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.

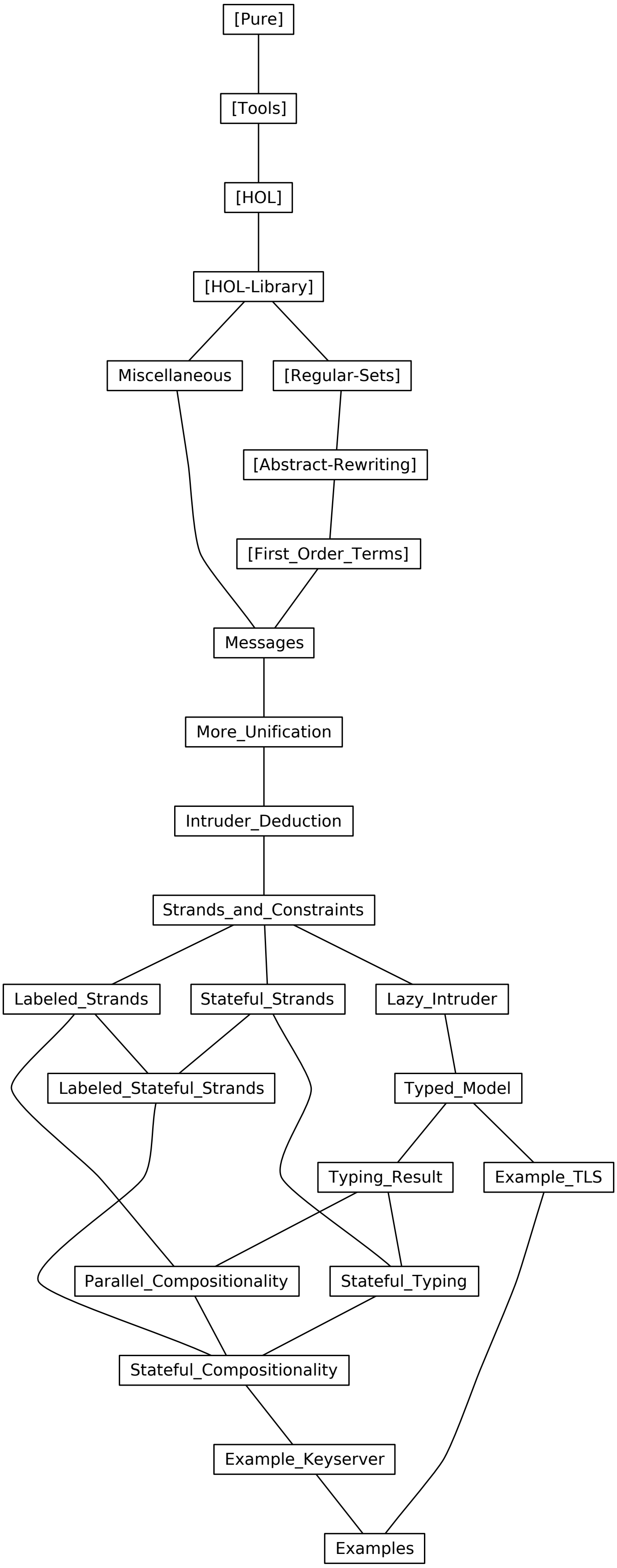

Figure 1.1: The Dependency Graph of the Isabelle Theories.

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⟩

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⟩

lemma filter_nth: "i < length (filter P xs) ⟹ P (filter P xs ! i)"
⟨proof⟩

lemma list_all_filter_eq: "list_all P xs ⟹ filter P xs = xs"
⟨proof⟩

lemma list_all_filter_nil:
  assumes "list_all P xs"
    and "⋀x. P x ⟹ ¬Q x"
  shows "filter Q xs = []"
⟨proof⟩

lemma list_all_concat: "list_all (list_all f) P ⟷ list_all f (concat P)"
⟨proof⟩

lemma map_upt_index_eq:
  assumes "j < length xs"
  shows "(map (λi. xs ! is i) [0..<length xs]) ! j = xs ! is j"
⟨proof⟩

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)"
⟨proof⟩

lemma map_append_inv: "map f xs = ys@zs ⟹ ∃vs ws. xs = vs@ws ∧ map f vs = ys ∧ map f ws = zs"
⟨proof⟩
```

2.1.2 List: subsequences

```
lemma subseqs_set_subset:
  assumes "ys ∈ set (subseqs xs)"
  shows "set ys ⊆ set xs"
⟨proof⟩
```

lemma *subset_sublist_exists:*
  "ys ⊆ set xs ⟹ ∃zs. set zs = ys ∧ zs ∈ set (subseqs xs)"
⟨*proof*⟩

lemma *map_subseqs: "map (map f) (subseqs xs) = subseqs (map f xs)"*
⟨*proof*⟩

lemma *subseqs_Cons:*
  assumes *"ys ∈ set (subseqs xs)"*
  shows *"ys ∈ set (subseqs (x#xs))"*
⟨*proof*⟩

lemma *subseqs_subset:*
  assumes *"ys ∈ set (subseqs xs)"*
  shows *"set ys ⊆ set xs"*
⟨*proof*⟩

2.1.3 List: prefixes, suffixes

lemma *suffix_Cons': "suffix [x] (y#ys) ⟹ suffix [x] ys ∨ (y = x ∧ ys = [])"*
⟨*proof*⟩

lemma *prefix_Cons': "prefix (x#xs) (x#ys) ⟹ prefix xs ys"*
⟨*proof*⟩

lemma *prefix_map: "prefix xs (map f ys) ⟹ ∃zs. prefix zs ys ∧ map f zs = xs"*
⟨*proof*⟩

lemma *length_prefix_ex:*
  assumes *"n ≤ length xs"*
  shows *"∃ys zs. xs = ys@zs ∧ length ys = n"*
  ⟨*proof*⟩

lemma *length_prefix_ex':*
  assumes *"n < length xs"*
  shows *"∃ys zs. xs = ys@xs ! n#zs ∧ length ys = n"*
⟨*proof*⟩

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"*
⟨*proof*⟩

2.1.4 List: products

lemma *product_lists_Cons:*
  "x#xs ∈ set (product_lists (y#ys)) ⟷ (xs ∈ set (product_lists ys) ∧ x ∈ set y)"
⟨*proof*⟩

lemma *product_lists_in_set_nth:*
  assumes *"xs ∈ set (product_lists ys)"*
  shows *"∀i<length ys. xs ! i ∈ set (ys ! i)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

2.1.5 Other Lemmata

lemma *inv_set_fset: "finite M ⟹ set (inv set M) = M"*

⟨proof⟩

```
lemma lfp_eqI':
  assumes "mono f"
    and "f C = C"
    and "∀X ∈ Pow C. f X = X ⟶ X = C"
  shows "lfp f = C"
⟨proof⟩

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⟩

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⟩

lemma preordered_finite_set_has_maxima:
  assumes "finite A" "A ≠ {}"
  shows "∃a::'a::{preorder} ∈ A. ∀b ∈ A. ¬(a < b)"
⟨proof⟩

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⟩

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⟩

lemma trancl_eqI:
  assumes "∀(a,b) ∈ A. ∀(c,d) ∈ A. b = c ⟶ (a,d) ∈ A"
  shows "A = A⁺"
⟨proof⟩

