in validset (Var (X 0)) (Var (X 1))⟩⟩,
    ⟨1, ∀X 0, X 1⟨PK 0 not in revokedset (Var (X 0)) (Var (X 1))⟩⟩,
    ⟨⋆, ⟨PK 0 not in beginauthset 0 (A 0) (A 1)⟩⟩,

    rule R^1_3
    ⟨⋆, ⟨PK 0 in beginauthset 0 (A 0) (A 1)⟩⟩,
    ⟨⋆, ⟨PK 0 in endauthset 0 (A 0) (A 1)⟩⟩,

    rule R^1_4
    ⟨⋆, receive⟨PK 0⟩⟩,
    ⟨⋆, receive⟨invkeymsg (PK 0)⟩⟩,

    rule R^1_5
    ⟨1, insert⟨PK 0, ringset (A 0)⟩⟩,
    ⟨⋆, insert⟨PK 0, validset (A 0) (A 1)⟩⟩,
    ⟨⋆, insert⟨PK 0, beginauthset 0 (A 0) (A 1)⟩⟩,
    ⟨⋆, insert⟨PK 0, endauthset 0 (A 0) (A 1)⟩⟩,

    rule R^1_6
    ⟨1, select⟨PK 0, ringset (A 0)⟩⟩,
    ⟨1, delete⟨PK 0, ringset (A 0)⟩⟩,
```

```
    rule R7_1
    ⟨⋆, ⟨PK 0 not in endauthset 0 (A 0) (A 1)⟩⟩,
    ⟨⋆, delete⟨PK 0, validset (A 0) (A 1)⟩⟩,
    ⟨1, insert⟨PK 0, revokedset (A 0) (A 1)⟩⟩,

    rule R8_1
    nothing new

    rule R9_1
    ⟨1, send⟨PK 0⟩⟩,

    rule R10_1
    ⟨1, send⟨Fun (attack 0) []⟩⟩,

constraint steps from the second protocol (duplicate steps are ignored)
    rule R1_2
    ⟨2, send⟨invkeymsg (PK 0)⟩⟩,
    ⟨⋆, ⟨PK 0 in validset (A 0) (A 1)⟩⟩,
    ⟨2, receive⟨Fun (attack 1) []⟩⟩,

    rule R2_2
    ⟨2, send⟨cryptmsg (PK 0) (updatemsg (A 0) (A 1) (PK 1) (pwmsg (A 0) (A 1)))⟩⟩,
    ⟨2, select⟨PK 0, pubkeysset (A 0)⟩⟩,
    ⟨2, ∀X 0⟨PK 0 not in pubkeysset (Var (X 0))⟩⟩,
    ⟨2, ∀X 0⟨PK 0 not in seenset (Var (X 0))⟩⟩,

    rule R3_2
    ⟨⋆, ⟨PK 0 in beginauthset 1 (A 0) (A 1)⟩⟩,
    ⟨⋆, ⟨PK 0 in endauthset 1 (A 0) (A 1)⟩⟩,

    rule R4_2
    ⟨⋆, receive⟨PK 0⟩⟩,
    ⟨⋆, receive⟨invkeymsg (PK 0)⟩⟩,

    rule R5_2
    ⟨2, select⟨PK 0, pubkeysset (A 0)⟩⟩,
    ⟨⋆, insert⟨PK 0, beginauthset 1 (A 0) (A 1)⟩⟩,
    ⟨2, receive⟨cryptmsg (PK 0) (updatemsg (A 0) (A 1) (PK 1) (pwmsg (A 0) (A 1)))⟩⟩,

    rule R6_2
    ⟨⋆, ⟨PK 0 not in endauthset 1 (A 0) (A 1)⟩⟩,
    ⟨⋆, insert⟨PK 0, validset (A 0) (A 1)⟩⟩,
    ⟨⋆, insert⟨PK 0, endauthset 1 (A 0) (A 1)⟩⟩,
    ⟨2, insert⟨PK 0, seenset (A 0)⟩⟩,

    rule R7_2
    ⟨2, receive⟨pwmsg (A 0) (A 1)⟩⟩,

    rule R8_2
    nothing new

    rule R9_2
    ⟨2, insert⟨PK 0, pubkeysset (A 0)⟩⟩,

    rule R10_2
    ⟨2, send⟨Fun (attack 1) []⟩⟩
]"

theorem "stm.tfr_sst (unlabel S'_ks)"
proof -
  let ?S = "unlabel S'_ks"
  let ?M = "concat (map subterms_list (trms_list_sst ?S@map (pair' tuple) (setops_list_sst ?S)))"
  have "comp_tfr_sst arity Ana Γ tuple ?M ?S" by eval
```

```
  thus ?thesis by (rule stm.tfr_sst_if_comp_tfr_sst)
qed
```

7.2.4 Theorem: The steps of the keyserver protocols from the ESORICS18 paper satisfy the conditions for parallel composition

```
theorem
  fixes S f
  defines "S ≡ [PK 0, invkeymsg (PK 0), Fun encodingsecret []]@concat (
              map (λs. [s, Fun tuple [PK 0, s]])
                  [validset (A 0) (A 1), beginauthset 0 (A 0) (A 1), endauthset 0 (A 0) (A 1),
                   beginauthset 1 (A 0) (A 1), endauthset 1 (A 0) (A 1)])@
              [A 0]"
    and "f ≡ λM. {t · δ | t δ. t ∈ M ∧ tm.wt_subst δ ∧ im.wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Sec ≡ (f (set S)) - {m. im.intruder_synth {} m}"
  shows "stm.par_comp_lsst S'_ks Sec"
proof -
  let ?N = "λP. concat (map subterms_list (trms_list_sst P@map (pair' tuple) (setops_list_sst P)))"
  let ?M = "λl. ?N (proj_unl l S'_ks)"
  have "comp_par_comp_lsst public arity Ana Γ tuple S'_ks ?M S"
    unfolding S_def by eval
  thus ?thesis
    using stm.par_comp_lsst_if_comp_par_comp_lsst[of S'_ks ?M S]
    unfolding Sec_def f_def wf_trm_def[symmetric] by blast
qed

end
```

Bibliography

[1] A. V. Hess. *Typing and Compositionality for Stateful Security Protocols*. PhD thesis, 2019. URL `https://orbit.dtu.dk/en/publications/typing-and-compositionality-for-stateful-security-protocols`.

[2] A. V. Hess and S. Mödersheim. Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 451–463. IEEE Computer Society, 2017. doi: 10.1109/CSF.2017.27.

[3] A. V. Hess and S. Mödersheim. A Typing Result for Stateful Protocols. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 374–388. IEEE Computer Society, 2018. doi: 10.1109/CSF.2018.00034.

[4] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful Protocol Composition. In J. López, J. Zhou, and M. Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 427–446. Springer, 2018. doi: 10.1007/978-3-319-99073-6_21.