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⟩

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⟩

lemma remdups_ex2:
  "length (remdups xs) > 1 ⟹ ∃a ∈ set xs. ∃b ∈ set xs. a ≠ b"
```

⟨*proof*⟩

lemma *trancl_minus_refl_idem:*
  defines "cl ≡ λts. {(a,b) ∈ ts⁺. a ≠ b}"
  shows "cl (cl ts) = cl ts"
⟨*proof*⟩

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*⟩

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*⟩

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"
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *fv_list$_{pairs}$_is_fv$_{pairs}$: "fv$_{pairs}$ F = set (fv_list$_{pairs}$ F)"*
⟨*proof*⟩

lemma *subterms_list_is_subterms: "subterms t = set (subterms_list t)"*
⟨*proof*⟩

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*⟩

lemma *subterm_of_ex_set_iff_subtermeqset[code_unfold]: "t ⊑$_{set}$ M = (∃t' ∈ M. subterm_of t t')"*
⟨*proof*⟩

2.2.4 The subterm relation is a partial order on terms

interpretation *"term": order "(⊑)" "(⊏)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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) = {}"*
⟨*proof*⟩

lemmas *subtermeqI'[intro] = term.eq_refl*

lemma *subtermeqI''[intro]: "t ∈ set T ⟹ t ⊑ Fun f T"*
⟨*proof*⟩

lemma *finite_fv_set[intro]: "finite M ⟹ finite (fv$_{set}$ M)"*

⟨proof⟩

lemma *finite_fun_symbols[simp]: "finite (funs_term t)"*
⟨proof⟩

lemma *fv_set_mono: "M ⊆ N ⟹ fv_{set} M ⊆ fv_{set} N"*
⟨proof⟩

lemma *$subterms_{set}$_mono: "M ⊆ N ⟹ $subterms_{set}$ M ⊆ $subterms_{set}$ N"*
⟨proof⟩

lemma *ground_empty[simp]: "ground {}"*
⟨proof⟩

lemma *ground_subset: "M ⊆ N ⟹ ground N ⟹ ground M"*
⟨proof⟩

lemma *fv_map_fv_set: "⋃(set (map fv L)) = fv_{set} (set L)"*
⟨proof⟩

lemma *fv_{set}_union: "fv_{set} (M ∪ N) = fv_{set} M ∪ fv_{set} N"*
⟨proof⟩

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)"*
⟨proof⟩

lemma *inv_set_fv: "finite M ⟹ ⋃(set (map fv (inv set M))) = fv_{set} M"*
⟨proof⟩

lemma *ground_subterm: "fv t = {} ⟹ t' ⊑ t ⟹ fv t' = {}"* ⟨proof⟩

lemma *empty_fv_not_var: "fv t = {} ⟹ t ≠ Var x"* ⟨proof⟩

lemma *empty_fv_exists_fun: "fv t = {} ⟹ ∃f X. t = Fun f X"* ⟨proof⟩

lemma *vars_iff_subtermeq: "x ∈ fv t ⟷ Var x ⊑ t"* ⟨proof⟩

lemma *vars_iff_subtermeq_set: "x ∈ fv_{set} M ⟷ Var x ∈ $subterms_{set}$ M"*
⟨proof⟩

lemma *vars_if_subtermeq_set: "Var x ∈ $subterms_{set}$ M ⟹ x ∈ fv_{set} M"*
⟨proof⟩

lemma *subtermeq_set_if_vars: "x ∈ fv_{set} M ⟹ Var x ∈ $subterms_{set}$ M"*
⟨proof⟩

lemma *vars_iff_subterm_or_eq: "x ∈ fv t ⟷ Var x ⊏ t ∨ Var x = t"*
⟨proof⟩

lemma *var_is_subterm: "x ∈ fv t ⟹ Var x ∈ subterms t"*
⟨proof⟩

lemma *subterm_is_var: "Var x ∈ subterms t ⟹ x ∈ fv t"*
⟨proof⟩

lemma *no_var_subterm: "¬t ⊏ Var v"* ⟨proof⟩

lemma *fun_if_subterm: "t ⊏ u ⟹ ∃f X. u = Fun f X"* ⟨proof⟩

lemma *subtermeq_vars_subset: "M ⊑ N ⟹ fv M ⊆ fv N"* ⟨proof⟩

lemma *fv_subterms[simp]: "fv_{set} (subterms t) = fv t"*
⟨*proof*⟩

lemma *fv_subterms_set[simp]: "fv_{set} ($subterms_{set}$ M) = fv_{set} M"*
⟨*proof*⟩

lemma *fv_subset: "t ∈ M ⟹ fv t ⊆ fv_{set} M"*
⟨*proof*⟩

lemma *fv_subset_subterms: "t ∈ $subterms_{set}$ M ⟹ fv t ⊆ fv_{set} M"*
⟨*proof*⟩

lemma *subterms_finite[simp]: "finite (subterms t)"* ⟨*proof*⟩

lemma *subterms_union_finite: "finite M ⟹ finite (⋃t ∈ M. subterms t)"*
⟨*proof*⟩

lemma *subterms_subset: "t' ⊑ t ⟹ subterms t' ⊆ subterms t"*
⟨*proof*⟩

lemma *subterms_subset_set: "M ⊆ subterms t ⟹ $subterms_{set}$ M ⊆ subterms t"*
⟨*proof*⟩

lemma *subset_subterms_Union[simp]: "M ⊆ $subterms_{set}$ M"* ⟨*proof*⟩

lemma *in_subterms_Union: "t ∈ M ⟹ t ∈ $subterms_{set}$ M"* ⟨*proof*⟩

lemma *in_subterms_subset_Union: "t ∈ $subterms_{set}$ M ⟹ subterms t ⊆ $subterms_{set}$ M"*
⟨*proof*⟩

lemma *subterm_param_split:*
  assumes *"t ⊏ Fun f X"*
  shows *"∃pre x suf. t ⊑ x ∧ X = pre@x#suf"*
⟨*proof*⟩

lemma *ground_iff_no_vars: "ground (M::('a,'b) terms) ⟷ (∀v. Var v ∉ (⋃m ∈ M. subterms m))"*
⟨*proof*⟩

lemma *index_Fun_subterms_subset[simp]: "i < length T ⟹ subterms (T ! i) ⊆ subterms (Fun f T)"*
⟨*proof*⟩

lemma *index_Fun_fv_subset[simp]: "i < length T ⟹ fv (T ! i) ⊆ fv (Fun f T)"*
⟨*proof*⟩

lemma *subterms_union_ground:*
  assumes *"ground M"*
  shows *"ground ($subterms_{set}$ M)"*
⟨*proof*⟩

lemma *Var_subtermeq: "t ⊑ Var v ⟹ t = Var v"* ⟨*proof*⟩

lemma *subtermeq_imp_funs_term_subset: "s ⊑ t ⟹ funs_term s ⊆ funs_term t"*
⟨*proof*⟩

lemma *subterms_const: "subterms (Fun f []) = {Fun f []}"* ⟨*proof*⟩

lemma *subterm_subtermeq_neq: "⟦t ⊏ u; u ⊑ v⟧ ⟹ t ≠ v"*
⟨*proof*⟩

lemma *subtermeq_subterm_neq: "⟦t ⊑ u; u ⊏ v⟧ ⟹ t ≠ v"*
⟨*proof*⟩

lemma *subterm_size_lt:* "x ⊏ y ⟹ size x < size y"
⟨*proof*⟩

lemma *in_subterms_eq:* "⟦x ∈ subterms y; y ∈ subterms x⟧ ⟹ subterms x = subterms y"
⟨*proof*⟩

lemma *Fun_gt_params:* "Fun f X ∉ (⋃x ∈ set X. subterms x)"
⟨*proof*⟩

lemma *params_subterms[simp]:* "set X ⊆ subterms (Fun f X)" ⟨*proof*⟩

lemma *params_subterms_Union[simp]:* "subterms$_{set}$ (set X) ⊆ subterms (Fun f X)" ⟨*proof*⟩

lemma *Fun_subterm_inside_params:* "t ⊏ Fun f X ⟷ t ∈ (⋃x ∈ (set X). subterms x)"
⟨*proof*⟩

lemma *Fun_param_is_subterm:* "x ∈ set X ⟹ x ⊏ Fun f X"
⟨*proof*⟩

lemma *Fun_param_in_subterms:* "x ∈ set X ⟹ x ∈ subterms (Fun f X)"
⟨*proof*⟩

lemma *Fun_not_in_param:* "x ∈ set X ⟹ ¬Fun f X ⊏ x"
⟨*proof*⟩

lemma *Fun_ex_if_subterm:* "t ⊏ s ⟹ ∃f T. Fun f T ⊑ s ∧ t ∈ set T"
⟨*proof*⟩

lemma *const_subterm_obtain:*
  assumes "fv t = {}"
  obtains *c* **where** "Fun c [] ⊑ t"
⟨*proof*⟩

lemma *const_subterm_obtain':* "fv t = {} ⟹ ∃c. Fun c [] ⊑ t"
⟨*proof*⟩

lemma *subterms_singleton:*
  assumes "(∃v. t = Var v) ∨ (∃f. t = Fun f [])"
  shows "subterms t = {t}"
⟨*proof*⟩

lemma *subtermeq_Var_const:*
  assumes "s ⊑ t"
  shows "t = Var v ⟹ s = Var v" "t = Fun f [] ⟹ s = Fun f []"
⟨*proof*⟩

lemma *subterms_singleton':*
  assumes "subterms t = {t}"
  shows "(∃v. t = Var v) ∨ (∃f. t = Fun f [])"
⟨*proof*⟩

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*⟩

lemmas *subtermI'[intro] = Fun_param_is_subterm*

lemma *funs_term_Fun_subterm:* "f ∈ funs_term t ⟹ ∃T. Fun f T ∈ subterms t"
⟨*proof*⟩

lemma *funs_term_Fun_subterm':* "Fun f T ∈ subterms t ⟹ f ∈ funs_term t"
⟨*proof*⟩

lemma *zip_arg_subterm:*
  assumes *"(s,t) ∈ set (zip X Y)"*
  shows *"s ⊏ Fun f X" "t ⊏ Fun g Y"*
⟨*proof*⟩

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*⟩

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)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *finite_fv$_{pairs}$[simp]: "finite (fv$_{pairs}$ x)"* ⟨*proof*⟩

lemma *fv$_{pairs}$_Nil[simp]: "fv$_{pairs}$ [] = {}"* ⟨*proof*⟩

lemma *fv$_{pairs}$_singleton[simp]: "fv$_{pairs}$ [(t,s)] = fv t ∪ fv s"* ⟨*proof*⟩

lemma *fv$_{pairs}$_Cons: "fv$_{pairs}$ ((s,t)#F) = fv s ∪ fv t ∪ fv$_{pairs}$ F"* ⟨*proof*⟩

lemma *fv$_{pairs}$_append: "fv$_{pairs}$ (F@G) = fv$_{pairs}$ F ∪ fv$_{pairs}$ G"* ⟨*proof*⟩

lemma *fv$_{pairs}$_mono: "set M ⊆ set N ⟹ fv$_{pairs}$ M ⊆ fv$_{pairs}$ N"* ⟨*proof*⟩

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"
⟨*proof*⟩

lemma *fv$_{pairs}$_cons_subset: "fv$_{pairs}$ F ⊆ fv$_{pairs}$ (f#F)"*
⟨*proof*⟩

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*⟩

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** "$\cdot_{list}$" 51) **where**
  "T $\cdot_{list}$ ϑ ≡ map (λt. t · ϑ) T"

abbreviation subst_apply_pair (**infixl** "$\cdot_{p}$" 60) **where**
  "d $\cdot_{p}$ ϑ ≡ (case d of (t,t') ⇒ (t · ϑ, t' · ϑ))"

abbreviation subst_apply_pair_set (**infixl** "$\cdot_{pset}$" 60) **where**
  "M $\cdot_{pset}$ ϑ ≡ (λd. d $\cdot_{p}$ ϑ) ‘ M"

definition subst_apply_pairs (**infix** "$\cdot_{pairs}$" 51) **where**
  "F $\cdot_{pairs}$ ϑ ≡ map (λf. f $\cdot_{p}$ ϑ) F"

abbreviation subst_more_general_than (**infixl** "$\preceq_\circ$" 50) **where**
  "σ $\preceq_\circ$ ϑ ≡ ∃γ. ϑ = σ $\circ_s$ 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 $\circ_s$ s = s"

lemma subst_support_def: "ϑ supports τ ⟷ τ = ϑ $\circ_s$ τ"
⟨proof⟩

lemma subst_supportD: "ϑ supports δ ⟹ ϑ $\preceq_\circ$ δ"
⟨proof⟩

lemma rm_vars_empty[simp]: "rm_vars {} s = s" "rm_vars (set []) s = s"
⟨proof⟩

lemma rm_vars_singleton: "rm_vars {v} s = rm_var v s"
⟨proof⟩

lemma subst_apply_terms_empty: "M $\cdot_{set}$ Var = M"
⟨proof⟩

lemma subst_agreement: "(t · r = t · s) ⟷ (∀v ∈ fv t. Var v · r = Var v · s)"
⟨proof⟩

lemma repl_invariance[dest?]: "v ∉ fv t ⟹ t · s(v := u) = t · s"
⟨proof⟩

lemma subst_idx_map:
  assumes "∀i ∈ set I. i < length T"
  shows "(map ((!) T) I) $\cdot_{list}$ δ = map ((!) (map (λt. t · δ) T)) I"
⟨proof⟩

lemma subst_idx_map':
  assumes "∀i ∈ fv_{set} (set K). i < length T"
  shows "(K $\cdot_{list}$ (!) T) $\cdot_{list}$ δ = K $\cdot_{list}$ ((!) (map (λt. t · δ) T))" (**is** "?A = ?B")
⟨proof⟩

lemma *subst_remove_var: "v ∉ fv s ⟹ v ∉ fv (t · Var(v := s))"*
⟨*proof*⟩

lemma *subst_set_map: "x ∈ set X ⟹ x · s ∈ set (map (λx. x · s) X)"*
⟨*proof*⟩

lemma *subst_set_idx_map:*
  assumes *"∀i ∈ I. i < length T"*
  shows *"(!) T ' I $\cdot_{set}$ δ = (!) (map (λt. t · δ) T) ' I"* (**is** *"?A = ?B"*)
⟨*proof*⟩

lemma *subst_set_idx_map':*
  assumes *"∀i ∈ fv_{set} K. i < length T"*
  shows *"K $\cdot_{set}$ (!) T $\cdot_{set}$ δ = K $\cdot_{set}$ (!) (map (λt. t · δ) T)"* (**is** *"?A = ?B"*)
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *subst_mono: "t ⊑ u ⟹ t · s ⊑ u · s"*
⟨*proof*⟩

lemma *subst_mono_fv: "x ∈ fv t ⟹ s x ⊑ t · s"*
⟨*proof*⟩

lemma *subst_mono_neq:*
  assumes *"t ⊏ u"*
  shows *"t · s ⊏ u · s"*
⟨*proof*⟩

lemma *subst_no_occs[dest]: "¬Var v ⊑ t ⟹ t · Var(v := s) = t"*
⟨*proof*⟩

lemma *var_comp[simp]: "σ $\circ_s$ Var = σ" "Var $\circ_s$ σ = σ"*
⟨*proof*⟩

lemma *subst_comp_all: "M $\cdot_{set}$ (δ $\circ_s$ ϑ) = (M $\cdot_{set}$ δ) $\cdot_{set}$ ϑ"*
⟨*proof*⟩

lemma *subst_all_mono: "M ⊆ M' ⟹ M $\cdot_{set}$ s ⊆ M' $\cdot_{set}$ s"*
⟨*proof*⟩

lemma *subst_comp_set_image: "(δ $\circ_s$ ϑ) ' X = δ ' X $\cdot_{set}$ ϑ"*
⟨*proof*⟩

lemma *subst_ground_ident[dest?]: "fv t = {} ⟹ t · s = t"*
⟨*proof*⟩

lemma *subst_ground_ident_compose:*
  "fv (σ x) = {} ⟹ (σ $\circ_s$ ϑ) x = σ x"
  "fv (t · σ) = {} ⟹ t · (σ $\circ_s$ ϑ) = t · σ"
⟨*proof*⟩

lemma *subst_all_ground_ident[dest?]: "ground M ⟹ M $\cdot_{set}$ s = M"*
⟨*proof*⟩

lemma *subst_eqI[intro]: "(⋀t. t · σ = t · ϑ) ⟹ σ = ϑ"*
⟨*proof*⟩

lemma *subst_cong: "⟦σ = σ'; ϑ = ϑ'⟧ ⟹ (σ $\circ_s$ ϑ) = (σ' $\circ_s$ ϑ')"*

⟨*proof*⟩

lemma *subst_mgt_bot[simp]: "Var* $\preceq_\circ$ ϑ*"*
⟨*proof*⟩

lemma *subst_mgt_refl[simp]: "*$\vartheta \preceq_\circ \vartheta$*"*
⟨*proof*⟩

lemma *subst_mgt_trans: "*$[\![\vartheta \preceq_\circ \delta;\ \delta \preceq_\circ \sigma]\!] \Longrightarrow \vartheta \preceq_\circ \sigma$*"*
⟨*proof*⟩

lemma *subst_mgt_comp: "*$\vartheta \preceq_\circ \vartheta \circ_s \delta$*"*
⟨*proof*⟩

lemma *subst_mgt_comp': "*$\vartheta \circ_s \delta \preceq_\circ \sigma \Longrightarrow \vartheta \preceq_\circ \sigma$*"*
⟨*proof*⟩

lemma *var_self: "(*λ*w. if w = v then Var v else Var w) = Var"*
⟨*proof*⟩

lemma *var_same[simp]: "Var(v := t) = Var* $\longleftrightarrow$ *t = Var v"*
⟨*proof*⟩

lemma *subst_eq_if_eq_vars: "(*$\bigwedge$*v. (Var v)* $\cdot\ \vartheta$ *= (Var v)* $\cdot\ \sigma$*)* $\Longrightarrow \vartheta = \sigma$*"*
⟨*proof*⟩

lemma *subst_all_empty[simp]: "{}* $\cdot_{set}\ \vartheta$ *= {}"*
⟨*proof*⟩

lemma *subst_all_insert:"(insert t M)* $\cdot_{set}\ \delta$ *= insert (t* $\cdot\ \delta$*) (M* $\cdot_{set}\ \delta$*)"*
⟨*proof*⟩

lemma *subst_apply_fv_subset: "fv t* $\subseteq$ *V* $\Longrightarrow$ *fv (t* $\cdot\ \delta$*)* $\subseteq$ fv_{set} *(*δ *` V)"*
⟨*proof*⟩

lemma *subst_apply_fv_empty:*
  assumes *"fv t = {}"*
  shows *"fv (t* $\cdot\ \sigma$*) = {}"*
⟨*proof*⟩

lemma *subst_compose_fv:*
  assumes *"fv (*ϑ *x) = {}"*
  shows *"fv ((*$\vartheta \circ_s \sigma$*) x) = {}"*
⟨*proof*⟩

lemma *subst_compose_fv':*
  fixes $\vartheta\ \sigma$*::"('a,'b) subst"*
  assumes *"y* $\in$ *fv ((*$\vartheta \circ_s \sigma$*) x)"*
  shows *"*$\exists$*z. z* $\in$ *fv (*ϑ *x)"*
⟨*proof*⟩

lemma *subst_apply_fv_unfold: "fv (t* $\cdot\ \delta$*) =* fv_{set} *(*δ *` fv t)"*
⟨*proof*⟩

lemma *subst_apply_fv_unfold': "fv (t* $\cdot\ \delta$*) = (*$\bigcup$*v* $\in$ *fv t. fv (*δ *v))"*
⟨*proof*⟩

lemma *subst_apply_fv_union: "*fv_{set} *(*δ *` V)* $\cup$ *fv (t* $\cdot\ \delta$*) =* $fv_{set}$ *(*$\delta$ *` (V* $\cup$ *fv t))"*
⟨*proof*⟩

lemma *subst_elimI[intro]: "(*$\bigwedge$*t. v* $\notin$ *fv (t* $\cdot\ \sigma$*))* $\Longrightarrow$ *subst_elim* σ *v"*
⟨*proof*⟩

lemma *subst_elimI'[intro]: "(⋀w. v ∉ fv (Var w · ϑ)) ⟹ subst_elim ϑ v"*
⟨*proof*⟩

lemma *subst_elimD[dest]: "subst_elim σ v ⟹ v ∉ fv (t · σ)"*
⟨*proof*⟩

lemma *subst_elimD'[dest]: "subst_elim σ v ⟹ σ v ≠ Var v"*
⟨*proof*⟩

lemma *subst_elimD''[dest]: "subst_elim σ v ⟹ v ∉ fv (σ w)"*
⟨*proof*⟩

lemma *subst_elim_rm_vars_dest[dest]:*
"subst_elim (σ::('a,'b) subst) v ⟹ v ∉ vs ⟹ subst_elim (rm_vars vs σ) v"
⟨*proof*⟩

lemma *occs_subst_elim: "¬Var v ⊏ t ⟹ subst_elim (Var(v := t)) v ∨ (Var(v := t)) = Var"*
⟨*proof*⟩

lemma *occs_subst_elim': "¬Var v ⊑ t ⟹ subst_elim (Var(v := t)) v"*
⟨*proof*⟩

lemma *subst_elim_comp: "subst_elim ϑ v ⟹ subst_elim (δ $\circ_s$ ϑ) v"*
⟨*proof*⟩

lemma *var_subst_idem: "subst_idem Var"*
⟨*proof*⟩

lemma *var_upd_subst_idem:*
assumes *"¬Var v ⊑ t"* **shows** *"subst_idem (Var(v := t))"*
⟨*proof*⟩

2.3.2 Lemmata: Domain and Range of Substitutions

lemma *range_vars_alt_def: "range_vars s ≡ fv_{set} (subst_range s)"*
⟨*proof*⟩

lemma *subst_dom_var_finite[simp]: "finite (subst_domain Var)"* ⟨*proof*⟩

lemma *subst_range_Var[simp]: "subst_range Var = {}"* ⟨*proof*⟩

lemma *range_vars_Var[simp]: "range_vars Var = {}"* ⟨*proof*⟩

lemma *finite_subst_img_if_finite_dom: "finite (subst_domain σ) ⟹ finite (range_vars σ)"*
⟨*proof*⟩

lemma *finite_subst_img_if_finite_dom': "finite (subst_domain σ) ⟹ finite (subst_range σ)"*
⟨*proof*⟩

lemma *subst_img_alt_def: "subst_range s = {t. ∃v. s v = t ∧ t ≠ Var v}"*
⟨*proof*⟩

lemma *subst_fv_img_alt_def: "range_vars s = (⋃t ∈ {t. ∃v. s v = t ∧ t ≠ Var v}. fv t)"*
⟨*proof*⟩

lemma *subst_domI[intro]: "σ v ≠ Var v ⟹ v ∈ subst_domain σ"*
⟨*proof*⟩

lemma *subst_imgI[intro]: "σ v ≠ Var v ⟹ σ v ∈ subst_range σ"*
⟨*proof*⟩

lemma *subst_fv_imgI[intro]: "σ v ≠ Var v ⟹ fv (σ v) ⊆ range_vars σ"*
⟨*proof*⟩

lemma *subst_domain_subst_Fun_single[simp]:*
"subst_domain (Var(x := Fun f T)) = {x}" (**is** "?A = ?B")
⟨*proof*⟩

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

lemma *range_vars_subst_Fun_single[simp]:*
"range_vars (Var(x := Fun f T)) = fv (Fun f T)"
⟨*proof*⟩

lemma *var_renaming_is_Fun_iff:*
assumes "subst_range δ ⊆ range Var"
shows "is_Fun t = is_Fun (t · δ)"
⟨*proof*⟩

lemma *subst_fv_dom_img_subset:* "fv t ⊆ subst_domain ϑ ⟹ fv (t · ϑ) ⊆ range_vars ϑ"
⟨*proof*⟩

lemma *subst_fv_dom_img_subset_set:* "fv_{set} M ⊆ subst_domain ϑ ⟹ fv_{set} (M $\cdot_{set}$ ϑ) ⊆ range_vars ϑ"
⟨*proof*⟩

lemma *subst_fv_dom_ground_if_ground_img:*
assumes "fv t ⊆ subst_domain s" "ground (subst_range s)"
shows "fv (t · s) = {}"
⟨*proof*⟩

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) = {}"
⟨*proof*⟩

lemma *subst_fv_unfold:* "fv (t · s) = (fv t - subst_domain s) ∪ fv_{set} (s ` (fv t ∩ subst_domain s))"
⟨*proof*⟩

lemma *subst_fv_unfold_ground_img:* "range_vars s = {} ⟹ fv (t · s) = fv t - subst_domain s"
⟨*proof*⟩

lemma *subst_img_update:*
"⟦σ v = Var v; t ≠ Var v⟧ ⟹ range_vars (σ(v := t)) = range_vars σ ∪ fv t"
⟨*proof*⟩

lemma *subst_dom_update1:* "v ∉ subst_domain σ ⟹ subst_domain (σ(v := Var v)) = subst_domain σ"
⟨*proof*⟩

lemma *subst_dom_update2:* "t ≠ Var v ⟹ subst_domain (σ(v := t)) = insert v (subst_domain σ)"
⟨*proof*⟩

lemma *subst_dom_update3:* "t = Var v ⟹ subst_domain (σ(v := t)) = subst_domain σ - {v}"
⟨*proof*⟩

lemma *var_not_in_subst_dom[elim]:* "v ∉ subst_domain s ⟹ s v = Var v"
⟨*proof*⟩

lemma *subst_dom_vars_in_subst[elim]:* "v ∈ subst_domain s ⟹ s v ≠ Var v"
⟨*proof*⟩

lemma *subst_not_dom_fixed:* "⟦v ∈ fv t; v ∉ subst_domain s⟧ ⟹ v ∈ fv (t · s)" ⟨*proof*⟩

lemma *subst_not_img_fixed:* "⟦v ∈ fv (t · s); v ∉ range_vars s⟧ ⟹ v ∈ fv t"
⟨*proof*⟩

lemma *ground_range_vars[intro]: "ground (subst_range s) ⟹ range_vars s = {}"*
⟨*proof*⟩

lemma *ground_subst_no_var[intro]: "ground (subst_range s) ⟹ x ∉ range_vars s"*
⟨*proof*⟩

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*⟩

lemma *ground_term_subst_domain_fv_subset:*
"fv (t · δ) = {} ⟹ fv t ⊆ subst_domain δ"
⟨*proof*⟩

lemma *ground_subst_range_empty_fv:*
"ground (subst_range ϑ) ⟹ x ∈ subst_domain ϑ ⟹ fv (ϑ x) = {}"
⟨*proof*⟩

lemma *subst_Var_notin_img: "x ∉ range_vars s ⟹ t · s = Var x ⟹ t = Var x"*
⟨*proof*⟩

lemma *fv_in_subst_img: "⟦s v = t; t ≠ Var v⟧ ⟹ fv t ⊆ range_vars s"*
⟨*proof*⟩

lemma *empty_dom_iff_empty_subst: "subst_domain ϑ = {} ⟷ ϑ = Var"* ⟨*proof*⟩

lemma *subst_dom_cong: "(⋀v t. ϑ v = t ⟹ δ v = t) ⟹ subst_domain ϑ ⊆ subst_domain δ"*
⟨*proof*⟩

lemma *subst_img_cong: "(⋀v t. ϑ v = t ⟹ δ v = t) ⟹ range_vars ϑ ⊆ range_vars δ"*
⟨*proof*⟩

lemma *subst_dom_elim: "subst_domain s ∩ range_vars s = {} ⟹ fv (t · s) ∩ subst_domain s = {}"*
⟨*proof*⟩

lemma *subst_dom_insert_finite: "finite (subst_domain s) = finite (subst_domain (s(v := t)))"*
⟨*proof*⟩

lemma *trm_subst_disj: "t · ϑ = t ⟹ fv t ∩ subst_domain ϑ = {}"*
⟨*proof*⟩

lemma *trm_subst_ident[intro]: "fv t ∩ subst_domain ϑ = {} ⟹ t · ϑ = t"*
⟨*proof*⟩

lemma *trm_subst_ident'[intro]: "v ∉ subst_domain ϑ ⟹ (Var v) · ϑ = Var v"*
⟨*proof*⟩

lemma *trm_subst_ident''[intro]: "(⋀x. x ∈ fv t ⟹ ϑ x = Var x) ⟹ t · ϑ = t"*
⟨*proof*⟩

lemma *set_subst_ident: "fv_{set} M ∩ subst_domain ϑ = {} ⟹ M $\cdot_{set}$ ϑ = M"*
⟨*proof*⟩

lemma *trm_subst_ident_subterms[intro]:*
"fv t ∩ subst_domain ϑ = {} ⟹ subterms t $\cdot_{set}$ ϑ = subterms t"
⟨*proof*⟩

lemma *trm_subst_ident_subterms'[intro]:*
"v ∉ fv t ⟹ subterms t $\cdot_{set}$ Var(v := s) = subterms t"
⟨*proof*⟩

lemma *const_mem_subst_cases:*
  assumes "Fun c [] $\in$ M $\cdot_{set}$ ϑ"
  shows "Fun c [] $\in$ M $\vee$ Fun c [] $\in$ ϑ ‘ fv$_{set}$ M"
⟨*proof*⟩

lemma *const_mem_subst_cases':*
  assumes "Fun c [] $\in$ M $\cdot_{set}$ ϑ"
  shows "Fun c [] $\in$ M $\vee$ Fun c [] $\in$ subst_range ϑ"
⟨*proof*⟩

lemma *fv_subterms_substI[intro]:* "y $\in$ fv t $\Longrightarrow$ ϑ y $\in$ subterms t $\cdot_{set}$ ϑ"
⟨*proof*⟩

lemma *fv_subterms_subst_eq[simp]:* "fv$_{set}$ (subterms (t $\cdot$ ϑ)) = fv$_{set}$ (subterms t $\cdot_{set}$ ϑ)"
⟨*proof*⟩

lemma *fv_subterms_set_subst:* "fv$_{set}$ (subterms$_{set}$ M $\cdot_{set}$ ϑ) = fv$_{set}$ (subterms$_{set}$ (M $\cdot_{set}$ ϑ))"
⟨*proof*⟩

lemma *fv_subterms_set_subst':* "fv$_{set}$ (subterms$_{set}$ M $\cdot_{set}$ ϑ) = fv$_{set}$ (M $\cdot_{set}$ ϑ)"
⟨*proof*⟩

lemma *fv_subst_subset:* "x $\in$ fv t $\Longrightarrow$ fv (ϑ x) $\subseteq$ fv (t $\cdot$ ϑ)"
⟨*proof*⟩

lemma *fv_subst_subset':* "fv s $\subseteq$ fv t $\Longrightarrow$ fv (s $\cdot$ ϑ) $\subseteq$ fv (t $\cdot$ ϑ)"
⟨*proof*⟩

lemma *fv_subst_obtain_var:*
  fixes δ::"('a,'b) subst"
  assumes "x $\in$ fv (t $\cdot$ δ)"
  shows "$\exists$y $\in$ fv t. x $\in$ fv (δ y)"
⟨*proof*⟩

lemma *set_subst_all_ident:* "fv$_{set}$ (M $\cdot_{set}$ ϑ) $\cap$ subst_domain δ = {} $\Longrightarrow$ M $\cdot_{set}$ (ϑ $\circ_s$ δ) = M $\cdot_{set}$ ϑ"
⟨*proof*⟩

lemma *subterms_subst:*
  "subterms (t $\cdot$ d) = (subterms t $\cdot_{set}$ d) $\cup$ subterms$_{set}$ (d ‘ (fv t $\cap$ subst_domain d))"
⟨*proof*⟩

lemma *subterms_subst':*
  fixes ϑ::"('a,'b) subst"
  assumes "$\forall$x $\in$ fv t. ($\exists$f. ϑ x = Fun f []) $\vee$ ($\exists$y. ϑ x = Var y)"
  shows "subterms (t $\cdot$ ϑ) = subterms t $\cdot_{set}$ ϑ"
⟨*proof*⟩

lemma *subterms_subst'':*
  fixes ϑ::"('a,'b) subst"
  assumes "$\forall$x $\in$ fv$_{set}$ M. ($\exists$f. ϑ x = Fun f []) $\vee$ ($\exists$y. ϑ x = Var y)"
  shows "subterms$_{set}$ (M $\cdot_{set}$ ϑ) = subterms$_{set}$ M $\cdot_{set}$ ϑ"
⟨*proof*⟩

lemma *subterms_subst_subterm:*
  fixes ϑ::"('a,'b) subst"
  assumes "$\forall$x $\in$ fv a. ($\exists$f. ϑ x = Fun f []) $\vee$ ($\exists$y. ϑ x = Var y)"
    and "b $\in$ subterms (a $\cdot$ ϑ)"
  shows "$\exists$c $\in$ subterms a. c $\cdot$ ϑ = b"
⟨*proof*⟩

lemma *subterms_subst_subset:* "subterms t $\cdot_{set}$ σ $\subseteq$ subterms (t $\cdot$ σ)"
⟨*proof*⟩

lemma subterms_subst_subset': "subterms$_{set}$ M $\cdot_{set}$ σ ⊆ subterms$_{set}$ (M $\cdot_{set}$ σ)"
⟨*proof*⟩

lemma subterms$_{set}$_subst:
  fixes ϑ::"('a,'b) subst"
  assumes "t ∈ subterms$_{set}$ (M $\cdot_{set}$ ϑ)"
  shows "t ∈ subterms$_{set}$ M $\cdot_{set}$ ϑ ∨ (∃x ∈ fv$_{set}$ M. t ∈ subterms (ϑ x))"
⟨*proof*⟩

lemma rm_vars_dom: "subst_domain (rm_vars V s) = subst_domain s - V"
⟨*proof*⟩

lemma rm_vars_dom_subset: "subst_domain (rm_vars V s) ⊆ subst_domain s"
⟨*proof*⟩

lemma rm_vars_dom_eq':
  "subst_domain (rm_vars (UNIV - V) s) = subst_domain s ∩ V"
⟨*proof*⟩

lemma rm_vars_img: "subst_range (rm_vars V s) = s ` subst_domain (rm_vars V s)"
⟨*proof*⟩

lemma rm_vars_img_subset: "subst_range (rm_vars V s) ⊆ subst_range s"
⟨*proof*⟩

lemma rm_vars_img_fv_subset: "range_vars (rm_vars V s) ⊆ range_vars s"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma rm_vars_apply: "v ∈ subst_domain (rm_vars V s) ⟹ (rm_vars V s) v = s v"
⟨*proof*⟩

lemma rm_vars_apply': "subst_domain δ ∩ vs = {} ⟹ rm_vars vs δ = δ"
⟨*proof*⟩

lemma rm_vars_ident: "fv t ∩ vs = {} ⟹ t · (rm_vars vs ϑ) = t · ϑ"
⟨*proof*⟩

lemma rm_vars_fv_subset: "fv (t · rm_vars X ϑ) ⊆ fv t ∪ fv (t · ϑ)"
⟨*proof*⟩

lemma rm_vars_fv_disj:
  assumes "fv t ∩ X = {}" "fv (t · ϑ) ∩ X = {}"
  shows "fv (t · rm_vars X ϑ) ∩ X = {}"
⟨*proof*⟩

lemma rm_vars_ground_supports:
  assumes "ground (subst_range ϑ)"
  shows "rm_vars X ϑ supports ϑ"
⟨*proof*⟩

lemma rm_vars_split:
  assumes "ground (subst_range ϑ)"
  shows "ϑ = rm_vars X ϑ $\circ_s$ rm_vars (subst_domain ϑ - X) ϑ"
⟨*proof*⟩

lemma rm_vars_fv_img_disj:
  assumes "fv t ∩ X = {}" "X ∩ range_vars ϑ = {}"
  shows "fv (t · rm_vars X ϑ) ∩ X = {}"

⟨proof⟩

lemma *subst_apply_dom_ident: "t · ϑ = t ⟹ subst_domain δ ⊆ subst_domain ϑ ⟹ t · δ = t"*
⟨proof⟩

lemma *rm_vars_subst_apply_ident:*
  assumes *"t · ϑ = t"*
  shows *"t · (rm_vars vs ϑ) = t"*
⟨proof⟩

lemma *rm_vars_subst_eq:*
  "t · δ = t · rm_vars (subst_domain δ - subst_domain δ ∩ fv t) δ"
⟨proof⟩

lemma *rm_vars_subst_eq':*
  "t · δ = t · rm_vars (UNIV - fv t) δ"
⟨proof⟩

lemma *rm_vars_comp:*
  assumes *"range_vars δ ∩ vs = {}"*
  shows *"t · rm_vars vs (δ $\circ_s$ ϑ) = t · (rm_vars vs δ $\circ_s$ rm_vars vs ϑ)"*
⟨proof⟩

lemma *rm_vars_fv$_{set}$_subst:*
  assumes *"x ∈ fv$_{set}$ (rm_vars X ϑ ‘ Y)"*
  shows *"x ∈ fv$_{set}$ (ϑ ‘ Y) ∨ x ∈ X"*
⟨proof⟩

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⟩

lemma *subst_sends_dom_to_img: "v ∈ subst_domain ϑ ⟹ fv (Var v · ϑ) ⊆ range_vars ϑ"*
⟨proof⟩

lemma *subst_sends_fv_to_img: "fv (t · s) ⊆ fv t ∪ range_vars s"*
⟨proof⟩

lemma *ident_comp_subst_trm_if_disj:*
  assumes *"subst_domain σ ∩ range_vars ϑ = {}" "v ∈ subst_domain ϑ"*
  shows *"(ϑ $\circ_s$ σ) v = ϑ v"*
⟨proof⟩

lemma *ident_comp_subst_trm_if_disj': "fv (ϑ v) ∩ subst_domain σ = {} ⟹ (ϑ $\circ_s$ σ) v = ϑ v"*
⟨proof⟩

lemma *subst_idemI[intro]: "subst_domain σ ∩ range_vars σ = {} ⟹ subst_idem σ"*
⟨proof⟩

lemma *subst_idemI'[intro]: "ground (subst_range σ) ⟹ subst_idem σ"*
⟨proof⟩

lemma *subst_idemE: "subst_idem σ ⟹ subst_domain σ ∩ range_vars σ = {}"*
⟨proof⟩

lemma *subst_idem_rm_vars: "subst_idem ϑ ⟹ subst_idem (rm_vars X ϑ)"*
⟨proof⟩

lemma *subst_fv_bounded_if_img_bounded: "range_vars ϑ ⊆ fv t ∪ V ⟹ fv (t · ϑ) ⊆ fv t ∪ V"*
⟨proof⟩

lemma *subst_fv_bound_singleton: "fv (t · Var(v := t')) ⊆ fv t ∪ fv t'"*

```
⟨proof⟩

lemma subst_fv_bounded_if_img_bounded':
  assumes "range_vars ϑ ⊆ fv_set M"
  shows "fv_set (M ·_set ϑ) ⊆ fv_set M"
⟨proof⟩

lemma ground_img_if_ground_subst: "(⋀v t. s v = t ⟹ fv t = {}) ⟹ range_vars s = {}"
⟨proof⟩

lemma ground_subst_fv_subset: "ground (subst_range ϑ) ⟹ fv (t · ϑ) ⊆ fv t"
⟨proof⟩

lemma ground_subst_fv_subset': "ground (subst_range ϑ) ⟹ fv_set (M ·_set ϑ) ⊆ fv_set M"
⟨proof⟩

lemma subst_to_var_is_var[elim]: "t · s = Var v ⟹ ∃w. t = Var w"
⟨proof⟩

lemma subst_dom_comp_inI:
  assumes "y ∉ subst_domain σ"
    and "y ∈ subst_domain δ"
  shows "y ∈ subst_domain (σ ∘_s δ)"
⟨proof⟩

lemma subst_comp_notin_dom_eq:
  "x ∉ subst_domain ϑ1 ⟹ (ϑ1 ∘_s ϑ2) x = ϑ2 x"
⟨proof⟩

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⟩

lemma subst_img_comp_subset[simp]:
  "range_vars (ϑ1 ∘_s ϑ2) ⊆ range_vars ϑ1 ∪ range_vars ϑ2"
⟨proof⟩

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⟩

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⟩

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⟩

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⟩

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⟩

```
lemma subst_img_comp_subset_ground:
  assumes "ground (subst_range ϑ1)"
  shows "subst_range (ϑ1 ∘s ϑ2) ⊆ subst_range ϑ1 ∪ subst_range ϑ2"
⟨proof⟩

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⟩

lemma subst_comp_upd1:
  "ϑ(v := t) ∘s σ = (ϑ ∘s σ)(v := t · σ)"
⟨proof⟩

lemma subst_comp_upd2:
  assumes "v ∉ subst_domain s" "v ∉ range_vars s"
  shows "s(v := t) = s ∘s (Var(v := t))"
⟨proof⟩

lemma ground_subst_dom_iff_img:
  "ground (subst_range σ) ⟹ x ∈ subst_domain σ ⟷ σ x ∈ subst_range σ"
⟨proof⟩

lemma finite_dom_subst_exists:
  "finite S ⟹ ∃σ::('f,'v) subst. subst_domain σ = S"
⟨proof⟩

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⟩

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⟩

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⟩

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⟩

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⟩

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)* $\circ_s$ *(rm_vars (subst_domain ϑ - V) ϑ)"* (**is** *?Q*)
⟨*proof*⟩

lemma *subst_comp_eq_if_disjoint_vars:*
assumes *"(subst_domain δ ∪ range_vars δ) ∩ (subst_domain γ ∪ range_vars γ) = {}"*
shows *"γ* $\circ_s$ *δ = δ* $\circ_s$ *γ"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *subst_img_bound: "subst_domain δ ∪ range_vars δ ⊆ fv t ⟹ range_vars δ ⊆ fv (t · δ)"*
⟨*proof*⟩

lemma *subst_all_fv_subset: "fv t ⊆* fv_{set} *M ⟹ fv (t · ϑ) ⊆* fv_{set} *(M* $\cdot_{set}$ *ϑ)"*
⟨*proof*⟩

lemma *subst_support_if_mgt_subst_idem:*
assumes *"ϑ* $\preceq_\circ$ *δ" "subst_idem ϑ"*
shows *"ϑ supports δ"*
⟨*proof*⟩

lemma *subst_support_iff_mgt_if_subst_idem:*
assumes *"subst_idem ϑ"*
shows *"ϑ* $\preceq_\circ$ *δ ⟷ ϑ supports δ"*
⟨*proof*⟩

lemma *subst_support_comp:*
fixes *ϑ δ* $\mathcal{I}$*::"('a,'b) subst"*
assumes *"ϑ supports* $\mathcal{I}$*" "δ supports* $\mathcal{I}$*"*
shows *"(ϑ* $\circ_s$ *δ) supports* $\mathcal{I}$*"*
⟨*proof*⟩

lemma *subst_support_comp':*
fixes *ϑ δ σ::"('a,'b) subst"*
assumes *"ϑ supports δ"*
shows *"ϑ supports (δ* $\circ_s$ *σ)" "σ supports δ ⟹ ϑ supports (σ* $\circ_s$ *δ)"*
⟨*proof*⟩

lemma *subst_support_comp_split:*
fixes *ϑ δ* $\mathcal{I}$*::"('a,'b) subst"*
assumes *"(ϑ* $\circ_s$ *δ) supports* $\mathcal{I}$*"*
shows *"subst_domain ϑ ∩ range_vars ϑ = {} ⟹ ϑ supports* $\mathcal{I}$*"*
and *"subst_domain ϑ ∩ subst_domain δ = {} ⟹ δ supports* $\mathcal{I}$*"*
⟨*proof*⟩

lemma *subst_idem_support: "subst_idem ϑ ⟹ ϑ supports ϑ* $\circ_s$ *δ"*
⟨*proof*⟩

lemma *subst_idem_iff_self_support: "subst_idem ϑ ⟷ ϑ supports ϑ"*
⟨*proof*⟩

lemma *subterm_subst_neq: "t ⊏ t' ⟹ t · s ≠ t' · s"*
⟨*proof*⟩

lemma *fv_Fun_subst_neq: "x ∈ fv (Fun f T) ⟹ σ x ≠ Fun f T · σ"*
⟨*proof*⟩

lemma *subterm_subst_unfold:*
assumes *"t ⊑ s · ϑ"*

shows *"(∃s'. s' ⊑ s ∧ t = s' · ϑ) ∨ (∃x ∈ fv s. t ⊏ ϑ x)"*
⟨*proof*⟩

lemma *subterm_subst_img_subterm:*
assumes *"t ⊑ s · ϑ" "⋀s'. s' ⊑ s ⟹ t ≠ s' · ϑ"*
shows *"∃w ∈ fv s. t ⊏ ϑ w"*
⟨*proof*⟩

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*⟩

lemma *subst_apply_img_var:*
assumes *"v ∈ fv (t · δ)" "v ∉ fv t"*
obtains *w* **where** *"w ∈ fv t" "v ∈ fv (δ w)"*
⟨*proof*⟩

lemma *subst_apply_img_var':*
assumes *"x ∈ fv (t · δ)" "x ∉ fv t"*
shows *"∃y ∈ fv t. x ∈ fv (δ y)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)"
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *subst_subterm'':*
assumes *"s ∈ subterms (t · ϑ)"*
shows *"(∃u ∈ subterms t. s = u · ϑ) ∨ s ∈ subterms$_{set}$ (subst_range ϑ)"*
⟨*proof*⟩

2.3.3 More Small Lemmata

lemma *funs_term_subst: "funs_term (t · ϑ) = funs_term t ∪ (⋃x ∈ fv t. funs_term (ϑ x))"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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⟩

lemma *const_subterm_subst:* "Fun c [] $\sqsubseteq$ t $\Longrightarrow$ Fun c [] $\sqsubseteq$ t $\cdot$ σ"
⟨proof⟩

lemma *const_subterm_subst_var_obtain:*
  assumes "Fun c [] $\sqsubseteq$ t $\cdot$ σ" "$\neg$Fun c [] $\sqsubseteq$ t"
  obtains x **where** "x $\in$ fv t" "Fun c [] $\sqsubseteq$ σ x"
⟨proof⟩

lemma *const_subterm_subst_cases:*
  assumes "Fun c [] $\sqsubseteq$ t $\cdot$ σ"
  shows "Fun c [] $\sqsubseteq$ t $\vee$ ($\exists$x $\in$ fv t. x $\in$ subst_domain σ $\wedge$ Fun c [] $\sqsubseteq$ σ x)"
⟨proof⟩

lemma *fv_{pairs}_subst_fv_subset:*
  assumes "x $\in$ fv_{pairs} F"
  shows "fv (ϑ x) $\subseteq$ fv_{pairs} (F $\cdot_{pairs}$ ϑ)"
  ⟨proof⟩

lemma *fv_{pairs}_step_subst:* "fv_{set} (δ ‘ fv_{pairs} F) = fv_{pairs} (F $\cdot_{pairs}$ δ)"
⟨proof⟩

lemma *fv_{pairs}_subst_obtain_var:*
  fixes δ::"('a,'b) subst"
  assumes "x $\in$ fv_{pairs} (F $\cdot_{pairs}$ δ)"
  shows "$\exists$y $\in$ fv_{pairs} F. x $\in$ fv (δ y)"
  ⟨proof⟩

lemma *pair_subst_ident[intro]:* "(fv t $\cup$ fv t') $\cap$ subst_domain ϑ = {} $\Longrightarrow$ (t,t') $\cdot_p$ ϑ = (t,t')"
⟨proof⟩

lemma *pairs_substI[intro]:*
  assumes "subst_domain ϑ $\cap$ ($\bigcup$(s,t) $\in$ M. fv s $\cup$ fv t) = {}"
  shows "M $\cdot_{pset}$ ϑ = M"
⟨proof⟩

lemma *fv_{pairs}_subst:* "fv_{pairs} (F $\cdot_{pairs}$ ϑ) = fv_{set} (ϑ ‘ (fv_{pairs} F))"
⟨proof⟩

lemma *fv_{pairs}_subst_subset:*
  assumes "fv_{pairs} (F $\cdot_{pairs}$ δ) $\subseteq$ subst_domain σ"
  shows "fv_{pairs} F $\subseteq$ subst_domain σ $\cup$ subst_domain δ"
  ⟨proof⟩

lemma *pairs_subst_comp:* "F $\cdot_{pairs}$ δ $\circ_s$ ϑ = ((F $\cdot_{pairs}$ δ) $\cdot_{pairs}$ ϑ)"
⟨proof⟩

lemma *pairs_substI'[intro]:*
  "subst_domain ϑ $\cap$ fv_{pairs} F = {} $\Longrightarrow$ F $\cdot_{pairs}$ ϑ = F"
⟨proof⟩

lemma *subst_pair_compose[simp]:* "d $\cdot_p$ (δ $\circ_s$ $\mathcal{I}$) = d $\cdot_p$ δ $\cdot_p$ $\mathcal{I}$"
⟨proof⟩

lemma *subst_pairs_compose[simp]:* "D $\cdot_{pset}$ (δ $\circ_s$ $\mathcal{I}$) = D $\cdot_{pset}$ δ $\cdot_{pset}$ $\mathcal{I}$"
⟨proof⟩

lemma *subst_apply_pair_pair:* "(t, s) $\cdot_p$ $\mathcal{I}$ = (t $\cdot$ $\mathcal{I}$, s $\cdot$ $\mathcal{I}$)"
⟨proof⟩

lemma *subst_apply_pairs_nil[simp]:* "[] $\cdot_{pairs}$ δ = []"
⟨proof⟩

lemma *subst_apply_pairs_singleton[simp]: "[(t,s)] $\cdot_{pairs}$ δ = [(t $\cdot$ δ,s $\cdot$ δ)]"*
⟨*proof*⟩

lemma *subst_apply_pairs_Var[iff]: "F $\cdot_{pairs}$ Var = F"* ⟨*proof*⟩

lemma *subst_apply_pairs_pset_subst: "set (F $\cdot_{pairs}$ ϑ) = set F $\cdot_{pset}$ ϑ"*
⟨*proof*⟩

2.3.4 Finite Substitutions

inductive_set *fsubst::"('a,'b) subst set"* **where**
fvar: "Var $\in$ fsubst"
| FUpdate: "$[\![\vartheta \in$ fsubst; v $\notin$ subst_domain ϑ; t $\neq$ Var v$]\!] \Longrightarrow \vartheta$(v := t) $\in$ fsubst"

lemma *finite_dom_iff_fsubst:*
"finite (subst_domain ϑ) $\longleftrightarrow$ $\vartheta \in$ fsubst"
⟨*proof*⟩

lemma *fsubst_induct[case_names fvar FUpdate, induct set: finite]:*
assumes *"finite (subst_domain δ)" "P Var"*
and *"$\bigwedge\vartheta$ v t. $[\![$finite (subst_domain ϑ); v $\notin$ subst_domain ϑ; t $\neq$ Var v; P $\vartheta]\!] \Longrightarrow$ P (ϑ(v := t))"*
shows *"P δ"*
⟨*proof*⟩

lemma *fun_upd_fsubst: "s(v := t) $\in$ fsubst $\longleftrightarrow$ s $\in$ fsubst"*
⟨*proof*⟩

lemma *finite_img_if_fsubst: "s $\in$ fsubst $\Longrightarrow$ finite (subst_range s)"*
⟨*proof*⟩

2.3.5 Unifiers and Most General Unifiers (MGUs)

abbreviation *Unifier::"('f,'v) subst $\Rightarrow$ ('f,'v) term $\Rightarrow$ ('f,'v) term $\Rightarrow$ bool"* **where**
"Unifier σ t u $\equiv$ (t $\cdot$ σ = u $\cdot$ σ)"

abbreviation *MGU::"('f,'v) subst $\Rightarrow$ ('f,'v) term $\Rightarrow$ ('f,'v) term $\Rightarrow$ bool"* **where**
"MGU σ t u $\equiv$ Unifier σ t u $\wedge$ ($\forall\vartheta$. Unifier ϑ t u $\longrightarrow$ $\sigma \preceq_\circ \vartheta$)"

lemma *MGUI[intro]:*
shows *"$[\![$t $\cdot$ σ = u $\cdot$ σ; $\bigwedge\vartheta$::('f,'v) subst. t $\cdot$ ϑ = u $\cdot$ $\vartheta \Longrightarrow \sigma \preceq_\circ \vartheta]\!] \Longrightarrow$ MGU σ t u"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

lemma *MGU_sym[sym]: "MGU σ s t $\Longrightarrow$ MGU σ t s"* ⟨*proof*⟩
lemma *Unifier_sym[sym]: "Unifier σ s t $\Longrightarrow$ Unifier σ t s"* ⟨*proof*⟩

lemma *MGU_nil: "MGU Var s t $\longleftrightarrow$ s = t"* ⟨*proof*⟩

lemma *Unifier_comp: "Unifier ($\vartheta \circ_s \delta$) t u $\Longrightarrow$ Unifier δ (t $\cdot$ ϑ) (u $\cdot$ ϑ)"*
⟨*proof*⟩

lemma *Unifier_comp'*: "Unifier δ (t $\cdot$ ϑ) (u $\cdot$ ϑ) $\Longrightarrow$ Unifier (ϑ $\circ_s$ δ) t u"
⟨*proof*⟩

lemma *Unifier_excludes_subterm*:
  assumes ϑ: "Unifier ϑ t u"
  shows "$\neg$t $\sqsubset$ u"
⟨*proof*⟩

lemma *MGU_is_Unifier*: "MGU σ t u $\Longrightarrow$ Unifier σ t u" ⟨*proof*⟩

lemma *MGU_Var1*:
  assumes "$\neg$Var v $\sqsubset$ t"
  shows "MGU (Var(v := t)) (Var v) t"
⟨*proof*⟩

lemma *MGU_Var2*: "v $\notin$ fv t $\Longrightarrow$ MGU (Var(v := t)) (Var v) t"
⟨*proof*⟩

lemma *MGU_Var3*: "MGU Var (Var v) (Var w) $\longleftrightarrow$ v = w" ⟨*proof*⟩

lemma *MGU_Const1*: "MGU Var (Fun c []) (Fun d []) $\longleftrightarrow$ c = d" ⟨*proof*⟩

lemma *MGU_Const2*: "MGU ϑ (Fun c []) (Fun d []) $\Longrightarrow$ c = d" ⟨*proof*⟩

lemma *MGU_Fun*:
  assumes "MGU ϑ (Fun f X) (Fun g Y)"
  shows "f = g" "length X = length Y"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *Unifier_subst_idem_subst*:
  "subst_idem r $\Longrightarrow$ Unifier s (t $\cdot$ r) (u $\cdot$ r) $\Longrightarrow$ Unifier (r $\circ_s$ s) (t $\cdot$ r) (u $\cdot$ r)"
⟨*proof*⟩

lemma *subst_idem_comp*:
  "subst_idem r $\Longrightarrow$ Unifier s (t $\cdot$ r) (u $\cdot$ r) $\Longrightarrow$
    ($\bigwedge$q. Unifier q (t $\cdot$ r) (u $\cdot$ r) $\Longrightarrow$ s $\circ_s$ q = q) $\Longrightarrow$
    subst_idem (r $\circ_s$ s)"
⟨*proof*⟩

lemma *Unifier_mgt*: "$[\![$Unifier δ t u; δ $\preceq_\circ$ $\vartheta]\!]$ $\Longrightarrow$ Unifier ϑ t u" ⟨*proof*⟩

lemma *Unifier_support*: "$[\![$Unifier δ t u; δ supports $\vartheta]\!]$ $\Longrightarrow$ Unifier ϑ t u"
⟨*proof*⟩

lemma *MGU_mgt*: "$[\![$MGU σ t u; MGU δ t u$]\!]$ $\Longrightarrow$ σ $\preceq_\circ$ δ" ⟨*proof*⟩

lemma *Unifier_trm_fv_bound*:
  "$[\![$Unifier s t u; v $\in$ fv t$]\!]$ $\Longrightarrow$ v $\in$ subst_domain s $\cup$ range_vars s $\cup$ fv u"
⟨*proof*⟩

lemma *Unifier_rm_var*: "$[\![$Unifier ϑ s t; v $\notin$ fv s $\cup$ fv t$]\!]$ $\Longrightarrow$ Unifier (rm_var v ϑ) s t"
⟨*proof*⟩

lemma *Unifier_ground_rm_vars*:
  assumes "ground (subst_range s)" "Unifier (rm_vars X s) t t'"
  shows "Unifier s t t'"
⟨*proof*⟩

```
lemma Unifier_dom_restrict:
  assumes "Unifier s t t'" "fv t ∪ fv t' ⊆ S"
  shows "Unifier (rm_vars (UNIV - S) s) t t'"
⟨proof⟩
```

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" ⟨proof⟩

lemma wf_subst_properties: "ϑ ∈ wf_subst_set = wf_subst ϑ"
⟨proof⟩

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⟩

lemma wf_subst_fsubst: "wf_subst δ ⟹ δ ∈ fsubst"
⟨proof⟩

lemma wf_subst_nil: "wf_subst Var" ⟨proof⟩

lemma wf_MGU_nil: "MGU Var s t ⟹ wf_MGU Var s t"
⟨proof⟩

lemma wf_MGU_dom_bound: "wf_MGU ϑ s t ⟹ subst_domain ϑ ⊆ fv s ∪ fv t" ⟨proof⟩

lemma wf_subst_single:
  assumes "v ∉ fv t" "σ v = t" "⋀w. v ≠ w ⟹ σ w = Var w"
  shows "wf_subst σ"
⟨proof⟩

lemma wf_subst_reduction:
  "wf_subst s ⟹ wf_subst (rm_var v s)"
⟨proof⟩

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)"
⟨proof⟩

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 "$subst\_domain\ \vartheta 1 \cap range\_vars\ \vartheta 2$ = {}"
and "$range\_vars\ \vartheta 1 \cap subst\_domain\ \vartheta 2$ = {}"
shows "wf_{subst} (λv. if ϑ1 v = Var v then ϑ2 v else ϑ1 v)"
⟨*proof*⟩

lemma *wf_subst_elim_append:*
assumes "$wf_{subst}\ \vartheta$" "subst_elim ϑ v" "v $\notin$ fv t"
shows "subst_elim (ϑ(w := t)) v"
⟨*proof*⟩

lemma *wf_subst_elim_dom:*
assumes "$wf_{subst}\ \vartheta$"
shows "$\forall$ v $\in$ subst_domain ϑ. subst_elim ϑ v"
⟨*proof*⟩

lemma *wf_subst_support_iff_mgt:* "$wf_{subst}\ \vartheta \Longrightarrow \vartheta$ supports $\delta \longleftrightarrow \vartheta \preceq_\circ \delta$"
⟨*proof*⟩

2.3.7 Interpretations

abbreviation *$interpretation_{subst}$::"('a,'b) subst $\Rightarrow$ bool"* **where**
"$interpretation_{subst}\ \vartheta \equiv$ subst_domain ϑ = UNIV $\wedge$ ground (subst_range ϑ)"

lemma *interpretation_substI:*
"($\bigwedge$v. fv (ϑ v) = {}) $\Longrightarrow$ $interpretation_{subst}\ \vartheta$"
⟨*proof*⟩

lemma *interpretation_grounds[simp]:*
"$interpretation_{subst}\ \vartheta \Longrightarrow$ fv (t $\cdot$ ϑ) = {}"
⟨*proof*⟩

lemma *interpretation_grounds_all:*
"$interpretation_{subst}\ \vartheta \Longrightarrow$ ($\bigwedge$v. fv (ϑ v) = {})"
⟨*proof*⟩

lemma *interpretation_grounds_all':*
"$interpretation_{subst}\ \vartheta \Longrightarrow$ ground (M $\cdot_{set}$ ϑ)"
⟨*proof*⟩

lemma *interpretation_comp:*
assumes "$interpretation_{subst}\ \vartheta$"
shows "$interpretation_{subst}$ ($\sigma \circ_s \vartheta$)" "$interpretation_{subst}$ ($\vartheta \circ_s \sigma$)"
⟨*proof*⟩

lemma *interpretation_subst_exists:*
"$\exists\mathcal{I}$::('f,'v) subst. $interpretation_{subst}\ \mathcal{I}$"
⟨*proof*⟩

lemma *interpretation_subst_exists':*
"$\exists\vartheta$::('f,'v) subst. subst_domain ϑ = X $\wedge$ ground (subst_range ϑ)"
⟨*proof*⟩

lemma *interpretation_subst_idem:*
"$interpretation_{subst}\ \vartheta \Longrightarrow$ subst_idem ϑ"
⟨*proof*⟩

lemma *subst_idem_comp_upd_eq:*
assumes "v $\notin$ subst_domain $\mathcal{I}$" "subst_idem ϑ"
shows "$\mathcal{I} \circ_s \vartheta = \mathcal{I}$(v := ϑ v) $\circ_s \vartheta$"
⟨*proof*⟩

lemma *interpretation_dom_img_disjoint:*
"$interpretation_{subst}\ \mathcal{I} \Longrightarrow$ subst_domain $\mathcal{I} \cap$ range_vars $\mathcal{I}$ = {}"

⟨proof⟩

2.3.8 Basic Properties of MGUs

lemma *MGU_is_mgu_singleton: "MGU ϑ t u = is_mgu ϑ {(t,u)}"*
⟨proof⟩

lemma *Unifier_in_unifiers_singleton: "Unifier ϑ s t $\longleftrightarrow$ ϑ $\in$ unifiers {(s,t)}"*
⟨proof⟩

lemma *subst_list_singleton_fv_subset:*
"($\bigcup$x $\in$ set (subst_list (subst v t) E). fv (fst x) $\cup$ fv (snd x))
$\subseteq$ fv t $\cup$ ($\bigcup$x $\in$ set E. fv (fst x) $\cup$ fv (snd x))"
⟨proof⟩

lemma *subst_of_dom_subset: "subst_domain (subst_of L) $\subseteq$ set (map fst L)"*
⟨proof⟩

lemma *wf_MGU_is_imgu_singleton: "wf$_{MGU}$ ϑ s t $\Longrightarrow$ is_imgu ϑ {(s,t)}"*
⟨proof⟩

lemma *mgu_subst_range_vars:*
assumes *"mgu s t = Some σ"* **shows** *"range_vars σ $\subseteq$ vars_term s $\cup$ vars_term t"*
⟨proof⟩

lemma *mgu_subst_domain_range_vars_disjoint:*
assumes *"mgu s t = Some σ"* **shows** *"subst_domain σ $\cap$ range_vars σ = {}"*
⟨proof⟩

lemma *mgu_same_empty: "mgu (t::('a,'b) term) t = Some Var"*
⟨proof⟩

lemma *mgu_var:* **assumes** *"x $\notin$ fv t"* **shows** *"mgu (Var x) t = Some (Var(x := t))"*
⟨proof⟩

lemma *mgu_gives_wellformed_subst:*
assumes *"mgu s t = Some ϑ"* **shows** *"wf$_{subst}$ ϑ"*
⟨proof⟩

lemma *mgu_gives_wellformed_MGU:*
assumes *"mgu s t = Some ϑ"* **shows** *"wf$_{MGU}$ ϑ s t"*
⟨proof⟩

lemma *mgu_vars_bounded[dest?]:*
"mgu M N = Some σ $\Longrightarrow$ subst_domain σ $\cup$ range_vars σ $\subseteq$ fv M $\cup$ fv N"
⟨proof⟩

lemma *mgu_gives_subst_idem: "mgu s t = Some ϑ $\Longrightarrow$ subst_idem ϑ"*
⟨proof⟩

lemma *mgu_always_unifies: "Unifier ϑ M N $\Longrightarrow$ $\exists\delta$. mgu M N = Some δ"*
⟨proof⟩

lemma *mgu_gives_MGU: "mgu s t = Some ϑ $\Longrightarrow$ MGU ϑ s t"*
⟨proof⟩

lemma *mgu_eliminates[dest?]:*
assumes *"mgu M N = Some σ"*
shows *"($\exists$v $\in$ fv M $\cup$ fv N. subst_elim σ v) $\vee$ σ = Var"*
(**is** *"?P M N σ"*)
⟨proof⟩

lemma *mgu_eliminates_dom:*

```
  assumes "mgu x y = Some ϑ" "v ∈ subst_domain ϑ"
  shows "subst_elim ϑ v"
⟨proof⟩

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)"
⟨proof⟩

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"
⟨proof⟩

lemma mgu_None_if_neq_ground:
  assumes "t ≠ t'" "fv t = {}" "fv t' = {}"
  shows "mgu t t' = None"
⟨proof⟩

lemma mgu_None_commutes:
  "mgu s t = None ⟹ mgu t s = None"
⟨proof⟩

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⟩

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⟩

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"
⟨proof⟩

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⟩
```

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"
⟨proof⟩

lemma subst_inj_on_is_bij_betw:
  "inj_on ϑ (subst_domain ϑ) = bij_betw ϑ (subst_domain ϑ) (subst_range ϑ)"
⟨proof⟩
```

```
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")
⟨proof⟩

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")
⟨proof⟩

lemma subterm_inj_on_subset:
  assumes "subterm_inj_on f A"
    and "B ⊆ A"
  shows "subterm_inj_on f B"
⟨proof⟩

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⟩

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⟩

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⟩
```

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⟩
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⟩
```

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⟩
```

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)"
⟨proof⟩

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) ⟹ fv_set (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 $\cdot_{list}$ δ,R $\cdot_{list}$ δ)"
begin

lemma *Ana_subterm:* **assumes** "Ana t = (K,T)" **shows** "set T ⊂ subterms t"
⟨*proof*⟩

lemma *Ana_subterm':* "s ∈ set (snd (Ana t)) ⟹ s ⊑ t"
⟨*proof*⟩

lemma *Ana_vars:* **assumes** "Ana t = (K,M)" **shows** "fv_{set} (set K) ⊆ fv t" "fv_{set} (set M) ⊆ fv t"
⟨*proof*⟩

abbreviation $\mathcal{V}$ **where** "$\mathcal{V}$ ≡ UNIV::'var set"
abbreviation Σn ("Σ^-") **where** "Σ^n ≡ {f::'fun. arity f = n}"
abbreviation Σnpub ("Σ_{pub}^-") **where** "${\Sigma_{pub}}^n$ ≡ {f. public f} ∩ Σ^n"
abbreviation Σnpriv ("Σ_{priv}^-") **where** "${\Sigma_{priv}}^n$ ≡ {f. ¬public f} ∩ Σ^n"
abbreviation Σ_{pub} **where** "Σ_{pub} ≡ (⋃n. ${\Sigma_{pub}}^n$)"
abbreviation Σ_{priv} **where** "Σ_{priv} ≡ (⋃n. ${\Sigma_{priv}}^n$)"
abbreviation Σ **where** "Σ ≡ (⋃n. Σ^n)"
abbreviation $\mathcal{C}$ **where** "$\mathcal{C}$ ≡ Σ^0"
abbreviation $\mathcal{C}_{pub}$ **where** "$\mathcal{C}_{pub}$ ≡ {f. public f} ∩ $\mathcal{C}$"
abbreviation $\mathcal{C}_{priv}$ **where** "$\mathcal{C}_{priv}$ ≡ {f. ¬public f} ∩ $\mathcal{C}$"
abbreviation Σ_f **where** "Σ_f ≡ Σ - $\mathcal{C}$"
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 ∩ $\mathcal{C}$ = {}" ⟨*proof*⟩
lemma *id_union_univ:* "Σ_f ∪ $\mathcal{C}$ = UNIV" "Σ = UNIV" ⟨*proof*⟩
lemma *const_arity_eq_zero[dest]:* "c ∈ $\mathcal{C}$ ⟹ arity c = 0" ⟨*proof*⟩
lemma *const_pub_arity_eq_zero[dest]:* "c ∈ $\mathcal{C}_{pub}$ ⟹ arity c = 0 ∧ public c" ⟨*proof*⟩
lemma *const_priv_arity_eq_zero[dest]:* "c ∈ $\mathcal{C}_{priv}$ ⟹ arity c = 0 ∧ ¬public c" ⟨*proof*⟩
lemma *fun_arity_gt_zero[dest]:* "f ∈ Σ_f ⟹ arity f > 0" ⟨*proof*⟩
lemma *pub_fun_public[dest]:* "f ∈ Σ_{fpub} ⟹ public f" ⟨*proof*⟩
lemma *pub_fun_arity_gt_zero[dest]:* "f ∈ Σ_{fpub} ⟹ arity f > 0" ⟨*proof*⟩

lemma *Σ_f_unfold:* "Σ_f = {f::'fun. arity f > 0}" ⟨*proof*⟩
lemma *$\mathcal{C}$_unfold:* "$\mathcal{C}$ = {f::'fun. arity f = 0}" ⟨*proof*⟩
lemma *$\mathcal{C}$pub_unfold:* "$\mathcal{C}_{pub}$ = {f::'fun. arity f = 0 ∧ public f}" ⟨*proof*⟩
lemma *$\mathcal{C}$priv_unfold:* "$\mathcal{C}_{priv}$ = {f::'fun. arity f = 0 ∧ ¬public f}" ⟨*proof*⟩
lemma *Σnpub_unfold:* "(${\Sigma_{pub}}^n$) = {f::'fun. arity f = n ∧ public f}" ⟨*proof*⟩
lemma *Σnpriv_unfold:* "(${\Sigma_{priv}}^n$) = {f::'fun. arity f = n ∧ ¬public f}" ⟨*proof*⟩
lemma *Σfpub_unfold:* "Σ_{fpub} = {f::'fun. arity f > 0 ∧ public f}" ⟨*proof*⟩
lemma *Σfpriv_unfold:* "Σ_{fpriv} = {f::'fun. arity f > 0 ∧ ¬public f}" ⟨*proof*⟩
lemma *Σn_m_eq:* "⟦(Σ^n) ≠ {}; (Σ^n) = (Σ^m)⟧ ⟹ n = m" ⟨*proof*⟩

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"
⟨*proof*⟩

lemma *wf_trm_Var[simp]:* "wf_{trm} (Var x)" ⟨*proof*⟩

lemma *wf_trm_subst_range_Var[simp]:* "wf_{trms} (subst_range Var)" ⟨*proof*⟩

lemma *wf_trm_subst_range_iff:* "(∀x. wf_{trm} (ϑ x)) ⟷ wf_{trms} (subst_range ϑ)"
⟨*proof*⟩

lemma *wf_trm_subst_rangeD:* "wf_{trms} (subst_range ϑ) ⟹ wf_{trm} (ϑ x)"

⟨proof⟩

lemma *wf_trm_subst_rangeI[intro]:*
  "(⋀x. wf_{trm} (δ x)) ⟹ wf_{trms} (subst_range δ)"
⟨proof⟩

lemma *wf_trmI[intro]:*
  assumes "⋀t. t ∈ set T ⟹ wf_{trm} t" "length T = arity f"
  shows "wf_{trm} (Fun f T)"
⟨proof⟩

lemma *wf_trm_subterm:* "⟦wf_{trm} t; s ⊏ t⟧ ⟹ wf_{trm} s"
⟨proof⟩

lemma *wf_trm_subtermeq:*
  assumes "wf_{trm} t" "s ⊑ t"
  shows "wf_{trm} s"
⟨proof⟩

lemma *wf_trm_param:*
  assumes "wf_{trm} (Fun f T)" "t ∈ set T"
  shows "wf_{trm} t"
⟨proof⟩

lemma *wf_trm_param_idx:*
  assumes "wf_{trm} (Fun f T)"
    and "i < length T"
  shows "wf_{trm} (T ! i)"
⟨proof⟩

lemma *wf_trm_subst:*
  assumes "wf_{trms} (subst_range δ)"
  shows "wf_{trm} t = wf_{trm} (t · δ)"
⟨proof⟩

lemma *wf_trm_subst_singleton:*
  assumes "wf_{trm} t" "wf_{trm} t'" **shows** "wf_{trm} (t · Var(v := t'))"
⟨proof⟩

lemma *wf_trm_subst_rm_vars:*
  assumes "wf_{trm} (t · δ)"
  shows "wf_{trm} (t · rm_vars X δ)"
⟨proof⟩

lemma *wf_trm_subst_rm_vars':* "wf_{trm} (δ v) ⟹ wf_{trm} (rm_vars X δ v)"
⟨proof⟩

lemma *wf_trms_subst:*
  assumes "wf_{trms} (subst_range δ)" "wf_{trms} M"
  shows "wf_{trms} (M $\cdot_{set}$ δ)"
⟨proof⟩

lemma *wf_trms_subst_rm_vars:*
  assumes "wf_{trms} (M $\cdot_{set}$ δ)"
  shows "wf_{trms} (M $\cdot_{set}$ rm_vars X δ)"
⟨proof⟩

lemma *wf_trms_subst_rm_vars':*
  assumes "wf_{trms} (subst_range δ)"
  shows "wf_{trms} (subst_range (rm_vars X δ))"
⟨proof⟩

lemma *wf_trms_subst_compose:*

  assumes "wf$_{trms}$ (subst_range ϑ)" "wf$_{trms}$ (subst_range δ)"
  shows "wf$_{trms}$ (subst_range (ϑ ∘$_s$ δ))"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *wf_trms_Var_range:*
  assumes "subst_range δ ⊆ range Var"
  shows "wf$_{trms}$ (subst_range δ)"
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *wf_trm_subst_inv:* "wf$_{trm}$ (t · δ) ⟹ wf$_{trm}$ t"
⟨*proof*⟩

lemma *wf_trms_subst_inv:* "wf$_{trms}$ (M ·$_{set}$ δ) ⟹ wf$_{trms}$ M"
⟨*proof*⟩

lemma *wf_trm_subterms:* "wf$_{trm}$ t ⟹ wf$_{trms}$ (subterms t)"
⟨*proof*⟩

lemma *wf_trms_subterms:* "wf$_{trms}$ M ⟹ wf$_{trms}$ (subterms$_{set}$ M)"
⟨*proof*⟩

lemma *wf_trm_arity:* "wf$_{trm}$ (Fun f T) ⟹ length T = arity f"
⟨*proof*⟩

lemma *wf_trm_subterm_arity:* "wf$_{trm}$ t ⟹ Fun f T ⊑ t ⟹ length T = arity f"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *mgu_wf_trm:*
  assumes "mgu s t = Some σ" "wf$_{trm}$ s" "wf$_{trm}$ t"
  shows "wf$_{trm}$ (σ v)"
⟨*proof*⟩

lemma *mgu_wf_trms:*
  assumes "mgu s t = Some σ" "wf$_{trm}$ s" "wf$_{trm}$ t"
  shows "wf$_{trms}$ (subst_range σ)"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *intruder_synth_induct[consumes 1, case_names AxiomC ComposeC]:*
  fixes *M::"('fun,'var) terms"* **and** *t::"('fun,'var) term"*
  assumes *"M $\vdash_c$ t" "⋀t. t ∈ M ⟹ P M t"*
  "⋀T f. ⟦length T = arity f; public f;
  ⋀t. t ∈ set T ⟹ M $\vdash_c$ t;
  ⋀t. t ∈ set T ⟹ P M t⟧ ⟹ P M (Fun f T)"
  shows *"P M t"*
⟨*proof*⟩

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 $\vdash_c$ t"

definition *analyzed_in* **where**
  "analyzed_in t M ≡ ∀K R. (Ana t = (K,R) ∧ (∀k ∈ set K. M $\vdash_c$ k)) ⟶ (∀r ∈ set R. M $\vdash_c$ 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 ∈ $\mathcal{C}_{pub}$"*
  shows *"M ⊢ Fun c []" "M $\vdash_c$ Fun c []"*
⟨*proof*⟩

lemma *public_const_deduct'[simp]:*
  assumes *"arity c = 0" "public c"*
  shows *"M ⊢ Fun c []" "M $\vdash_c$ Fun c []"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *pgwt_public: "⟦public_ground_wf_term t; Fun f T ⊑ t⟧ ⟹ public f"*
⟨*proof*⟩

lemma *pgwt_ground: "public_ground_wf_term t ⟹ fv t = {}"*
⟨*proof*⟩

lemma *pgwt_fun: "public_ground_wf_term t ⟹ ∃f T. t = Fun f T"*
⟨*proof*⟩

lemma *pgwt_arity: "⟦public_ground_wf_term t; Fun f T ⊑ t⟧ ⟹ arity f = length T"*
⟨*proof*⟩

lemma *pgwt_wellformed: "public_ground_wf_term t ⟹ wf$_{trm}$ t"*
⟨*proof*⟩

lemma *pgwt_deducible: "public_ground_wf_term t ⟹ M ⊢$_c$ t"*
⟨*proof*⟩

lemma *pgwt_is_empty_synth: "public_ground_wf_term t ⟷ {} ⊢$_c$ t"*
⟨*proof*⟩

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*⟩

2.4.6 Lemmata: Monotonicity, deduction private constants, etc.

context
begin
lemma *ideduct_mono:*
  "⟦M ⊢ t; M ⊆ M'⟧ ⟹ M' ⊢ t"
⟨*proof*⟩

lemma *ideduct_synth_mono:*
  fixes *M::"('fun,'var) terms"* **and** *t::"('fun,'var) term"*
  shows *"⟦M ⊢$_c$ t; M ⊆ M'⟧ ⟹ M' ⊢$_c$ t"*
⟨*proof*⟩

lemma *ideduct_reduce:*
  "⟦M ∪ M' ⊢ t; ⋀t'. t' ∈ M' ⟹ M ⊢ t'⟧ ⟹ M ⊢ t"
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *ideduct_mono_eq:*

  assumes "$\forall t.$ M ⊢ t ⟷ M' ⊢ t" **shows** "M ∪ N ⊢ t ⟷ M' ∪ N ⊢ t"
⟨*proof*⟩

lemma *deduct_synth_subterm:*
  fixes M::"('fun,'var) terms" **and** t::"('fun,'var) term"
  assumes "M $\vdash_c$ t" "s ∈ subterms t" "∀m ∈ M. ∀s ∈ subterms m. M $\vdash_c$ s"
  shows "M $\vdash_c$ s"
⟨*proof*⟩

lemma *deduct_if_synth[intro, dest]:* "M $\vdash_c$ t ⟹ M ⊢ t"
⟨*proof*⟩ **lemma** *ideduct_ik_eq:* **assumes** "∀t ∈ M. M' ⊢ t" **shows** "M' ⊢ t ⟷ M' ∪ M ⊢ t"
⟨*proof*⟩ **lemma** *synth_if_deduct_empty:* "{} ⊢ t ⟹ {} $\vdash_c$ t"
⟨*proof*⟩ **lemma** *ideduct_deduct_synth_mono_eq:*
  assumes "∀t. M ⊢ t ⟷ M' $\vdash_c$ t" "M ⊆ M'"
  and "∀t. M' ∪ N ⊢ t ⟷ M' ∪ N ∪ D $\vdash_c$ t"
  shows "M ∪ N ⊢ t ⟷ M' ∪ N ∪ D $\vdash_c$ t"
⟨*proof*⟩

lemma *ideduct_subst:* "M ⊢ t ⟹ M $\cdot_{set}$ δ ⊢ t · δ"
⟨*proof*⟩

lemma *ideduct_synth_subst:*
  fixes M::"('fun,'var) terms" **and** t::"('fun,'var) term" **and** δ::"('fun,'var) subst"
  shows "M $\vdash_c$ t ⟹ M $\cdot_{set}$ δ $\vdash_c$ t · δ"
⟨*proof*⟩

lemma *ideduct_vars:*
  assumes "M ⊢ t"
  shows "fv t ⊆ fv_{set} M"
⟨*proof*⟩

lemma *ideduct_synth_vars:*
  fixes M::"('fun,'var) terms" **and** t::"('fun,'var) term"
  assumes "M $\vdash_c$ t"
  shows "fv t ⊆ fv_{set} M"
⟨*proof*⟩

lemma *ideduct_synth_priv_fun_in_ik:*
  fixes M::"('fun,'var) terms" **and** t::"('fun,'var) term"
  assumes "M $\vdash_c$ t" "f ∈ funs_term t" "¬public f"
  shows "f ∈ ⋃(funs_term ‘ M)"
⟨*proof*⟩

lemma *ideduct_synth_priv_const_in_ik:*
  fixes M::"('fun,'var) terms" **and** t::"('fun,'var) term"
  assumes "M $\vdash_c$ Fun c []" "¬public c"
  shows "Fun c [] ∈ M"
⟨*proof*⟩

lemma *ideduct_synth_ik_replace:*
  fixes M::"('fun,'var) terms" **and** t::"('fun,'var) term"
  assumes "∀t ∈ M. N $\vdash_c$ t"
    and "M $\vdash_c$ t"
  shows "N $\vdash_c$ t"
⟨*proof*⟩
end

2.4.7 Lemmata: Analyzed Intruder Knowledge Closure

lemma *deducts_eq_if_analyzed:* "analyzed M ⟹ M ⊢ t ⟷ M $\vdash_c$ t"
⟨*proof*⟩

lemma *closure_is_superset:* "decomp_closure M M' ⟹ M ⊆ M'"

⟨*proof*⟩

lemma *deduct_if_closure_deduct: "⟦M' ⊢ t; decomp_closure M M'⟧ ⟹ M ⊢ t"*
⟨*proof*⟩

lemma *deduct_if_closure_synth: "⟦decomp_closure M M'; M' $\vdash_c$ t⟧ ⟹ M ⊢ t"*
⟨*proof*⟩

lemma *decomp_closure_subterms_composable:*
  assumes *"decomp_closure M M'"*
  and *"M' $\vdash_c$ t'" "M' ⊢ t" "t ⊑ t'"*
  shows *"M' $\vdash_c$ t"*
⟨*proof*⟩

lemma *decomp_closure_analyzed:*
  assumes *"decomp_closure M M'"*
  shows *"analyzed M'"*
⟨*proof*⟩

lemma *analyzed_if_all_analyzed_in:*
  assumes *M: "∀t ∈ M. analyzed_in t M"*
  shows *"analyzed M"*
⟨*proof*⟩

lemma *analyzed_is_all_analyzed_in:*
  "(∀t ∈ M. analyzed_in t M) ⟷ analyzed M"
⟨*proof*⟩

lemma *ik_has_synth_ik_closure:*
  fixes *M :: "('fun,'var) terms"*
  shows *"∃M'. (∀t. M ⊢ t ⟷ M' $\vdash_c$ t) ∧ decomp_closure M M' ∧ (finite M ⟶ finite M')"*
⟨*proof*⟩

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"
  ("⟨_;_⟩ $\vdash_r$ _" 50)
where
  AxiomR[simp]:   "t ∈ M ⟹ ⟨M; Q⟩ $\vdash_r$ t"
| ComposeR[simp]: "⟦length T = arity f; public f; ⋀t. t ∈ set T ⟹ ⟨M; Q⟩ $\vdash_r$ t; Q (Fun f T)⟧
    ⟹ ⟨M; Q⟩ $\vdash_r$ Fun f T"
| DecomposeR: "⟦⟨M; Q⟩ $\vdash_r$ t; Ana t = (K, T); ⋀k. k ∈ set K ⟹ ⟨M; Q⟩ $\vdash_r$ k; t_i ∈ set T⟧
    ⟹ ⟨M; Q⟩ $\vdash_r$ t_i"

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

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

lemma *intruder_deduct_restricted_induct[consumes 1, case_names AxiomR ComposeR DecomposeR]:*

assumes "$\langle M;\ Q\rangle \vdash_r t$" "$\bigwedge t.\ t \in M \Longrightarrow P\ M\ Q\ t$"
"$\bigwedge T\ f.\ [\![length\ T = arity\ f;\ public\ f;$
$\bigwedge t.\ t \in set\ T \Longrightarrow \langle M;\ Q\rangle \vdash_r t;$
$\bigwedge t.\ t \in set\ T \Longrightarrow P\ M\ Q\ t;\ Q\ (Fun\ f\ T)$
$]\!] \Longrightarrow P\ M\ Q\ (Fun\ f\ T)$"
"$\bigwedge t\ K\ T\ t_i.\ [\![\langle M;\ Q\rangle \vdash_r t;\ P\ M\ Q\ t;\ Ana\ t = (K,\ T);\ \bigwedge k.\ k \in set\ K \Longrightarrow \langle M;\ Q\rangle \vdash_r k;$
$\bigwedge k.\ k \in set\ K \Longrightarrow P\ M\ Q\ k;\ t_i \in set\ T]\!] \Longrightarrow P\ M\ Q\ t_i$"
shows "$P\ M\ Q\ t$"
⟨proof⟩

lemma *intruder_deduct_num_induct[consumes 1, case_names AxiomN ComposeN DecomposeN]:*
assumes "$\langle M;\ n\rangle \vdash_n t$" "$\bigwedge t.\ t \in M \Longrightarrow P\ M\ 0\ t$"
"$\bigwedge T\ f\ steps.$
$[\![length\ T = arity\ f;\ public\ f;$
$\bigwedge t.\ t \in set\ T \Longrightarrow \langle M;\ steps\ t\rangle \vdash_n t;$
$\bigwedge t.\ t \in set\ T \Longrightarrow P\ M\ (steps\ t)\ t]\!]$
$\Longrightarrow P\ M\ (Suc\ (Max\ (insert\ 0\ (steps\ `\ set\ T))))\ (Fun\ f\ T)$"
"$\bigwedge t\ K\ T\ t_i\ steps\ n.$
$[\![\langle M;\ n\rangle \vdash_n t;\ P\ M\ n\ t;\ Ana\ t = (K,\ T);$
$\bigwedge k.\ k \in set\ K \Longrightarrow \langle M;\ steps\ k\rangle \vdash_n k;$
$t_i \in set\ T;\ \bigwedge k.\ k \in set\ K \Longrightarrow P\ M\ (steps\ k)\ k]\!]$
$\Longrightarrow P\ M\ (Suc\ (Max\ (insert\ n\ (steps\ `\ set\ K))))\ t_i$"
shows "$P\ M\ n\ t$"
⟨proof⟩

lemma *ideduct_restricted_mono:*
"$[\![\langle M;\ P\rangle \vdash_r t;\ M \subseteq M']\!] \Longrightarrow \langle M';\ P\rangle \vdash_r t$"
⟨proof⟩

2.4.9 Lemmata: Intruder Deduction Equivalences

lemma *deduct_if_restricted_deduct:* "$\langle M;P\rangle \vdash_r m \Longrightarrow M \vdash m$"
⟨proof⟩

lemma *restricted_deduct_if_restricted_ik:*
assumes "$\langle M;P\rangle \vdash_r m$" "$\forall m \in M.\ P\ m$"
and *P:* "$\forall t\ t'.\ P\ t \longrightarrow t' \sqsubseteq t \longrightarrow P\ t'$"
shows "$P\ m$"
⟨proof⟩

lemma *deduct_restricted_if_synth:*
assumes *P:* "$P\ m$" "$\forall t\ t'.\ P\ t \longrightarrow t' \sqsubseteq t \longrightarrow P\ t'$"
and *m:* "$M \vdash_c m$"
shows "$\langle M;\ P\rangle \vdash_r m$"
⟨proof⟩

lemma *deduct_zero_in_ik:*
assumes "$\langle M;\ 0\rangle \vdash_n t$" **shows** "$t \in M$"
⟨proof⟩

lemma *deduct_if_deduct_num:* "$\langle M;\ k\rangle \vdash_n t \Longrightarrow M \vdash t$"
⟨proof⟩

lemma *deduct_num_if_deduct:* "$M \vdash t \Longrightarrow \exists k.\ \langle M;\ k\rangle \vdash_n t$"
⟨proof⟩

lemma *deduct_normalize:*
assumes *M:* "$\forall m \in M.\ \forall f\ T.\ Fun\ f\ T \sqsubseteq m \longrightarrow P\ f\ T$"
and *t:* "$\langle M;\ k\rangle \vdash_n t$" "$Fun\ f\ T \sqsubseteq t$" "$\neg P\ f\ T$"
shows "$\exists l \le k.\ (\langle M;\ l\rangle \vdash_n Fun\ f\ T) \wedge (\forall t \in set\ T.\ \exists j < l.\ \langle M;\ j\rangle \vdash_n t)$"
⟨proof⟩

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")
⟨proof⟩

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⟩

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"
⟨proof⟩

lemma private_const_deduct:
  assumes c: "¬public c" "M ⊢ (Fun c []::('fun,'var) term)"
  shows "Fun c [] ∈ M ∨
        (∃m ∈ subterms_set M. M ⊢ m ∧ (∀k ∈ set (fst (Ana m)). M ⊢ m) ∧
                          Fun c [] ∈ set (snd (Ana m)))"
⟨proof⟩

lemma private_fun_deduct_in_ik'':
  assumes t: "M ⊢ Fun f T" "Fun c [] ∈ set T" "∀m ∈ subterms_set M. Fun f T ∉ set (snd (Ana m))"
    and c: "¬public c" "Fun c [] ∉ M" "∀m ∈ subterms_set M. Fun c [] ∉ set (snd (Ana m))"
  shows "Fun f T ∈ M"
⟨proof⟩

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"
⟨proof⟩
```

lemma (**in** *intruder_model*) *wf$_{trm}$_code[code_unfold]:*
  "wf$_{trm}$ t = wf$_{trm}$' arity t"
⟨*proof*⟩

lemma (**in** *intruder_model*) *wf$_{trms}$_code[code_unfold]:*
  "wf$_{trms}$ M = wf$_{trms}$' arity M"
⟨*proof*⟩

lemma (**in** *intruder_model*) *intruder_synth_code[code_unfold]:*
  "intruder_synth M t = intruder_synth' public arity M t"
  (**is** *"?A $\longleftrightarrow$?B"*)
⟨*proof*⟩

lemma (**in** *intruder_model*) *analyzed_in_code[code_unfold]:*
  "analyzed_in t M = analyzed_in' Ana public arity t M"
⟨*proof*⟩

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)"
⟨proof⟩
```

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))"
⟨*proof*⟩

lemma *measure$_{st}$_trans: "trans measure$_{st}$"*
⟨*proof*⟩

Some lemmata

lemma *trms_list$_{st}$_is_trms$_{st}$: "trms$_{st}$ S = set (trms_list$_{st}$ S)"*
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *subst_apply_strand_nil[simp]: "[] ·$_{st}$ δ = []"*
⟨*proof*⟩

lemma *finite_funs$_{stp}$[simp]: "finite (funs$_{stp}$ x)"* ⟨*proof*⟩
lemma *finite_funs$_{st}$[simp]: "finite (funs$_{st}$ S)"* ⟨*proof*⟩
lemma *finite_trms$_{pairs}$[simp]: "finite (trms$_{pairs}$ x)"* ⟨*proof*⟩
lemma *finite_trms$_{stp}$[simp]: "finite (trms$_{stp}$ x)"* ⟨*proof*⟩
lemma *finite_vars$_{stp}$[simp]: "finite (vars$_{stp}$ x)"* ⟨*proof*⟩
lemma *finite_bvars$_{stp}$[simp]: "finite (set (bvars$_{stp}$ x))"* ⟨*proof*⟩
lemma *finite_fv$_{snd}$[simp]: "finite (fv$_{snd}$ x)"* ⟨*proof*⟩
lemma *finite_fv$_{rcv}$[simp]: "finite (fv$_{rcv}$ x)"* ⟨*proof*⟩
lemma *finite_fv$_{stp}$[simp]: "finite (fv$_{stp}$ x)"* ⟨*proof*⟩
lemma *finite_vars$_{st}$[simp]: "finite (vars$_{st}$ S)"* ⟨*proof*⟩
lemma *finite_bvars$_{st}$[simp]: "finite (bvars$_{st}$ S)"* ⟨*proof*⟩
lemma *finite_fv$_{st}$[simp]: "finite (fv$_{st}$ S)"* ⟨*proof*⟩

lemma *finite_wfrestrictedvars$_{stp}$[simp]: "finite (wfrestrictedvars$_{stp}$ x)"*
⟨*proof*⟩

lemma *finite_wfrestrictedvars$_{st}$[simp]: "finite (wfrestrictedvars$_{st}$ S)"*
⟨*proof*⟩

lemma *finite_wfvarsoccs$_{stp}$[simp]: "finite (wfvarsoccs$_{stp}$ x)"*
⟨*proof*⟩

lemma *finite_wfvarsoccs$_{st}$[simp]: "finite (wfvarsoccs$_{st}$ S)"*
⟨*proof*⟩

lemma *finite_ik$_{st}$[simp]: "finite (ik$_{st}$ S)"*
⟨*proof*⟩

lemma *finite_assignment_rhs$_{st}$[simp]: "finite (assignment_rhs$_{st}$ S)"*
⟨*proof*⟩

lemma *ik$_{st}$_is_rcv_set: "ik$_{st}$ A = {t. Receive t ∈ set A}"*
⟨*proof*⟩

lemma *ik$_{st}$D[dest]: "t ∈ ik$_{st}$ S ⟹ Receive t ∈ set S"*
⟨*proof*⟩

lemma *ik$_{st}$D'[dest]: "t ∈ ik$_{st}$ S ⟹ t ∈ trms$_{st}$ S"*
⟨*proof*⟩

lemma *ik$_{st}$D''[dest]: "t ∈ subterms$_{set}$ (ik$_{st}$ S) ⟹ t ∈ subterms$_{set}$ (trms$_{st}$ S)"*
⟨*proof*⟩

lemma *ik$_{st}$_subterm_exD:*
  assumes *"t ∈ ik$_{st}$ S"*
  shows *"∃x ∈ set S. t ∈ subterms$_{set}$ (trms$_{stp}$ x)"*
⟨*proof*⟩

lemma *assignment_rhs$_{st}$D[dest]: "t ∈ assignment_rhs$_{st}$ S ⟹ ∃t'. Equality Assign t' t ∈ set S"*
⟨*proof*⟩

lemma *assignment_rhs$_{st}$D'[dest]: "t ∈ subterms$_{set}$ (assignment_rhs$_{st}$ S) ⟹ t ∈ subterms$_{set}$ (trms$_{st}$ S)"*
⟨*proof*⟩

lemma *bvars$_{st}$_split: "bvars$_{st}$ (S@S') = bvars$_{st}$ S ∪ bvars$_{st}$ S'"*
⟨*proof*⟩

lemma *bvars$_{st}$_singleton: "bvars$_{st}$ [x] = set (bvars$_{stp}$ x)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

lemma *strand_subst_hom[iff]:*
  "(S@S') $\cdot_{st}$ ϑ = (S $\cdot_{st}$ ϑ)@(S' $\cdot_{st}$ ϑ)" "(x#S) $\cdot_{st}$ ϑ = (x $\cdot_{stp}$ ϑ)#(S $\cdot_{st}$ ϑ)"
⟨*proof*⟩

lemma *strand_subst_comp: "range_vars δ ∩ bvars$_{st}$ S = {} ⟹ S $\cdot_{st}$ δ $\circ_s$ ϑ = ((S $\cdot_{st}$ δ) $\cdot_{st}$ ϑ)"*
⟨*proof*⟩

lemma *strand_substI[intro]:*
  "subst_domain ϑ ∩ fv$_{st}$ S = {} ⟹ S $\cdot_{st}$ ϑ = S"
  "subst_domain ϑ ∩ vars$_{st}$ S = {} ⟹ S $\cdot_{st}$ ϑ = S"
⟨*proof*⟩

lemma *strand_substI':*
  "fv$_{st}$ S = {} ⟹ S $\cdot_{st}$ ϑ = S"

"vars_{st} S = {} $\Longrightarrow$ S $\cdot_{st}$ ϑ = S"
⟨proof⟩

lemma *strand_subst_set:* "(set (S $\cdot_{st}$ ϑ)) = ((λx. x $\cdot_{stp}$ ϑ) ` (set S))"
⟨proof⟩

lemma *strand_map_inv_set_snd_rcv_subst:*
assumes "finite (M::('a,'b) terms)"
shows "set ((map Send (inv set M)) $\cdot_{st}$ ϑ) = Send ` (M $\cdot_{set}$ ϑ)" (**is** ?A)
"set ((map Receive (inv set M)) $\cdot_{st}$ ϑ) = Receive ` (M $\cdot_{set}$ ϑ)" (**is** ?B)
⟨proof⟩

lemma *strand_ground_subst_vars_subset:*
assumes "ground (subst_range ϑ)" **shows** "vars_{st} (S $\cdot_{st}$ ϑ) $\subseteq$ vars_{st} S"
⟨proof⟩

lemma *ik_union_subset:* "$\bigcup$(P ` $\text{ik}_{st}$ S) $\subseteq$ ($\bigcup$x $\in$ (set S). $\bigcup$(P ` trms_{stp} x))"
⟨proof⟩

lemma *ik_snd_empty[simp]:* "ik_{st} (map Send X) = {}"
⟨proof⟩

lemma *ik_snd_empty'[simp]:* "ik_{st} [Send t] = {}" ⟨proof⟩

lemma *ik_append[iff]:* "ik_{st} (S@S') = ik_{st} S $\cup$ ik_{st} S'" ⟨proof⟩

lemma *ik_cons:* "ik_{st} (x#S) = ik_{st} [x] $\cup$ ik_{st} S" ⟨proof⟩

lemma *assignment_rhs_append[iff]:* "$\text{assignment\_rhs}_{st}$ (S@S') = $\text{assignment\_rhs}_{st}$ S $\cup$ $\text{assignment\_rhs}_{st}$ S'"
⟨proof⟩

lemma *eqs_rcv_map_empty:* "$\text{assignment\_rhs}_{st}$ (map Receive M) = {}"
⟨proof⟩

lemma *ik_rcv_map:* **assumes** "t $\in$ set L" **shows** "t $\in$ ik_{st} (map Receive L)"
⟨proof⟩

lemma *ik_subst:* "ik_{st} (S $\cdot_{st}$ δ) = ik_{st} S $\cdot_{set}$ δ"
⟨proof⟩

lemma *ik_rcv_map':* **assumes** "t $\in$ ik_{st} (map Receive L)" **shows** "t $\in$ set L"
⟨proof⟩

lemma *ik_append_subset[simp]:* "ik_{st} S $\subseteq$ ik_{st} (S@S')" "ik_{st} S' $\subseteq$ ik_{st} (S@S')"
⟨proof⟩

lemma *assignment_rhs_append_subset[simp]:*
"$\text{assignment\_rhs}_{st}$ S $\subseteq$ $\text{assignment\_rhs}_{st}$ (S@S')"
"$\text{assignment\_rhs}_{st}$ S' $\subseteq$ $\text{assignment\_rhs}_{st}$ (S@S')"
⟨proof⟩

lemma *trms_{st}_cons:* "trms_{st} (x#S) = trms_{stp} x $\cup$ trms_{st} S" ⟨proof⟩

lemma *trm_strand_subst_cong:*
"t $\in$ trms_{st} S $\Longrightarrow$ t $\cdot$ δ $\in$ trms_{st} (S $\cdot_{st}$ δ)
$\vee$ ($\exists$X F. Inequality X F $\in$ set S $\wedge$ t $\cdot$ rm_vars (set X) δ $\in$ trms_{st} (S $\cdot_{st}$ δ))"
(**is** "t $\in$ trms_{st} S $\Longrightarrow$?P t δ S")
"t $\in$ trms_{st} (S $\cdot_{st}$ δ) $\Longrightarrow$ ($\exists$t'. t = t' $\cdot$ δ $\wedge$ t' $\in$ trms_{st} S)
$\vee$ ($\exists$X F. Inequality X F $\in$ set S $\wedge$ ($\exists$t' $\in$ trms_{pairs} F. t = t' $\cdot$ rm_vars (set X) δ))"
(**is** "t $\in$ trms_{st} (S $\cdot_{st}$ δ) $\Longrightarrow$?Q t δ S")
⟨proof⟩

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"
⟨proof⟩

lemma in_strand_fv_subset: "x ∈ set S ⟹ $vars_{stp}$ x ⊆ $vars_{st}$ S" ⟨proof⟩
lemma in_strand_fv_subset_snd: "Send t ∈ set S ⟹ fv t ⊆ ⋃(set (map fv_{snd} S))" ⟨proof⟩
lemma in_strand_fv_subset_rcv: "Receive t ∈ set S ⟹ fv t ⊆ ⋃(set (map fv_{rcv} S))" ⟨proof⟩

lemma fv_{snd}E:
  assumes "v ∈ ⋃(set (map fv_{snd} S))"
  obtains t **where** "send⟨t⟩$_{st}$ ∈ set S" "v ∈ fv t"
⟨proof⟩

lemma fv_{rcv}E:
  assumes "v ∈ ⋃(set (map fv_{rcv} S))"
  obtains t **where** "receive⟨t⟩$_{st}$ ∈ set S" "v ∈ fv t"
⟨proof⟩

lemma $vars_{stp}$I[intro]: "x ∈ fv_{stp} s ⟹ x ∈ $vars_{stp}$ s"
⟨proof⟩

lemma $vars_{st}$I[intro]: "x ∈ fv_{st} S ⟹ x ∈ $vars_{st}$ S" ⟨proof⟩

lemma fv_{st}_subset_$vars_{st}$[simp]: "fv_{st} S ⊆ $vars_{st}$ S" ⟨proof⟩

lemma $vars_{st}$_is_fv_{st}_$bvars_{st}$: "$vars_{st}$ S = fv_{st} S ∪ $bvars_{st}$ S"
⟨proof⟩

lemma fv_{stp}_is_subterm_$trms_{stp}$: "x ∈ fv_{stp} a ⟹ Var x ∈ $subterms_{set}$ ($trms_{stp}$ a)"
⟨proof⟩

lemma fv_{st}_is_subterm_$trms_{st}$: "x ∈ fv_{st} A ⟹ Var x ∈ $subterms_{set}$ ($trms_{st}$ A)"
⟨proof⟩

lemma vars_st_snd_map: "$vars_{st}$ (map Send X) = fv (Fun f X)" ⟨proof⟩

lemma vars_st_rcv_map: "$vars_{st}$ (map Receive X) = fv (Fun f X)" ⟨proof⟩

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⟩

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⟩

lemma fv_snd_rcv_empty[simp]: "fv_{snd} x = {} ∨ fv_{rcv} x = {}" ⟨proof⟩

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"
⟨proof⟩

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)))"
⟨proof⟩

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)))"
⟨proof⟩

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⟩

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"
⟨proof⟩

lemma *$wfrestrictedvars_{st}$_subset_$vars_{st}$:* "$wfrestrictedvars_{st}$ S ⊆ $vars_{st}$ S"
⟨proof⟩

lemma *subst_sends_strand_step_fv_to_img:* "fv_{stp} (x $\cdot_{stp}$ δ) ⊆ fv_{stp} x ∪ range_vars δ"
⟨proof⟩

lemma *subst_sends_strand_fv_to_img:* "fv_{st} (S $\cdot_{st}$ δ) ⊆ fv_{st} S ∪ range_vars δ"
⟨proof⟩

lemma *ineq_apply_subst:*
assumes "subst_domain δ ∩ set X = {}"
shows "(Inequality X F) $\cdot_{stp}$ δ = Inequality X (F $\cdot_{pairs}$ δ)"
⟨proof⟩

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 $\cdot_{stp}$ δ)"
⟨proof⟩

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 $\cdot_{st}$ δ)))"
⟨proof⟩

lemma *fv_strand_subst2:*
assumes "$bvars_{st}$ S ∩ (subst_domain δ ∪ range_vars δ) = {}"
shows "fv_{set} (δ ‘ ($wfrestrictedvars_{st}$ S)) = $wfrestrictedvars_{st}$ (S $\cdot_{st}$ δ)"
⟨proof⟩

lemma *fv_strand_subst':*
assumes "$bvars_{st}$ S ∩ (subst_domain δ ∪ range_vars δ) = {}"
shows "fv_{set} (δ ‘ (fv_{st} S)) = fv_{st} (S $\cdot_{st}$ δ)"
⟨proof⟩

lemma *fv_$trms_{pairs}$_is_fv_{pairs}:*
"fv_{set} ($trms_{pairs}$ F) = fv_{pairs} F"
⟨proof⟩

lemma *fv_{pairs}_in_fv_$trms_{pairs}$:* "x ∈ fv_{pairs} F ⟹ x ∈ fv_{set} ($trms_{pairs}$ F)"
⟨proof⟩

lemma *$trms_{st}$_append:* "$trms_{st}$ (A@B) = $trms_{st}$ A ∪ $trms_{st}$ B"
⟨proof⟩

lemma *$trms_{pairs}$_subst:* "$trms_{pairs}$ (a $\cdot_{pairs}$ ϑ) = $trms_{pairs}$ a $\cdot_{set}$ ϑ"
⟨proof⟩

lemma *trms$_{pairs}$_fv_subst_subset:*
  "t ∈ trms$_{pairs}$ F ⟹ fv (t · ϑ) ⊆ fv$_{pairs}$ (F ·$_{pairs}$ ϑ)"
⟨proof⟩

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⟩

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))"
⟨proof⟩

lemma *trm$_{stp}$_subst:*
  assumes "subst_domain ϑ ∩ set (bvars$_{stp}$ a) = {}"
  shows "trms$_{stp}$ (a ·$_{stp}$ ϑ) = trms$_{stp}$ a ·$_{set}$ ϑ"
⟨proof⟩

lemma *trms$_{st}$_subst:*
  assumes "subst_domain ϑ ∩ bvars$_{st}$ A = {}"
  shows "trms$_{st}$ (A ·$_{st}$ ϑ) = trms$_{st}$ A ·$_{set}$ ϑ"
⟨proof⟩

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"
⟨proof⟩

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)"
⟨proof⟩

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)"
⟨proof⟩

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⟩

lemma *subst_apply_fv_subset_strand2:*
  assumes P: "P = fv$_{stp}$ ∨ P = fv$_{rcv}$ ∨ P = fv$_{snd}$ ∨ P = fv$_{eq}$ ac ∨ P = fv$_{ineq}$ ∨ P = fv_r$_{eq}$ ac"
  and P_subset: "P x ⊆ wfrestrictedvars$_{st}$ 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) ⊆ wfrestrictedvars$_{st}$ (S ·$_{st}$ δ) ∪ fv$_{set}$ (δ ‘ V)"
⟨proof⟩

lemma *strand_subst_fv_bounded_if_img_bounded:*
  assumes "range_vars δ ⊆ fv$_{st}$ S"
  shows "fv$_{st}$ (S ·$_{st}$ δ) ⊆ fv$_{st}$ S"
⟨proof⟩

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 $\cdot_{st}$ δ)"
⟨proof⟩

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 $\cdot_{st}$ δ) ⊂ fv_{st} S"
⟨proof⟩

lemma fv_ik_is_fv_rcv: "fv_{set} (ik_{st} S) = ⋃(set (map fv_{rcv} S))"
⟨proof⟩

lemma fv_ik_subset_fv_st[simp]: "fv_{set} (ik_{st} S) ⊆ $wfrestrictedvars_{st}$ S"
⟨proof⟩

lemma fv_assignment_rhs_subset_fv_st[simp]: "fv_{set} ($assignment\_rhs_{st}$ S) ⊆ $wfrestrictedvars_{st}$ S"
⟨proof⟩

lemma fv_ik_subset_fv_st'[simp]: "fv_{set} (ik_{st} S) ⊆ fv_{st} S"
⟨proof⟩

lemma ik_{st}_var_is_fv: "Var x ∈ $subterms_{set}$ (ik_{st} A) ⟹ x ∈ fv_{st} A"
⟨proof⟩

lemma fv_assignment_rhs_subset_fv_st'[simp]: "fv_{set} ($assignment\_rhs_{st}$ S) ⊆ fv_{st} S"
⟨proof⟩

lemma ik_{st}_$assignment_rhs_{st}$_wfrestrictedvars_subset:
  "fv_{set} (ik_{st} A ∪ $assignment\_rhs_{st}$ A) ⊆ $wfrestrictedvars_{st}$ A"
⟨proof⟩

lemma strand_step_id_subst[iff]: "x $\cdot_{stp}$ Var = x" ⟨proof⟩

lemma strand_id_subst[iff]: "S $\cdot_{st}$ Var = S" ⟨proof⟩

lemma strand_subst_vars_union_bound[simp]: "$vars_{st}$ (S $\cdot_{st}$ δ) ⊆ $vars_{st}$ S ∪ range_vars δ"
⟨proof⟩

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'"
⟨proof⟩

lemma bvars_subst_ident: "$bvars_{st}$ S = $bvars_{st}$ (S $\cdot_{st}$ δ)"
⟨proof⟩

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 $\cdot_{st}$ δ) $\cdot_{st}$ ϑ = (S $\cdot_{st}$ δ)"
  and "(S $\cdot_{st}$ δ) $\cdot_{st}$ (ϑ $\circ_s$ δ) = (S $\cdot_{st}$ δ)"
⟨proof⟩

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 $\cdot_{st}$ δ)"
⟨proof⟩

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 $\cdot_{st}$ δ)"
⟨proof⟩

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 $\cdot_{st}$ δ)"
⟨proof⟩

lemma strand_subst_not_dom_fixed:
assumes "v ∈ fv_{st} S" **and** "v ∉ subst_domain δ"
shows "v ∈ fv_{st} (S $\cdot_{st}$ δ)"
⟨proof⟩

lemma strand_vars_unfold: "v ∈ $vars_{st}$ S ⟹ ∃S' x S''. S = S'@x#S'' ∧ v ∈ $vars_{stp}$ x"
⟨proof⟩

lemma strand_fv_unfold: "v ∈ fv_{st} S ⟹ ∃S' x S''. S = S'@x#S'' ∧ v ∈ fv_{stp} x"
⟨proof⟩

lemma subterm_if_in_strand_ik:
"t ∈ ik_{st} S ⟹ ∃t'. Receive t' ∈ set S ∧ t ⊑ t'"
⟨proof⟩

lemma fv_subset_if_in_strand_ik:
"t ∈ ik_{st} S ⟹ fv t ⊆ ⋃(set (map fv_{rcv} S))"
⟨proof⟩

lemma fv_subset_if_in_strand_ik':
"t ∈ ik_{st} S ⟹ fv t ⊆ fv_{st} S"
⟨proof⟩

lemma vars_subset_if_in_strand_ik2:
"t ∈ ik_{st} S ⟹ fv t ⊆ $wfrestrictedvars_{st}$ S"
⟨proof⟩

3.1.3 Lemmata: Simple Strands

lemma simple_Cons[dest]: "simple (s#S) ⟹ simple S"
⟨proof⟩

lemma simple_split[dest]:
assumes "simple (S@S')"
shows "simple S" "simple S'"
⟨proof⟩

lemma simple_append[intro]: "⟦simple S; simple S'⟧ ⟹ simple (S@S')"
⟨proof⟩

lemma simple_append_sym[sym]: "simple (S@S') ⟹ simple (S'@S)" ⟨proof⟩

lemma not_simple_if_snd_fun: "(∃S' S'' f X. S = S'@Send (Fun f X)#S'') ⟹ ¬simple S"
⟨proof⟩

lemma not_list_all_elim: "¬list_all P A ⟹ ∃B x C. A = B@x#C ∧ ¬P x ∧ list_all P B"
⟨proof⟩

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))"

⟨proof⟩

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 ∧ ¬(∃$\mathcal{I}$. ineq_model $\mathcal{I}$ X F))"
⟨proof⟩

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⟩

lemma *simple_snd_is_var: "⟦Send t ∈ set S; simple S⟧ ⟹ ∃v. t = Var v"*
⟨proof⟩

3.1.4 Lemmata: Strand Measure

lemma *measure$_{st}$_wellfounded: "wf measure$_{st}$"* ⟨proof⟩

lemma *strand_size_append[iff]: "size$_{st}$ (S@S') = size$_{st}$ S + size$_{st}$ S'"*
⟨proof⟩

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)]"
⟨proof⟩

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')"
⟨proof⟩

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⟩

lemma *strand_fv_card_rm_fun_le[simp]: "card (fv$_{st}$ (S@S')) ≤ card (fv$_{st}$ (S@Send (Fun f X)#S'))"*
⟨proof⟩

lemma *strand_fv_card_rm_eq_le[simp]: "card (fv$_{st}$ (S@S')) ≤ card (fv$_{st}$ (S@Equality a t t'#S'))"*
⟨proof⟩

3.1.5 Lemmata: Well-formed Strands

lemma *wf_prefix[dest]: "wf$_{st}$ V (S@S') ⟹ wf$_{st}$ V S"*
⟨proof⟩

lemma *wf_vars_mono[simp]: "wf$_{st}$ V S ⟹ wf$_{st}$ (V ∪ W) S"*
⟨proof⟩

lemma *wf$_{st}$I[intro]: "wfrestrictedvars$_{st}$ S ⊆ V ⟹ wf$_{st}$ V S"*
⟨proof⟩

lemma *wf$_{st}$I'[intro]: "⋃(fv$_{rcv}$ ` set S) ∪ ⋃(fv_r$_{eq}$ assign ` set S) ⊆ V ⟹ wf$_{st}$ V S"*
⟨proof⟩

lemma *wf_append_exec: "wf$_{st}$ V (S@S') ⟹ wf$_{st}$ (V ∪ wfvarsoccs$_{st}$ S) S'"*
⟨proof⟩

lemma *wf_append_suffix:*

"wf_{st} V S ⟹ $wfrestrictedvars_{st}$ S' ⊆ $wfrestrictedvars_{st}$ S ∪ V ⟹ wf_{st} V (S@S')"
⟨proof⟩

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')"
⟨proof⟩

lemma *wf_send_compose:* "wf_{st} V (S@(map Send X)@S') = wf_{st} V (S@Send (Fun f X)#S')"
⟨proof⟩

lemma *wf_snd_append[iff]:* "wf_{st} V (S@[Send t]) = wf_{st} V S"
⟨proof⟩

lemma *wf_snd_append':* "wf_{st} V S ⟹ wf_{st} V (Send t#S)"
⟨proof⟩

lemma *wf_rcv_append[dest]:* "wf_{st} V (S@Receive t#S') ⟹ wf_{st} V (S@S')"
⟨proof⟩

lemma *wf_rcv_append'[intro]:*
"⟦wf_{st} V (S@S'); fv t ⊆ $wfrestrictedvars_{st}$ S ∪ V⟧ ⟹ wf_{st} V (S@Receive t#S')"
⟨proof⟩

lemma *wf_rcv_append''[intro]:* "⟦wf_{st} V S; fv t ⊆ ⋃(set (map fv$_{snd}$ S))⟧ ⟹ wf_{st} V (S@[Receive t])"
⟨proof⟩

lemma *wf_rcv_append'''[intro]:* "⟦wf_{st} V S; fv t ⊆ $wfrestrictedvars_{st}$ S ∪ V⟧ ⟹ wf_{st} V (S@[Receive t])"
⟨proof⟩

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⟩

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⟩

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⟩

lemma *wf_eq_append'''[intro]:*
"⟦wf_{st} V S; fv t' ⊆ $wfrestrictedvars_{st}$ S ∪ V⟧ ⟹ wf_{st} V (S@[Equality a t t'])"
⟨proof⟩

lemma *wf_eq_check_append[dest]:* "wf_{st} V (S@Equality Check t t'#S') ⟹ wf_{st} V (S@S')"
⟨proof⟩

lemma *wf_eq_check_append'[intro]:* "wf_{st} V (S@S') ⟹ wf_{st} V (S@Equality Check t t'#S')"
⟨proof⟩

lemma *wf_eq_check_append''[intro]:* "wf_{st} V S ⟹ wf_{st} V (S@[Equality Check t t'])"
⟨proof⟩

lemma *wf_ineq_append[dest]:* "wf_{st} V (S@Inequality X F#S') ⟹ wf_{st} V (S@S')"
⟨proof⟩

lemma *wf_ineq_append'[intro]:* "wf_{st} V (S@S') ⟹ wf_{st} V (S@Inequality X F#S')"
⟨proof⟩

lemma *wf_ineq_append''[intro]:* "wf_{st} V S ⟹ wf_{st} V (S@[Inequality X F])"
⟨*proof*⟩

lemma *wf_rcv_fv_single[elim]:* "wf_{st} V (Receive t#S') ⟹ fv t ⊆ V"
⟨*proof*⟩

lemma *wf_rcv_fv:* "wf_{st} V (S@Receive t#S') ⟹ fv t ⊆ $wfvarsoccs_{st}$ S ∪ V"
⟨*proof*⟩

lemma *wf_eq_fv:* "wf_{st} V (S@Equality Assign t t'#S') ⟹ fv t' ⊆ $wfvarsoccs_{st}$ S ∪ V"
⟨*proof*⟩

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}"
⟨*proof*⟩

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 $\cdot_{st}$ δ)))"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *wf_subst_apply:*
  "wf_{st} V S ⟹ wf_{st} (fv_{set} (δ ‘ V)) (S $\cdot_{st}$ δ)"
⟨*proof*⟩

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') $\cdot_{st}$ δ)"
⟨*proof*⟩

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') $\cdot_{st}$ δ)"
⟨*proof*⟩

lemma *wf_rcv_prefix_ground:*
  "wf_{st} {} ((map Receive M)@S) ⟹ $vars_{st}$ (map Receive M) = {}"
⟨*proof*⟩

lemma *simple_$wfvarsoccs_{st}$_is_fv_{snd}:*
  assumes "simple S"
  shows "$wfvarsoccs_{st}$ S = ⋃(set (map fv_{snd} S))"
⟨*proof*⟩

```
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"
⟨proof⟩

lemma wf_trm_stp_dom_fv_disjoint:
  "⟦wf_constr S ϑ; t ∈ trms_st S⟧ ⟹ subst_domain ϑ ∩ fv t = {}"
⟨proof⟩

lemma wf_constr_bvars_disj: "wf_constr S ϑ ⟹ (subst_domain ϑ ∪ range_vars ϑ) ∩ bvars_st S = {}"
⟨proof⟩

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⟩

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")
⟨proof⟩

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")
⟨proof⟩
```

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⟩

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⟩

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⟩ **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⟩

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⟩

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"
⟨proof⟩

lemma strand_sem_Receive_map: "⟦M; map Receive T⟧_c ℐ" "⟦M; map Receive T⟧_d ℐ"
⟨proof⟩

lemma strand_sem_append[intro]:
  "⟦⟦M; S⟧_c ϑ; ⟦M ∪ (ik_st S ·_set ϑ); S'⟧_c ϑ⟧ ⟹ ⟦M; S@S'⟧_c ϑ"
  "⟦⟦M; S⟧_d ϑ; ⟦M ∪ (ik_st S ·_set ϑ); S'⟧_d ϑ⟧ ⟹ ⟦M; S@S'⟧_d ϑ"
⟨proof⟩

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⟩

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 $\cdot_{pairs}$ δ)"*
shows *"ineq_model (δ $\circ_s$ ϑ) X F"*
⟨*proof*⟩

lemma *ineq_model_ground_subst:*
fixes *F::"(('a,'b) term × ('a,'b) term) list"*
assumes *"fv_{pairs} F - set X ⊆ subst_domain δ"*
and *"ground (subst_range δ)"*
and *"ineq_model δ X F"*
shows *"ineq_model (δ $\circ_s$ ϑ) X F"*
⟨*proof*⟩

context
begin
private lemma *strand_sem_subst_c:*
assumes *"(subst_domain δ ∪ range_vars δ) ∩ $bvars_{st}$ S = {}"*
shows *"⟦M; S⟧$_c$ (δ $\circ_s$ ϑ) ⟹ ⟦M; S $\cdot_{st}$ δ⟧$_c$ ϑ"*
⟨*proof*⟩ **lemma** *strand_sem_subst_c':*
assumes *"(subst_domain δ ∪ range_vars δ) ∩ $bvars_{st}$ S = {}"*
shows *"⟦M; S $\cdot_{st}$ δ⟧$_c$ ϑ ⟹ ⟦M; S⟧$_c$ (δ $\circ_s$ ϑ)"*
⟨*proof*⟩ **lemma** *strand_sem_subst_d:*
assumes *"(subst_domain δ ∪ range_vars δ) ∩ $bvars_{st}$ S = {}"*
shows *"⟦M; S⟧$_d$ (δ $\circ_s$ ϑ) ⟹ ⟦M; S $\cdot_{st}$ δ⟧$_d$ ϑ"*
⟨*proof*⟩ **lemma** *strand_sem_subst_d':*
assumes *"(subst_domain δ ∪ range_vars δ) ∩ $bvars_{st}$ S = {}"*
shows *"⟦M; S $\cdot_{st}$ δ⟧$_d$ ϑ ⟹ ⟦M; S⟧$_d$ (δ $\circ_s$ ϑ)"*
⟨*proof*⟩

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 $\cdot_{st}$ δ⟧$_c$ (δ $\circ_s$ ϑ); subst_idem δ⟧ ⟹ ⟦M; S⟧$_c$ (δ $\circ_s$ ϑ)"*
⟨*proof*⟩

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 $\circ_s$ δ)"*
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *strand_sem_c_imp_ineq_model:*
assumes *"⟦M; S⟧$_c$ ℐ" "Inequality X F ∈ set S"*
shows *"ineq_model ℐ X F"*
⟨*proof*⟩

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 $\cdot_{set}$ ℐ $\vdash_c$ ℐ v"*
⟨*proof*⟩

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"*

```
      "tᵢ ∈ set T" "¬ik_st A ·_set ℐ ⊢_c tᵢ" "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⟩
```

```
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"
⟨proof⟩
```

```
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')]"
⟨proof⟩
```

```
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"
⟨proof⟩
```

```
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⟩
```

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⟩
```

```
lemma strand_sem_split'[dest]:
  "⟦M; S@S'⟧_c' ϑ ⟹ ⟦M; S⟧_c' ϑ"
  "⟦M; S@S'⟧_c' ϑ ⟹ ⟦M ∪ ik_st S; S'⟧_c' ϑ"
  "⟦M; S@S'⟧_d' ϑ ⟹ ⟦M; S⟧_d' ϑ"
  "⟦M; S@S'⟧_d' ϑ ⟹ ⟦M ∪ ik_st S; S'⟧_d' ϑ"
⟨proof⟩
```

```
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"
⟨proof⟩

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)"
⟨proof⟩

lemma dualst_self_inverse: "dualst (dualst S) = S"
⟨proof⟩

lemma dualst_trms_eq: "trmsst (dualst S) = trmsst S"
⟨proof⟩

lemma dualst_fv: "fvst (dualst A) = fvst A"
⟨proof⟩

lemma dualst_bvars: "bvarsst (dualst A) = bvarsst A"
⟨proof⟩


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' ∈ ikst 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', ϑ_1), (S@Send (Fun f T)#S', ϑ_2)) $\in$ measure$_{st}$"
⟨*proof*⟩ **lemma** *LI_unify_measure_lt:*
  assumes "Some δ = mgu (Fun f T) t" "fv t $\subseteq$ fv$_{st}$ S"
  shows "(((S@S') $\cdot_{st}$ δ, ϑ_1), (S@Send (Fun f T)#S', ϑ_2)) $\in$ measure$_{st}$"
⟨*proof*⟩ **lemma** *LI_equality_measure_lt:*
  assumes "Some δ = mgu t t'"
  shows "(((S@S') $\cdot_{st}$ δ, ϑ_1), (S@Equality a t t'#S', ϑ_2)) $\in$ measure$_{st}$"
⟨*proof*⟩ **lemma** *LI_in_measure:* "(S_1, ϑ_1) $\rightsquigarrow$ (S_2, ϑ_2) $\Longrightarrow$ ((S_2, ϑ_2), (S_1, ϑ_1)) $\in$ measure$_{st}$"
⟨*proof*⟩ **lemma** *LI_in_measure_trans:* "(S_1, ϑ_1) $\rightsquigarrow^+$ (S_2, ϑ_2) $\Longrightarrow$ ((S_2, ϑ_2), (S_1, ϑ_1)) $\in$ measure$_{st}$"
⟨*proof*⟩ **lemma** *LI_converse_wellfounded_trans:* "wf ((LI_rel$^+$)$^{-1}$)"
⟨*proof*⟩ **lemma** *LI_acyclic_trans:* "acyclic (LI_rel$^+$)"
⟨*proof*⟩ **lemma** *LI_acyclic:* "acyclic LI_rel"
⟨*proof*⟩

lemma *LI_no_infinite_chain:* "$\neg$($\exists$f. $\forall$i. f i $\rightsquigarrow^+$ f (Suc i))"
⟨*proof*⟩ **lemma** *LI_unify_finite:*
  assumes "finite M"
  shows "finite {((S@Send (Fun f T)#S', ϑ), ((S@S') $\cdot_{st}$ δ, $\vartheta \circ_s \delta$)) | δ T'.
      simple S $\wedge$ Fun f T' $\in$ M $\wedge$ Some δ = mgu (Fun f T) (Fun f T')}"
⟨*proof*⟩
end

3.2.3 Lemma: The Lazy Intruder Preserves Well-formedness

context
begin
private lemma *LI_preserves_subst_wf_single:*
  assumes "(S_1, ϑ_1) $\rightsquigarrow$ (S_2, ϑ_2)" "fv$_{st}$ S_1 $\cap$ bvars$_{st}$ S_1 = {}" "wf$_{subst}$ ϑ_1"
  and "subst_domain ϑ_1 $\cap$ vars$_{st}$ S_1 = {}" "range_vars ϑ_1 $\cap$ bvars$_{st}$ S_1 = {}"
  shows "fv$_{st}$ S_2 $\cap$ bvars$_{st}$ S_2 = {}" "wf$_{subst}$ ϑ_2"
  and "subst_domain ϑ_2 $\cap$ vars$_{st}$ S_2 = {}" "range_vars ϑ_2 $\cap$ bvars$_{st}$ S_2 = {}"
⟨*proof*⟩ **lemma** *LI_preserves_subst_wf:*
  assumes "(S_1, ϑ_1) $\rightsquigarrow^*$ (S_2, ϑ_2)" "fv$_{st}$ S_1 $\cap$ bvars$_{st}$ S_1 = {}" "wf$_{subst}$ ϑ_1"
  and "subst_domain ϑ_1 $\cap$ vars$_{st}$ S_1 = {}" "range_vars ϑ_1 $\cap$ bvars$_{st}$ S_1 = {}"
  shows "fv$_{st}$ S_2 $\cap$ bvars$_{st}$ S_2 = {}" "wf$_{subst}$ ϑ_2"
  and "subst_domain ϑ_2 $\cap$ vars$_{st}$ S_2 = {}" "range_vars ϑ_2 $\cap$ bvars$_{st}$ S_2 = {}"
⟨*proof*⟩

lemma *LI_preserves_wellformedness:*
  assumes "(S_1, ϑ_1) $\rightsquigarrow^*$ (S_2, ϑ_2)" "wf$_{constr}$ S_1 ϑ_1"
  shows "wf$_{constr}$ S_2 ϑ_2"
⟨*proof*⟩

lemma *LI_preserves_trm_wf:*
  assumes "(S, ϑ) $\rightsquigarrow^*$ (S', ϑ')" "wf$_{trms}$ (trms$_{st}$ S)"
  shows "wf$_{trms}$ (trms$_{st}$ S')"
⟨*proof*⟩
end

3.2.4 Theorem: Soundness of the Lazy Intruder

context
begin
private lemma *LI_soundness_single:*
  assumes "wf$_{constr}$ S_1 ϑ_1" "(S_1, ϑ_1) $\rightsquigarrow$ (S_2, ϑ_2)" "$\mathcal{I} \models_c \langle S_2, \vartheta_2 \rangle$"
  shows "$\mathcal{I} \models_c \langle S_1, \vartheta_1 \rangle$"
⟨*proof*⟩

theorem *LI_soundness:*
  assumes "wf_{constr} S_1 ϑ_1" "(S_1,ϑ_1) $\leadsto^*$ (S_2,ϑ_2)" "$\mathcal{I}$ $\models_c$ $\langle S_2,\ \vartheta_2\rangle$"
  shows "$\mathcal{I}$ $\models_c$ $\langle S_1,\ \vartheta_1\rangle$"
⟨*proof*⟩
end

3.2.5 Theorem: Completeness of the Lazy Intruder

context
begin
private lemma *LI_completeness_single:*
  assumes "wf_{constr} S_1 ϑ_1" "$\mathcal{I}$ $\models_c$ $\langle S_1,\ \vartheta_1\rangle$" "¬simple S_1"
  shows "∃S_2 ϑ_2. (S_1,ϑ_1) $\leadsto$ (S_2,ϑ_2) ∧ ($\mathcal{I}$ $\models_c$ $\langle S_2,\ \vartheta_2\rangle$)"
⟨*proof*⟩

theorem *LI_completeness:*
  assumes "wf_{constr} S_1 ϑ_1" "$\mathcal{I}$ $\models_c$ $\langle S_1,\ \vartheta_1\rangle$"
  shows "∃S_2 ϑ_2. (S_1,ϑ_1) $\leadsto^*$ (S_2,ϑ_2) ∧ simple S_2 ∧ ($\mathcal{I}$ $\models_c$ $\langle S_2,\ \vartheta_2\rangle$)"
⟨*proof*⟩
end

3.2.6 Corollary: Soundness and Completeness as a Single Theorem

corollary *LI_soundness_and_completeness:*
  assumes "wf_{constr} S_1 ϑ_1"
  shows "$\mathcal{I}$ $\models_c$ $\langle S_1,\ \vartheta_1\rangle$ ⟷ (∃S_2 ϑ_2. (S_1,ϑ_1) $\leadsto^*$ (S_2,ϑ_2) ∧ simple S_2 ∧ ($\mathcal{I}$ $\models_c$ $\langle S_2,\ \vartheta_2\rangle$))"
⟨*proof*⟩

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 "𝔗a ≡ UNIV::('atom set)"
```

Well-typed substitutions

```
definition wtsubst where
  "wtsubst 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; wtsubst δ; wftrms (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 tfrset where
  "tfrset 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 tfrstp where
  "tfrstp (Equality a t t') = ((∃δ. Unifier δ t t') ⟶ Γ t = Γ t')"
| "tfrstp (Inequality X F) = (
      (∀x ∈ fvpairs F - set X. ∃a. Γ (Var x) = TAtom a) ∨
      (∀f T. Fun f T ∈ subtermsset (trmspairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X)))"
| "tfrstp _ = 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 tfrst where
  "tfrst S ≡ tfrset (trmsst S) ∧ list_all tfrstp S"
```

3.3.2 Small Lemmata

```
lemma tfrstp_list_all_alt_def:
  "list_all tfrstp 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 ∈ fvpairs F - set X. ∃a. Γ (Var x) = TAtom a)
    ∨ (∀f T. Fun f T ∈ subtermsset (trmspairs F) ⟶ T = [] ∨ (∃s ∈ set T. s ∉ Var ` set X))))"
  (is "?P S ⟷ ?Q S")
⟨proof⟩
```

```
lemma Γ_wf': "wftrm t ⟹ wftrm (Γ t)"
⟨proof⟩
```

```
lemma fun_type_inv: assumes "Γ t = TComp f T" shows "arity f > 0" "public f"
⟨proof⟩
```

```
lemma fun_type_inv_wf: assumes "Γ t = TComp f T" "wftrm t" shows "arity f = length T"
⟨proof⟩
```

```
lemma const_type_inv: "Γ (Fun c X) = TAtom a ⟹ arity c = 0"
⟨proof⟩
```

```
lemma const_type_inv_wf: assumes "Γ (Fun c X) = TAtom a" and "wftrm (Fun c X)" shows "X = []"
⟨proof⟩
```

lemma *const_type'*: "$\forall c \in \mathcal{C}.\ \exists a \in \mathfrak{T}_a.\ \forall X.\ \Gamma$ (Fun c X) = TAtom a" ⟨*proof*⟩
lemma *fun_type'*: "$\forall f \in \Sigma_f.\ \forall X.\ \Gamma$ (Fun f X) = TComp f (map Γ X)" ⟨*proof*⟩

lemma *infinite_public_consts[simp]*: "infinite {c. public c $\wedge$ arity c = 0}"
⟨*proof*⟩

lemma *infinite_fun_syms[simp]*:
"infinite {c. public c $\wedge$ arity c > 0} $\Longrightarrow$ infinite Σ_f"
"infinite $\mathcal{C}$" "infinite $\mathcal{C}_{pub}$" "infinite (UNIV::'fun set)"
⟨*proof*⟩

lemma *id_univ_proper_subset[simp]*: "$\Sigma_f \subset$ UNIV" "($\exists$f. arity f > 0) $\Longrightarrow \mathcal{C} \subset$ UNIV"
⟨*proof*⟩

lemma *exists_fun_notin_funs_term*: "$\exists$f::'fun. f $\notin$ funs_term t"
⟨*proof*⟩

lemma *exists_fun_notin_funs_terms*:
assumes "finite M" **shows** "$\exists$f::'fun. f $\notin \bigcup$(funs_term ‘ M)"
⟨*proof*⟩

lemma *exists_notin_funs$_{st}$*: "$\exists$f. f $\notin$ funs$_{st}$ (S::('fun,'var) strand)"
⟨*proof*⟩

lemma *infinite_typed_consts'*: "infinite {c. Γ (Fun c []) = TAtom a $\wedge$ public c $\wedge$ arity c = 0}"
⟨*proof*⟩

lemma *atypes_inhabited*: "$\exists$c. Γ (Fun c []) = TAtom a $\wedge$ wf$_{trm}$ (Fun c []) $\wedge$ public c $\wedge$ arity c = 0"
⟨*proof*⟩

lemma *atype_ground_term_ex*: "$\exists$t. fv t = {} $\wedge$ Γ t = TAtom a $\wedge$ wf$_{trm}$ t"
⟨*proof*⟩

lemma *fun_type_id_eq*: "Γ (Fun f X) = TComp g Y $\Longrightarrow$ f = g"
⟨*proof*⟩

lemma *fun_type_length_eq*: "Γ (Fun f X) = TComp g Y $\Longrightarrow$ length X = length Y"
⟨*proof*⟩

lemma *type_ground_inhabited*: "$\exists$t'. fv t' = {} $\wedge$ Γ t = Γ t'"
⟨*proof*⟩

lemma *type_wfttype_inhabited*:
assumes "$\bigwedge$f T. Fun f T $\sqsubseteq \tau \Longrightarrow$ 0 < arity f" "wf$_{trm}$ τ"
shows "$\exists$t. Γ t = τ $\wedge$ wf$_{trm}$ t"
⟨*proof*⟩

lemma *type_pgwt_inhabited*: "wf$_{trm}$ t $\Longrightarrow$ $\exists$t'. Γ t = Γ t' $\wedge$ public_ground_wf_term t'"
⟨*proof*⟩

lemma *pgwt_type_map*:
assumes "public_ground_wf_term t"
shows "Γ t = TAtom a $\Longrightarrow$ $\exists$f. t = Fun f []" "Γ t = TComp g Y $\Longrightarrow$ $\exists$X. t = Fun g X $\wedge$ map Γ X = Y"
⟨*proof*⟩

lemma *wt_subst_Var[simp]*: "wt$_{subst}$ Var" ⟨*proof*⟩

lemma *wt_subst_trm*: "($\bigwedge$v. v $\in$ fv t $\Longrightarrow$ Γ (Var v) = Γ (ϑ v)) $\Longrightarrow$ Γ t = Γ (t $\cdot$ ϑ)"
⟨*proof*⟩

lemma *wt_subst_trm'*: "$[\![$wt$_{subst}$ σ; Γ s = Γ t$]\!]$ $\Longrightarrow$ Γ (s $\cdot$ σ) = Γ (t $\cdot$ σ)"
⟨*proof*⟩

lemma *wt_subst_trm'':* "$wt_{subst}\ \sigma \Longrightarrow \Gamma\ t = \Gamma\ (t \cdot \sigma)$"
⟨*proof*⟩

lemma *wt_subst_compose:*
assumes "$wt_{subst}\ \vartheta$" "$wt_{subst}\ \delta$" **shows** "$wt_{subst}\ (\vartheta \circ_s \delta)$"
⟨*proof*⟩

lemma *wt_subst_TAtom_Var_cases:*
assumes ϑ: "$wt_{subst}\ \vartheta$" "wf_{trms} (subst_range ϑ)"
and x: "Γ (Var x) = TAtom a"
shows "$(\exists y.\ \vartheta\ x = Var\ y) \vee (\exists c.\ \vartheta\ x = Fun\ c\ [])$"
⟨*proof*⟩

lemma *wt_subst_TAtom_fv:*
assumes ϑ: "$wt_{subst}\ \vartheta$" "$\forall x.\ wf_{trm}\ (\vartheta\ x)$"
and "$\forall x \in fv\ t - X.\ \exists a.\ \Gamma\ (Var\ x) = TAtom\ a$"
shows "$\forall x \in fv\ (t \cdot \vartheta) - fv_{set}\ (\vartheta$ ` $X).\ \exists a.\ \Gamma\ (Var\ x) = TAtom\ a$"
⟨*proof*⟩

lemma *wt_subst_TAtom_subterms_subst:*
assumes "$wt_{subst}\ \vartheta$" "$\forall x \in fv\ t.\ \exists a.\ \Gamma\ (Var\ x) = TAtom\ a$" "$wf_{trms}\ (\vartheta$ ` fv t)"
shows "subterms $(t \cdot \vartheta)$ = subterms $t \cdot_{set} \vartheta$"
⟨*proof*⟩

lemma *wt_subst_TAtom_subterms_set_subst:*
assumes "$wt_{subst}\ \vartheta$" "$\forall x \in fv_{set}\ M.\ \exists a.\ \Gamma\ (Var\ x) = TAtom\ a$" "$wf_{trms}\ (\vartheta$ ` $fv_{set}\ M)$"
shows "$subterms_{set}\ (M \cdot_{set} \vartheta) = subterms_{set}\ M \cdot_{set} \vartheta$"
⟨*proof*⟩

lemma *wt_subst_subst_upd:*
assumes "$wt_{subst}\ \vartheta$"
and "Γ (Var x) = Γ t"
shows "$wt_{subst}\ (\vartheta(x := t))$"
⟨*proof*⟩

lemma *wt_subst_const_fv_type_eq:*
assumes "$\forall x \in fv\ t.\ \exists a.\ \Gamma\ (Var\ x) = TAtom\ a$"
and δ: "$wt_{subst}\ \delta$" "wf_{trms} (subst_range δ)"
shows "$\forall x \in fv\ (t \cdot \delta).\ \exists y \in fv\ t.\ \Gamma\ (Var\ x) = \Gamma\ (Var\ y)$"
⟨*proof*⟩

lemma *TComp_term_cases:*
assumes "$wf_{trm}\ t$" "Γ t = TComp f T"
shows "$(\exists v.\ t = Var\ v) \vee (\exists T'.\ t = Fun\ f\ T' \wedge T = map\ \Gamma\ T' \wedge T' \neq [])$"
⟨*proof*⟩

lemma *TAtom_term_cases:*
assumes "$wf_{trm}\ t$" "$\Gamma\ t = TAtom\ \tau$"
shows "$(\exists v.\ t = Var\ v) \vee (\exists f.\ t = Fun\ f\ [])$"
⟨*proof*⟩

lemma *subtermeq_imp_subtermtypeeq:*
assumes "$wf_{trm}\ t$" "$s \sqsubseteq t$"
shows "$\Gamma\ s \sqsubseteq \Gamma\ t$"
⟨*proof*⟩

lemma *subterm_funs_term_in_type:*
assumes "$wf_{trm}\ t$" "$Fun\ f\ T \sqsubseteq t$" "$\Gamma\ (Fun\ f\ T) = TComp\ f\ (map\ \Gamma\ T)$"
shows "$f \in$ funs_term $(\Gamma\ t)$"
⟨*proof*⟩

lemma *wt_subst_fv_termtype_subterm:*
assumes "$x \in fv\ (\vartheta\ y)$"

```
    and "wt_subst ϑ"
    and "wf_trm (ϑ y)"
  shows "Γ (Var x) ⊑ Γ (Var y)"
⟨proof⟩

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)"
⟨proof⟩

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")
⟨proof⟩

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⟩

lemma Ana_subterm_type:
  assumes "Ana t = (K,M)"
    and "wf_trm t"
    and "m ∈ set M"
  shows "Γ m ⊑ Γ t"
⟨proof⟩

lemma wf_trm_TAtom_subterms:
  assumes "wf_trm t" "Γ t = TAtom τ"
  shows "subterms t = {t}"
⟨proof⟩

lemma wf_trm_TComp_subterm:
  assumes "wf_trm s" "t ⊏ s"
  obtains f T where "Γ s = TComp f T"
⟨proof⟩

lemma SMP_empty[simp]: "SMP {} = {}"
⟨proof⟩

lemma SMP_I:
  assumes "s ∈ M" "wt_subst δ" "t ⊑ s · δ" "⋀v. wf_trm (δ v)"
  shows "t ∈ SMP M"
⟨proof⟩

lemma SMP_wf_trm:
  assumes "t ∈ SMP M" "wf_trms M"
  shows "wf_trm t"
⟨proof⟩

lemma SMP_ikI[intro]: "t ∈ ik_st S ⟹ t ∈ SMP (trms_st S)" ⟨proof⟩

lemma MP_setI[intro]: "x ∈ set S ⟹ trms_stp x ⊆ trms_st S" ⟨proof⟩

lemma SMP_setI[intro]: "x ∈ set S ⟹ trms_stp x ⊆ SMP (trms_st S)" ⟨proof⟩

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*⟩

lemma *SMP_union: "SMP (A ∪ B) = SMP A ∪ SMP B"*
⟨*proof*⟩

lemma *SMP_append[simp]: "SMP ($trms_{st}$ (S@S')) = SMP ($trms_{st}$ S) ∪ SMP ($trms_{st}$ S')" (**is** "?A = ?B")*
⟨*proof*⟩

lemma *SMP_mono: "A ⊆ B ⟹ SMP A ⊆ SMP B"*
⟨*proof*⟩

lemma *SMP_Union: "SMP (⋃m ∈ M. f m) = (⋃m ∈ M. SMP (f m))"*
⟨*proof*⟩

lemma *SMP_singleton_ex:*
  "t ∈ SMP M ⟹ (∃m ∈ M. t ∈ SMP {m})"
  "m ∈ M ⟹ t ∈ SMP {m} ⟹ t ∈ SMP M"
⟨*proof*⟩

lemma *SMP_Cons: "SMP ($trms_{st}$ (x#S)) = SMP ($trms_{st}$ [x]) ∪ SMP ($trms_{st}$ S)"*
⟨*proof*⟩

lemma *SMP_Nil[simp]: "SMP ($trms_{st}$ []) = {}"*
⟨*proof*⟩

lemma *SMP_subset_union_eq:* **assumes** *"M ⊆ SMP N"* **shows** *"SMP N = SMP (M ∪ N)"*
⟨*proof*⟩

lemma *SMP_subterms_subset: "$subterms_{set}$ M ⊆ SMP M"*
⟨*proof*⟩

lemma *SMP_SMP_subset: "N ⊆ SMP M ⟹ SMP N ⊆ SMP M"*
⟨*proof*⟩

lemma *wt_subst_rm_vars: "wt_{subst} δ ⟹ wt_{subst} (rm_vars X δ)"*
⟨*proof*⟩

lemma *wt_subst_SMP_subset:*
  assumes *"$trms_{st}$ S ⊆ SMP S'" "wt_{subst} δ" "wf_{trms} (subst_range δ)"*
  shows *"$trms_{st}$ (S $\cdot_{st}$ δ) ⊆ SMP S'"*
⟨*proof*⟩

lemma *MP_subset_SMP: "⋃($trms_{stp}$ ` set S) ⊆ SMP ($trms_{st}$ S)" "$trms_{st}$ S ⊆ SMP ($trms_{st}$ S)" "M ⊆ SMP M"*
⟨*proof*⟩

lemma *SMP_fun_map_snd_subset: "SMP ($trms_{st}$ (map Send X)) ⊆ SMP ($trms_{st}$ [Send (Fun f X)])"*
⟨*proof*⟩

lemma *SMP_wt_subst_subset:*
  assumes *"t ∈ SMP (M $\cdot_{set}$ $\mathcal{I}$)" "wt_{subst} $\mathcal{I}$" "wf_{trms} (subst_range $\mathcal{I}$)"*
  shows *"t ∈ SMP M"*
⟨*proof*⟩

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*⟩

lemma *SMP_consts:*
  assumes *"∀t ∈ M. ∃c. t = Fun c []"*

```
    and "∀t ∈ M. Ana t = ([], [])"
  shows "SMP M = M"
⟨proof⟩

lemma SMP_subterms_eq:
  "SMP (subterms_set M) = SMP M"
⟨proof⟩

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)))"
⟨proof⟩

lemma id_type_eq:
  assumes "Γ (Fun f X) = Γ (Fun g Y)"
  shows "f ∈ C ⟹ g ∈ C" "f ∈ Σ_f ⟹ g ∈ Σ_f"
⟨proof⟩

lemma fun_type_arg_cong:
  assumes "f ∈ Σ_f" "g ∈ Σ_f" "Γ (Fun f (x#X)) = Γ (Fun g (y#Y))"
  shows "Γ x = Γ y" "Γ (Fun f X) = Γ (Fun g Y)"
⟨proof⟩

lemma fun_type_arg_cong':
  assumes "f ∈ Σ_f" "g ∈ Σ_f" "Γ (Fun f (X@x#X')) = Γ (Fun g (Y@y#Y'))" "length X = length Y"
  shows "Γ x = Γ y"
⟨proof⟩

lemma fun_type_param_idx: "Γ (Fun f T) = Fun g S ⟹ i < length T ⟹ Γ (T ! i) = S ! i"
⟨proof⟩

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"
⟨proof⟩

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')"
⟨proof⟩

lemma tfr_stp_all_append:
  assumes "list_all tfr_stp S" "list_all tfr_stp S'"
  shows "list_all tfr_stp (S@S')"
⟨proof⟩

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)"
⟨proof⟩

lemma tfr_stp_all_same_type:
  "list_all tfr_stp (S@Equality a t t'#S') ⟹ Unifier δ t t' ⟹ Γ t = Γ t'"
⟨proof⟩
```

lemma *tfr_subset:*
  "$\bigwedge$A B. tfr$_{set}$ (A ∪ B) ⟹ tfr$_{set}$ A"
  "$\bigwedge$A B. tfr$_{set}$ B ⟹ A ⊆ B ⟹ tfr$_{set}$ A"
  "$\bigwedge$A B. tfr$_{set}$ B ⟹ SMP A ⊆ SMP B ⟹ tfr$_{set}$ A"
⟨*proof*⟩

lemma *tfr_empty[simp]:* "tfr$_{set}$ {}"
⟨*proof*⟩

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*⟩

lemma *dual$_{st}$_tfr$_{stp}$:* "list_all tfr$_{stp}$ S ⟹ list_all tfr$_{stp}$ (dual$_{st}$ S)"
⟨*proof*⟩

lemma *subst_var_inv_wt:*
  assumes "wt$_{subst}$ δ"
  shows "wt$_{subst}$ (subst_var_inv δ X)"
⟨*proof*⟩

lemma *subst_var_inv_wf_trms:*
  "wf$_{trms}$ (subst_range (subst_var_inv δ X))"
⟨*proof*⟩

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"
⟨*proof*⟩

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*⟩

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)"
⟨*proof*⟩

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⟩

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"
⟨proof⟩

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"
⟨proof⟩

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⟩

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⟩

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"
⟨proof⟩

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"
⟨proof⟩

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)"
⟨proof⟩

lemma (in typed_model) has_all_instances_of_if_subset:
  assumes "N ⊆ M"
  shows "has_all_wt_instances_of Γ N M"
⟨proof⟩

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"
⟨proof⟩
```

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⟩

lemma var_rename_fv_disjoint:
  "fv s ∩ fv (t · var_rename (max_var s)) = {}"
⟨proof⟩

lemma var_rename_fv_set_disjoint:
  assumes "finite M" "s ∈ M"
  shows "fv s ∩ fv (t · var_rename (max_var_set (fvset M))) = {}"
⟨proof⟩

lemma var_rename_fv_set_disjoint':
  assumes "finite M"
  shows "fvset M ∩ fvset (N ·set var_rename (max_var_set (fvset M))) = {}"
⟨proof⟩

lemma var_rename_is_renaming[simp]:
  "subst_range (var_rename n) ⊆ range Var"
  "subst_range (var_rename_inv n) ⊆ range Var"
⟨proof⟩

lemma var_rename_wt[simp]:
  "wtsubst (var_rename n)"
  "wtsubst (var_rename_inv n)"
⟨proof⟩

lemma var_rename_wt':
  assumes "wtsubst δ" "s = m · δ"
```

```
  shows "wt_subst (var_rename_inv n ∘s δ)" "s = m · var_rename n · var_rename_inv n ∘s δ"
⟨proof⟩

lemma var_rename_wf_trms_range[simp]:
  "wf_trms (subst_range (var_rename n))"
  "wf_trms (subst_range (var_rename_inv n))"
⟨proof⟩

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))"
⟨proof⟩

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"
⟨proof⟩

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"
⟨proof⟩

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⟩

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⟩

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⟩

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"
⟨proof⟩
```

```
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)"
⟨proof⟩ 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⟩ 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⟩

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⟩

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⟩

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⟩
end

lemma tfr_set_if_comp_tfr_set:
  assumes "comp_tfr_set arity Ana Γ M"
  shows "tfr_set (set M)"
⟨proof⟩

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)"
⟨proof⟩

lemma tfr_stp_is_comp_tfr_stp: "tfr_stp a = comp_tfr_stp Γ a"
```

⟨proof⟩

```
lemma tfr_st_if_comp_tfr_st:
  assumes "comp_tfr_st arity Ana Γ M S"
  shows "tfr_st S"
⟨proof⟩

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"
⟨proof⟩
```

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⟩ 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⟩ 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⟩

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⟩

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'"
⟨proof⟩ 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"
⟨proof⟩ 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')"
⟨proof⟩ 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')"
⟨proof⟩

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⟩

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⟩
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⟩

lemma wt_grounding_subst_exists:
```

```
  "∃ϑ. wt_subst ϑ ∧ wf_trms (subst_range ϑ) ∧ fv (t · ϑ) = {}"
⟨proof⟩ 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⟩ lemma fresh_pgwt_empty_synth:
  assumes "finite S" "wf_trm t"
  shows "{} ⊢_c fresh_pgwt S (Γ t)"
⟨proof⟩ 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⟩ 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⟩ 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⟩

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)"
⟨proof⟩ 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⟩

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)"
⟨proof⟩

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⟩
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⟩
```

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⟩))"
⟨proof⟩

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"
⟨proof⟩

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)"
⟨proof⟩

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)"
⟨proof⟩ 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)"
```

```
⟨proof⟩ definition subst_apply_extstrand (infix "·_est" 51) where "S ·_est ϑ ≡ map (λx. x ·_estp ϑ) S"

private abbreviation update_st::"('fun,'var) strands ⇒ ('fun,'var) strand ⇒ ('fun,'var) strands"
where
  "update_st 𝒮 S ≡ (case S of Nil ⇒ 𝒮 - {S} | Cons _ S' ⇒ insert S' (𝒮 - {S}))"

private inductive_set decomps_est::
  "('fun,'var) terms ⇒ ('fun,'var) terms ⇒ ('fun,'var) subst ⇒ ('fun,'var) extstrands"

for ℳ and 𝒩 and ℐ where
  Nil: "[] ∈ decomps_est ℳ 𝒩 ℐ"
| Decomp: "⟦𝒟 ∈ decomps_est ℳ 𝒩 ℐ; Fun f T ∈ subterms_set (ℳ ∪ 𝒩);
           Ana (Fun f T) = (K,M); M ≠ [];
           (ℳ ∪ ik_est 𝒟) ·_set ℐ ⊢_c Fun f T · ℐ;
           ⋀k. k ∈ set K ⟹ (ℳ ∪ ik_est 𝒟) ·_set ℐ ⊢_c k · ℐ⟧
           ⟹ 𝒟@[Decomp (Fun f T)] ∈ decomps_est ℳ 𝒩 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 M₀ ℐ []"
| Send: "sem_est_d M₀ ℐ S ⟹ (ik_est S ∪ M₀) ·_set ℐ ⊢ t · ℐ ⟹ sem_est_d M₀ ℐ (S@[Step (send⟨t⟩_st)])"
| Receive: "sem_est_d M₀ ℐ S ⟹ sem_est_d M₀ ℐ (S@[Step (receive⟨t⟩_st)])"
| Equality: "sem_est_d M₀ ℐ S ⟹ t · ℐ = t' · ℐ ⟹ sem_est_d M₀ ℐ (S@[Step (⟨a: t ≐ t'⟩_st)])"
| Inequality: "sem_est_d M₀ ℐ S
    ⟹ ineq_model ℐ X F
    ⟹ sem_est_d M₀ ℐ (S@[Step (∀X⟨∨≠: F⟩_st)])"
| Decompose: "sem_est_d M₀ ℐ S ⟹ (ik_est S ∪ M₀) ·_set ℐ ⊢ t · ℐ ⟹ Ana t = (K, M)
    ⟹ (⋀k. k ∈ set K ⟹ (ik_est S ∪ M₀) ·_set ℐ ⊢ k · ℐ) ⟹ sem_est_d M₀ 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 M₀ ℐ []"
| Send: "sem_est_c M₀ ℐ S ⟹ (ik_est S ∪ M₀) ·_set ℐ ⊢_c t · ℐ ⟹ sem_est_c M₀ ℐ (S@[Step (send⟨t⟩_st)])"
| Receive: "sem_est_c M₀ ℐ S ⟹ sem_est_c M₀ ℐ (S@[Step (receive⟨t⟩_st)])"
| Equality: "sem_est_c M₀ ℐ S ⟹ t · ℐ = t' · ℐ ⟹ sem_est_c M₀ ℐ (S@[Step (⟨a: t ≐ t'⟩_st)])"
| Inequality: "sem_est_c M₀ ℐ S
    ⟹ ineq_model ℐ X F
    ⟹ sem_est_c M₀ ℐ (S@[Step (∀X⟨∨≠: F⟩_st)])"
| Decompose: "sem_est_c M₀ ℐ S ⟹ (ik_est S ∪ M₀) ·_set ℐ ⊢_c t · ℐ ⟹ Ana t = (K, M)
    ⟹ (⋀k. k ∈ set K ⟹ (ik_est S ∪ M₀) ·_set ℐ ⊢_c k · ℐ) ⟹ sem_est_c M₀ ℐ (S@[Decomp t])"
```

Preliminary Lemmata

```
private lemma wf_sts_wf_sts':
  "wf_sts 𝒮 = wf_sts' 𝒮 []"
⟨proof⟩ lemma decomp_ik:
  assumes "Ana t = (K,M)"
  shows "ik_st (decomp t) = set M"
⟨proof⟩ lemma decomp_assignment_rhs_empty:
  assumes "Ana t = (K,M)"
  shows "assignment_rhs_st (decomp t) = {}"
⟨proof⟩ lemma decomp_tfr_stp:
  "list_all tfr_stp (decomp t)"
⟨proof⟩ lemma trms_est_ikI:
  "t ∈ ik_est A ⟹ t ∈ subterms_set (trms_est A)"
⟨proof⟩ lemma trms_est_ik_assignment_rhsI:
  "t ∈ ik_est A ∪ assignment_rhs_est A ⟹ t ∈ subterms_set (trms_est A)"
⟨proof⟩ lemma trms_est_ik_subtermsI:
```

assumes *"t ∈ subterms$_{set}$ (ik$_{est}$ A)"*
shows *"t ∈ subterms$_{set}$ (trms$_{est}$ A)"*
⟨*proof*⟩ **lemma** *trms$_{est}$D:*
assumes *"t ∈ trms$_{est}$ A"*
shows *"t ∈ trms$_{st}$ (to_st A)"*
⟨*proof*⟩ **lemma** *subst_apply_extstrand_nil[simp]:*
"[] $\cdot_{est}$ ϑ = []"
⟨*proof*⟩ **lemma** *subst_apply_extstrand_singleton[simp]:*
"[Step (receive⟨t⟩$_{st}$)] $\cdot_{est}$ ϑ = [Step (Receive (t $\cdot$ ϑ))]"
"[Step (send⟨t⟩$_{st}$)] $\cdot_{est}$ ϑ = [Step (Send (t $\cdot$ ϑ))]"
"[Step (⟨a: t $\doteq$ t'⟩$_{st}$)] $\cdot_{est}$ ϑ = [Step (Equality a (t $\cdot$ ϑ) (t' $\cdot$ ϑ))]"
"[Decomp t] $\cdot_{est}$ ϑ = [Decomp (t $\cdot$ ϑ)]"
⟨*proof*⟩ **lemma** *extstrand_subst_hom:*
"(S@S') $\cdot_{est}$ ϑ = (S $\cdot_{est}$ ϑ)@(S' $\cdot_{est}$ ϑ)" "(x#S) $\cdot_{est}$ ϑ = (x $\cdot_{estp}$ ϑ)#(S $\cdot_{est}$ ϑ)"
⟨*proof*⟩ **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*⟩ **lemma** *bvars$_{est}$_cons: "bvars$_{est}$ (x#X) = bvars$_{est}$ [x] ∪ bvars$_{est}$ X"*
⟨*proof*⟩ **lemma** *bvars$_{est}$_append: "bvars$_{est}$ (A@B) = bvars$_{est}$ A ∪ bvars$_{est}$ B"*
⟨*proof*⟩ **lemma** *fv$_{est}$_cons: "fv$_{est}$ (x#X) = fv$_{est}$ [x] ∪ fv$_{est}$ X"*
⟨*proof*⟩ **lemma** *fv$_{est}$_append: "fv$_{est}$ (A@B) = fv$_{est}$ A ∪ fv$_{est}$ B"*
⟨*proof*⟩ **lemma** *bvars_decomp: "bvars$_{est}$ (A@[Decomp t]) = bvars$_{est}$ A" "bvars$_{est}$ (Decomp t#A) = bvars$_{est}$ A"*
⟨*proof*⟩ **lemma** *bvars_decomp_rm: "bvars$_{est}$ (decomp_rm$_{est}$ A) = bvars$_{est}$ A"*
⟨*proof*⟩ **lemma** *fv_decomp_rm: "fv$_{est}$ (decomp_rm$_{est}$ A) ⊆ fv$_{est}$ A"*
⟨*proof*⟩ **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*⟩ **lemma** *wfrestrictedvars$_{est}$_decomp_rm$_{est}$_subset:*
"wfrestrictedvars$_{est}$ (decomp_rm$_{est}$ A) ⊆ wfrestrictedvars$_{est}$ A"
⟨*proof*⟩ **lemma** *wfrestrictedvars$_{est}$_eq_wfrestrictedvars$_{st}$:*
"wfrestrictedvars$_{est}$ A = wfrestrictedvars$_{st}$ (to_st A)"
⟨*proof*⟩ **lemma** *decomp_set_unfold:*
assumes *"Ana t = (K, M)"*
shows *"set (decomp t) = {send⟨t⟩$_{st}$} ∪ (Send ‘ set K) ∪ (Receive ‘ set M)"*
⟨*proof*⟩ **lemma** *ik$_{est}$_finite: "finite (ik$_{est}$ A)"*
⟨*proof*⟩ **lemma** *assignment_rhs$_{est}$_finite: "finite (assignment_rhs$_{est}$ A)"*
⟨*proof*⟩ **lemma** *to_est_append: "to_est (A@B) = to_est A@to_est B"*
⟨*proof*⟩ **lemma** *to_st_to_est_inv: "to_st (to_est A) = A"*
⟨*proof*⟩ **lemma** *to_st_append: "to_st (A@B) = (to_st A)@(to_st B)"*
⟨*proof*⟩ **lemma** *to_st_cons: "to_st (a#B) = (to_st [a])@(to_st B)"*
⟨*proof*⟩ **lemma** *wfrestrictedvars$_{est}$_split:*
"wfrestrictedvars$_{est}$ (x#S) = wfrestrictedvars$_{est}$ [x] ∪ wfrestrictedvars$_{est}$ S"
"wfrestrictedvars$_{est}$ (S@S') = wfrestrictedvars$_{est}$ S ∪ wfrestrictedvars$_{est}$ S'"
⟨*proof*⟩ **lemma** *ik$_{est}$_append: "ik$_{est}$ (A@B) = ik$_{est}$ A ∪ ik$_{est}$ B"*
⟨*proof*⟩ **lemma** *assignment_rhs$_{est}$_append:*
"assignment_rhs$_{est}$ (A@B) = assignment_rhs$_{est}$ A ∪ assignment_rhs$_{est}$ B"
⟨*proof*⟩ **lemma** *ik$_{est}$_cons: "ik$_{est}$ (a#A) = ik$_{est}$ [a] ∪ ik$_{est}$ A"*
⟨*proof*⟩ **lemma** *ik$_{est}$_append_subst:*
"ik$_{est}$ (A@B $\cdot_{est}$ ϑ) = ik$_{est}$ (A $\cdot_{est}$ ϑ) ∪ ik$_{est}$ (B $\cdot_{est}$ ϑ)"
"ik$_{est}$ (A@B) $\cdot_{set}$ ϑ = (ik$_{est}$ A $\cdot_{set}$ ϑ) ∪ (ik$_{est}$ B $\cdot_{set}$ ϑ)"
⟨*proof*⟩ **lemma** *assignment_rhs$_{est}$_append_subst:*
"assignment_rhs$_{est}$ (A@B $\cdot_{est}$ ϑ) = assignment_rhs$_{est}$ (A $\cdot_{est}$ ϑ) ∪ assignment_rhs$_{est}$ (B $\cdot_{est}$ ϑ)"
"assignment_rhs$_{est}$ (A@B) $\cdot_{set}$ ϑ = (assignment_rhs$_{est}$ A $\cdot_{set}$ ϑ) ∪ (assignment_rhs$_{est}$ B $\cdot_{set}$ ϑ)"
⟨*proof*⟩ **lemma** *ik$_{est}$_cons_subst:*
"ik$_{est}$ (a#A $\cdot_{est}$ ϑ) = ik$_{est}$ ([a $\cdot_{estp}$ ϑ]) ∪ ik$_{est}$ (A $\cdot_{est}$ ϑ)"
"ik$_{est}$ (a#A) $\cdot_{set}$ ϑ = (ik$_{est}$ [a] $\cdot_{set}$ ϑ) ∪ (ik$_{est}$ A $\cdot_{set}$ ϑ)"
⟨*proof*⟩ **lemma** *decomp_rm$_{est}$_append: "decomp_rm$_{est}$ (S@S') = (decomp_rm$_{est}$ S)@(decomp_rm$_{est}$ S')"*
⟨*proof*⟩ **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] = []"
⟨*proof*⟩ **lemma** *decomp_rm$_{est}$_ik_subset: "ik$_{est}$ (decomp_rm$_{est}$ S) ⊆ ik$_{est}$ S"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_ik_subset: "D ∈ decomps$_{est}$ M N $\mathcal{I}$ $\Longrightarrow$ ik$_{est}$ D ⊆ subterms$_{set}$ (M ∪ N)"*

⟨*proof*⟩ **lemma** *decomps$_{est}$_decomp_rm$_{est}$_empty: "D ∈ decomps$_{est}$ M N $\mathcal{I}$ ⟹ decomp_rm$_{est}$ D = []"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_append:*
assumes *"A ∈ decomps$_{est}$ S N $\mathcal{I}$" "B ∈ decomps$_{est}$ S N $\mathcal{I}$"*
shows *"A@B ∈ decomps$_{est}$ S N $\mathcal{I}$"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_subterms:*
assumes *"A' ∈ decomps$_{est}$ M N $\mathcal{I}$"*
shows *"subterms$_{set}$ (ik$_{est}$ A') ⊆ subterms$_{set}$ (M ∪ N)"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_assignment_rhs_empty:*
assumes *"A' ∈ decomps$_{est}$ M N $\mathcal{I}$"*
shows *"assignment_rhs$_{est}$ A' = {}"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_finite_ik_append:*
assumes *"finite M" "M ⊆ decomps$_{est}$ A N $\mathcal{I}$"*
shows *"∃D ∈ decomps$_{est}$ A N $\mathcal{I}$. ik$_{est}$ D = (⋃m ∈ M. ik$_{est}$ m)"*
⟨*proof*⟩ **lemma** *decomp_snd_exists[simp]: "∃D. decomp t = send⟨t⟩$_{st}$#D"*
⟨*proof*⟩ **lemma** *decomp_nonnil[simp]: "decomp t ≠ []"*
⟨*proof*⟩ **lemma** *to_st_nil_inv[dest]: "to_st A = [] ⟹ A = []"*
⟨*proof*⟩ **lemma** *well_analyzedD:*
assumes *"well_analyzed A" "Decomp t ∈ set A"*
shows *"∃f T. t = Fun f T"*
⟨*proof*⟩ **lemma** *well_analyzed_inv:*
assumes *"well_analyzed (A@[Decomp t])"*
shows *"t ∈ subterms$_{set}$ (ik$_{est}$ A ∪ assignment_rhs$_{est}$ A) - (Var ‘ $\mathcal{V}$)"*
⟨*proof*⟩ **lemma** *well_analyzed_split_left_single: "well_analyzed (A@[a]) ⟹ well_analyzed A"*
⟨*proof*⟩ **lemma** *well_analyzed_split_left: "well_analyzed (A@B) ⟹ well_analyzed A"*
⟨*proof*⟩ **lemma** *well_analyzed_append:*
assumes *"well_analyzed A" "well_analyzed B"*
shows *"well_analyzed (A@B)"*
⟨*proof*⟩ **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*⟩ **lemma** *well_analyzed_decomp_rm$_{est}$_fv: "well_analyzed A ⟹ fv$_{est}$ (decomp_rm$_{est}$ A) = fv$_{est}$ A"*
⟨*proof*⟩ **lemma** *sem$_{est}$_d_split_left:* **assumes** *"sem$_{est}$_d M_0 $\mathcal{I}$ ($\mathcal{A}$@$\mathcal{A}$')"* **shows** *"sem$_{est}$_d M_0 $\mathcal{I}$ $\mathcal{A}$"*
⟨*proof*⟩ **lemma** *sem$_{est}$_d_eq_sem_st: "sem$_{est}$_d M_0 $\mathcal{I}$ $\mathcal{A}$ = ⟦M_0; to_st $\mathcal{A}$⟧$_d$' $\mathcal{I}$"*
⟨*proof*⟩ **lemma** *sem$_{est}$_c_eq_sem_st: "sem$_{est}$_c M_0 $\mathcal{I}$ $\mathcal{A}$ = ⟦M_0; to_st $\mathcal{A}$⟧$_c$' $\mathcal{I}$"*
⟨*proof*⟩ **lemma** *sem$_{est}$_c_decomp_rm$_{est}$_deduct_aux:*
assumes *"sem$_{est}$_c M_0 $\mathcal{I}$ A" "t ∈ ik$_{est}$ A ·$_{set}$ $\mathcal{I}$" "t ∉ ik$_{est}$ (decomp_rm$_{est}$ A) ·$_{set}$ $\mathcal{I}$"*
shows *"ik$_{est}$ (decomp_rm$_{est}$ A) ∪ M_0 ·$_{set}$ $\mathcal{I}$ ⊢ t"*
⟨*proof*⟩ **lemma** *sem$_{est}$_c_decomp_rm$_{est}$_deduct:*
assumes *"sem$_{est}$_c M_0 $\mathcal{I}$ A" "ik$_{est}$ A ∪ M_0 ·$_{set}$ $\mathcal{I}$ ⊢$_c$ t"*
shows *"ik$_{est}$ (decomp_rm$_{est}$ A) ∪ M_0 ·$_{set}$ $\mathcal{I}$ ⊢ t"*
⟨*proof*⟩ **lemma** *sem$_{est}$_d_decomp_rm$_{est}$_if_sem$_{est}$_c: "sem$_{est}$_c M_0 $\mathcal{I}$ A ⟹ sem$_{est}$_d M_0 $\mathcal{I}$ (decomp_rm$_{est}$ A)"*
⟨*proof*⟩ **lemma** *sem$_{est}$_c_decomps$_{est}$_append:*
assumes *"sem$_{est}$_c {} $\mathcal{I}$ A" "D ∈ decomps$_{est}$ (ik$_{est}$ A) (assignment_rhs$_{est}$ $\mathcal{A}$) $\mathcal{I}$"*
shows *"sem$_{est}$_c {} $\mathcal{I}$ (A@D)"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_preserves_wf:*
assumes *"D ∈ decomps$_{est}$ (ik$_{est}$ A) (assignment_rhs$_{est}$ A) $\mathcal{I}$" "wf$_{est}$ V A"*
shows *"wf$_{est}$ V (A@D)"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_preserves_model_c:*
assumes *"D ∈ decomps$_{est}$ (ik$_{est}$ A) (assignment_rhs$_{est}$ A) $\mathcal{I}$" "sem$_{est}$_c M_0 $\mathcal{I}$ A"*
shows *"sem$_{est}$_c M_0 $\mathcal{I}$ (A@D)"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_exist_aux:*
assumes *"D ∈ decomps$_{est}$ M N $\mathcal{I}$" "M ∪ ik$_{est}$ D ⊢ t" "¬(M ∪ (ik$_{est}$ D) ⊢$_c$ t)"*
obtains *D'* **where**
"D@D' ∈ decomps$_{est}$ M N $\mathcal{I}$" "M ∪ ik$_{est}$ (D@D') ⊢$_c$ t" "M ∪ ik$_{est}$ D ⊂ M ∪ ik$_{est}$ (D@D')"
⟨*proof*⟩ **lemma** *decomps$_{est}$_ik_max_exist:*
assumes *"finite A" "finite N"*
shows *"∃D ∈ decomps$_{est}$ A N $\mathcal{I}$. ∀D' ∈ decomps$_{est}$ A N $\mathcal{I}$. ik$_{est}$ D' ⊆ ik$_{est}$ D"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_exist:*
assumes *"finite A" "finite N"*
shows *"∃D ∈ decomps$_{est}$ A N $\mathcal{I}$. ∀t. A ⊢ t ⟶ A ∪ ik$_{est}$ D ⊢$_c$ t"*
⟨*proof*⟩ **lemma** *decomps$_{est}$_exist_subst:*
assumes *"ik$_{est}$ A ·$_{set}$ $\mathcal{I}$ ⊢ t · $\mathcal{I}$"*

and "sem$_{est}$_c {} $\mathcal{I}$ A" "wf$_{est}$ {} A" "interpretation$_{subst}$ $\mathcal{I}$"
and "Ana_invar_subst (ik$_{est}$ A $\cup$ assignment_rhs$_{est}$ A)"
and "well_analyzed A"
shows "$\exists D \in$ decomps$_{est}$ (ik$_{est}$ A) (assignment_rhs$_{est}$ A) $\mathcal{I}$. ik$_{est}$ (A@D) $\cdot_{set}$ $\mathcal{I} \vdash_c$ t $\cdot$ $\mathcal{I}$"
⟨*proof*⟩ **lemma** wf$_{sts}$'_update$_{st}$_nil: **assumes** "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" **shows** "wf$_{sts}$' (update$_{st}$ $\mathcal{S}$ []) $\mathcal{A}$"
⟨*proof*⟩ **lemma** wf$_{sts}$'_update$_{st}$_snd:
assumes "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" "send⟨t⟩$_{st}$#S $\in$ $\mathcal{S}$"
shows "wf$_{sts}$' (update$_{st}$ $\mathcal{S}$ (send⟨t⟩$_{st}$#S)) ($\mathcal{A}$@[Step (receive⟨t⟩$_{st}$)])"
⟨*proof*⟩ **lemma** wf$_{sts}$'_update$_{st}$_rcv:
assumes "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" "receive⟨t⟩$_{st}$#S $\in$ $\mathcal{S}$"
shows "wf$_{sts}$' (update$_{st}$ $\mathcal{S}$ (receive⟨t⟩$_{st}$#S)) ($\mathcal{A}$@[Step (send⟨t⟩$_{st}$)])"
⟨*proof*⟩ **lemma** wf$_{sts}$'_update$_{st}$_eq:
assumes "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" "⟨a: t $\doteq$ t'⟩$_{st}$#S $\in$ $\mathcal{S}$"
shows "wf$_{sts}$' (update$_{st}$ $\mathcal{S}$ (⟨a: t $\doteq$ t'⟩$_{st}$#S)) ($\mathcal{A}$@[Step (⟨a: t $\doteq$ t'⟩$_{st}$)])"
⟨*proof*⟩ **lemma** wf$_{sts}$'_update$_{st}$_ineq:
assumes "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" "$\forall X$⟨$\vee\neq$: F⟩$_{st}$#S $\in$ $\mathcal{S}$"
shows "wf$_{sts}$' (update$_{st}$ $\mathcal{S}$ ($\forall X$⟨$\vee\neq$: F⟩$_{st}$#S)) ($\mathcal{A}$@[Step ($\forall X$⟨$\vee\neq$: F⟩$_{st}$)])"
⟨*proof*⟩ **lemma** trms$_{st}$_update$_{st}$_eq:
assumes "x#S $\in$ $\mathcal{S}$"
shows "$\bigcup$(trms$_{st}$ ` update$_{st}$ $\mathcal{S}$ (x#S)) $\cup$ trms$_{stp}$ x = $\bigcup$(trms$_{st}$ ` $\mathcal{S}$)" (**is** "?A = ?B")
⟨*proof*⟩ **lemma** trms$_{st}$_update$_{st}$_eq_snd:
assumes "send⟨t⟩$_{st}$#S $\in$ $\mathcal{S}$" "$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ (send⟨t⟩$_{st}$#S)" "$\mathcal{A}$' = $\mathcal{A}$@[Step (receive⟨t⟩$_{st}$)]"
shows "($\bigcup$(trms$_{st}$ ` $\mathcal{S}$)) $\cup$ (trms$_{est}$ $\mathcal{A}$) = ($\bigcup$(trms$_{st}$ ` $\mathcal{S}$')) $\cup$ (trms$_{est}$ $\mathcal{A}$')"
⟨*proof*⟩ **lemma** trms$_{st}$_update$_{st}$_eq_rcv:
assumes "receive⟨t⟩$_{st}$#S $\in$ $\mathcal{S}$" "$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ (receive⟨t⟩$_{st}$#S)" "$\mathcal{A}$' = $\mathcal{A}$@[Step (send⟨t⟩$_{st}$)]"
shows "($\bigcup$(trms$_{st}$ ` $\mathcal{S}$)) $\cup$ (trms$_{est}$ $\mathcal{A}$) = ($\bigcup$(trms$_{st}$ ` $\mathcal{S}$')) $\cup$ (trms$_{est}$ $\mathcal{A}$')"
⟨*proof*⟩ **lemma** trms$_{st}$_update$_{st}$_eq_eq:
assumes "⟨a: t $\doteq$ t'⟩$_{st}$#S $\in$ $\mathcal{S}$" "$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ (⟨a: t $\doteq$ t'⟩$_{st}$#S)" "$\mathcal{A}$' = $\mathcal{A}$@[Step (⟨a: t $\doteq$ t'⟩$_{st}$)]"
shows "($\bigcup$(trms$_{st}$ ` $\mathcal{S}$)) $\cup$ (trms$_{est}$ $\mathcal{A}$) = ($\bigcup$(trms$_{st}$ ` $\mathcal{S}$')) $\cup$ (trms$_{est}$ $\mathcal{A}$')"
⟨*proof*⟩ **lemma** trms$_{st}$_update$_{st}$_eq_ineq:
assumes "$\forall X$⟨$\vee\neq$: F⟩$_{st}$#S $\in$ $\mathcal{S}$" "$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ ($\forall X$⟨$\vee\neq$: F⟩$_{st}$#S)" "$\mathcal{A}$' = $\mathcal{A}$@[Step ($\forall X$⟨$\vee\neq$: F⟩$_{st}$)]"
shows "($\bigcup$(trms$_{st}$ ` $\mathcal{S}$)) $\cup$ (trms$_{est}$ $\mathcal{A}$) = ($\bigcup$(trms$_{st}$ ` $\mathcal{S}$')) $\cup$ (trms$_{est}$ $\mathcal{A}$')"
⟨*proof*⟩ **lemma** ik$_{st}$_update$_{st}$_subset:
assumes "x#S $\in$ $\mathcal{S}$"
shows "$\bigcup$(ik$_{st}$`dual$_{st}$ ` (update$_{st}$ $\mathcal{S}$ (x#S))) $\subseteq$ $\bigcup$(ik$_{st}$`dual$_{st}$ ` $\mathcal{S}$)" (**is** ?A)
"$\bigcup$(assignment_rhs$_{st}$ ` (update$_{st}$ $\mathcal{S}$ (x#S))) $\subseteq$ $\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$)" (**is** ?B)
⟨*proof*⟩ **lemma** ik$_{st}$_update$_{st}$_subset_snd:
assumes "send⟨t⟩$_{st}$#S $\in$ $\mathcal{S}$"
"$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ (send⟨t⟩$_{st}$#S)"
"$\mathcal{A}$' = $\mathcal{A}$@[Step (receive⟨t⟩$_{st}$)]"
shows "($\bigcup$(ik$_{st}$ ` dual$_{st}$ ` $\mathcal{S}$')) $\cup$ (ik$_{est}$ $\mathcal{A}$') $\subseteq$
($\bigcup$(ik$_{st}$ ` dual$_{st}$ ` $\mathcal{S}$)) $\cup$ (ik$_{est}$ $\mathcal{A}$)" (**is** ?A)
"($\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$')) $\cup$ (assignment_rhs$_{est}$ $\mathcal{A}$') $\subseteq$
($\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$)) $\cup$ (assignment_rhs$_{est}$ $\mathcal{A}$)" (**is** ?B)
⟨*proof*⟩ **lemma** ik$_{st}$_update$_{st}$_subset_rcv:
assumes "receive⟨t⟩$_{st}$#S $\in$ $\mathcal{S}$"
"$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ (receive⟨t⟩$_{st}$#S)"
"$\mathcal{A}$' = $\mathcal{A}$@[Step (send⟨t⟩$_{st}$)]"
shows "($\bigcup$(ik$_{st}$ ` dual$_{st}$ ` $\mathcal{S}$')) $\cup$ (ik$_{est}$ $\mathcal{A}$') $\subseteq$
($\bigcup$(ik$_{st}$ ` dual$_{st}$ ` $\mathcal{S}$)) $\cup$ (ik$_{est}$ $\mathcal{A}$)" (**is** ?A)
"($\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$')) $\cup$ (assignment_rhs$_{est}$ $\mathcal{A}$') $\subseteq$
($\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$)) $\cup$ (assignment_rhs$_{est}$ $\mathcal{A}$)" (**is** ?B)
⟨*proof*⟩ **lemma** ik$_{st}$_update$_{st}$_subset_eq:
assumes "⟨a: t $\doteq$ t'⟩$_{st}$#S $\in$ $\mathcal{S}$"
"$\mathcal{S}$' = update$_{st}$ $\mathcal{S}$ (⟨a: t $\doteq$ t'⟩$_{st}$#S)"
"$\mathcal{A}$' = $\mathcal{A}$@[Step (⟨a: t $\doteq$ t'⟩$_{st}$)]"
shows "($\bigcup$(ik$_{st}$ ` dual$_{st}$ ` $\mathcal{S}$')) $\cup$ (ik$_{est}$ $\mathcal{A}$') $\subseteq$
($\bigcup$(ik$_{st}$ ` dual$_{st}$ ` $\mathcal{S}$)) $\cup$ (ik$_{est}$ $\mathcal{A}$)" (**is** ?A)
"($\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$')) $\cup$ (assignment_rhs$_{est}$ $\mathcal{A}$') $\subseteq$
($\bigcup$(assignment_rhs$_{st}$ ` $\mathcal{S}$)) $\cup$ (assignment_rhs$_{est}$ $\mathcal{A}$)" (**is** ?B)
⟨*proof*⟩ **lemma** ik$_{st}$_update$_{st}$_subset_ineq:
assumes "$\forall X$⟨$\vee\neq$: F⟩$_{st}$#S $\in$ $\mathcal{S}$"

```
          "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⟩
```

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"
⟨proof⟩ 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 "$\bigwedge$f T. ⟦Fun f T ∈ subterms$_{set}$ (ik$_{est}$ $\mathcal{A}$ ∪ assignment_rhs$_{est}$ $\mathcal{A}$); $\mathcal{S}$' = $\mathcal{S}$; $\mathcal{A}$' = $\mathcal{A}$@[Decomp (Fun f T)]⟧ ⟹ P"
shows "P"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_wf_prot:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')" "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$"
shows "wf$_{sts}$' $\mathcal{S}$' $\mathcal{A}$'"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_wf_is:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')" "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" "wf$_{st}$ V (to_st $\mathcal{A}$)"
shows "wf$_{st}$ V (to_st $\mathcal{A}$')"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_tfr$_{set}$:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')"
and "tfr$_{set}$ (($\bigcup$(trms$_{st}$ ‘ $\mathcal{S}$)) ∪ (trms$_{est}$ $\mathcal{A}$))"
and "wf$_{trms}$ (($\bigcup$(trms$_{st}$ ‘ $\mathcal{S}$)) ∪ (trms$_{est}$ $\mathcal{A}$))"
shows "tfr$_{set}$ (($\bigcup$(trms$_{st}$ ‘ $\mathcal{S}$')) ∪ (trms$_{est}$ $\mathcal{A}$')) ∧ wf$_{trms}$ (($\bigcup$(trms$_{st}$ ‘ $\mathcal{S}$')) ∪ (trms$_{est}$ $\mathcal{A}$'))"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_tfr$_{stp}$:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')" "∀S ∈ $\mathcal{S}$ ∪ {to_st $\mathcal{A}$}. list_all tfr$_{stp}$ S"
shows "∀S ∈ $\mathcal{S}$' ∪ {to_st $\mathcal{A}$'}. list_all tfr$_{stp}$ S"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_well_analyzed:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')" "well_analyzed $\mathcal{A}$"
shows "well_analyzed $\mathcal{A}$'"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_Ana_invar_subst:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')"
and "Ana_invar_subst (
($\bigcup$(ik$_{st}$ ‘ dual$_{st}$ ‘ $\mathcal{S}$) ∪ (ik$_{est}$ $\mathcal{A}$)) ∪
($\bigcup$(assignment_rhs$_{st}$ ‘ $\mathcal{S}$) ∪ (assignment_rhs$_{est}$ $\mathcal{A}$)))"
shows "Ana_invar_subst (
($\bigcup$(ik$_{st}$ ‘ dual$_{st}$ ‘ $\mathcal{S}$') ∪ (ik$_{est}$ $\mathcal{A}$')) ∪
($\bigcup$(assignment_rhs$_{st}$ ‘ $\mathcal{S}$') ∪ (assignment_rhs$_{est}$ $\mathcal{A}$')))"
⟨*proof*⟩ **lemma** *pts_symbolic_c_preserves_constr_disj_vars:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')" "wf$_{sts}$' $\mathcal{S}$ $\mathcal{A}$" "fv$_{est}$ $\mathcal{A}$ ∩ bvars$_{est}$ $\mathcal{A}$ = {}"
shows "fv$_{est}$ $\mathcal{A}$' ∩ bvars$_{est}$ $\mathcal{A}$' = {}"
⟨*proof*⟩

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"
⟨*proof*⟩ **lemma** *decomps$_{est}$_pts_symbolic_c:*
assumes "D ∈ decomps$_{est}$ (ik$_{est}$ A) (assignment_rhs$_{est}$ A) $\mathcal{I}$"
shows "(S,A) $\Rightarrow^{\bullet}{}_c{}^*$ (S,A@D)"
⟨*proof*⟩ **lemma** *pts_symbolic_to_pts_symbolic_c:*
assumes "($\mathcal{S}$,to_st (decomp_rm$_{est}$ $\mathcal{A}_d$)) $\Rightarrow^{\bullet *}$ ($\mathcal{S}$',$\mathcal{A}$')" "sem$_{est}$_d {} $\mathcal{I}$ (to_est $\mathcal{A}$')" "sem$_{est}$_c {} $\mathcal{I}$ $\mathcal{A}_d$"
and *wf:* "wf$_{sts}$' $\mathcal{S}$ (decomp_rm$_{est}$ $\mathcal{A}_d$)" "wf$_{est}$ {} $\mathcal{A}_d$"
and *tar:* "Ana_invar_subst (($\bigcup$(ik$_{st}$‘ dual$_{st}$‘ $\mathcal{S}$) ∪ (ik$_{est}$ $\mathcal{A}_d$))
∪ ($\bigcup$(assignment_rhs$_{st}$‘ $\mathcal{S}$) ∪ (assignment_rhs$_{est}$ $\mathcal{A}_d$)))"
and *wa:* "well_analyzed $\mathcal{A}_d$"
and *$\mathcal{I}$:* "interpretation$_{subst}$ $\mathcal{I}$"
shows "∃$\mathcal{A}_d$'. $\mathcal{A}$' = to_st (decomp_rm$_{est}$ $\mathcal{A}_d$') ∧ ($\mathcal{S}$,$\mathcal{A}_d$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}_d$') ∧ sem$_{est}$_c {} $\mathcal{I}$ $\mathcal{A}_d$'"
⟨*proof*⟩ **lemma** *pts_symbolic_c_to_pts_symbolic:*
assumes "($\mathcal{S}$,$\mathcal{A}$) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$',$\mathcal{A}$')" "sem$_{est}$_c {} $\mathcal{I}$ $\mathcal{A}$'"
shows "($\mathcal{S}$,to_st (decomp_rm$_{est}$ $\mathcal{A}$)) $\Rightarrow^{\bullet *}$ ($\mathcal{S}$',to_st (decomp_rm$_{est}$ $\mathcal{A}$'))"
"sem$_{est}$_d {} $\mathcal{I}$ (decomp_rm$_{est}$ $\mathcal{A}$')"
⟨*proof*⟩ **lemma** *pts_symbolic_to_pts_symbolic_c_from_initial:*
assumes "($\mathcal{S}_0$,[]) $\Rightarrow^{\bullet *}$ ($\mathcal{S}$,$\mathcal{A}$)" "$\mathcal{I}$ ⊨ ⟨$\mathcal{A}$⟩" "wf$_{sts}$' $\mathcal{S}_0$ []"
and "Ana_invar_subst ($\bigcup$(ik$_{st}$ ‘ dual$_{st}$ ‘ $\mathcal{S}_0$) ∪ $\bigcup$(assignment_rhs$_{st}$ ‘ $\mathcal{S}_0$))" "interpretation$_{subst}$ $\mathcal{I}$"
shows "∃$\mathcal{A}_d$. $\mathcal{A}$ = to_st (decomp_rm$_{est}$ $\mathcal{A}_d$) ∧ ($\mathcal{S}_0$,[]) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$,$\mathcal{A}_d$) ∧ ($\mathcal{I}$ $\models_c$ ⟨to_st $\mathcal{A}_d$⟩)"
⟨*proof*⟩ **lemma** *pts_symbolic_c_to_pts_symbolic_from_initial:*
assumes "($\mathcal{S}_0$,[]) $\Rightarrow^{\bullet}{}_c{}^*$ ($\mathcal{S}$,$\mathcal{A}$)" "$\mathcal{I}$ $\models_c$ ⟨to_st $\mathcal{A}$⟩"
shows "($\mathcal{S}_0$,[]) $\Rightarrow^{\bullet *}$ ($\mathcal{S}$,to_st (decomp_rm$_{est}$ $\mathcal{A}$))" "$\mathcal{I}$ ⊨ ⟨to_st (decomp_rm$_{est}$ $\mathcal{A}$)⟩"
⟨*proof*⟩ **lemma** *to_st_trms_wf:*
assumes "wf$_{trms}$ (trms$_{est}$ A)"
shows "wf$_{trms}$ (trms$_{st}$ (to_st A))"

⟨proof⟩ **lemma** to_st_trms_SMP_subset: "$trms_{st}$ (to_st A) ⊆ SMP ($trms_{est}$ A)"
⟨proof⟩ **lemma** $to\_st\_trms\_tfr_{set}$:
assumes "tfr_{set} ($trms_{est}$ A)"
shows "tfr_{set} ($trms_{st}$ (to_st A))"
⟨proof⟩

theorem wt_attack_if_tfr_attack_pts:
assumes "wf_{sts} $\mathcal{S}_0$" "tfr_{set} ($\bigcup$($trms_{st}$ ‘ $\mathcal{S}_0$))" "wf_{trms} ($\bigcup$($trms_{st}$ ‘ $\mathcal{S}_0$))" "$\forall S \in \mathcal{S}_0$. list_all tfr_{stp} S"
and "Ana_invar_subst ($\bigcup$(ik_{st} ‘ $dual_{st}$ ‘ $\mathcal{S}_0$) ∪ $\bigcup$($assignment\_rhs_{st}$ ‘ $\mathcal{S}_0$))"
and "($\mathcal{S}_0$,[]) $\Rightarrow^{\bullet *}$ ($\mathcal{S}$,$\mathcal{A}$)" "$interpretation_{subst}$ $\mathcal{I}$" "$\mathcal{I} \models \langle \mathcal{A}$, Var⟩"
shows "$\exists \mathcal{I}_\tau$. $interpretation_{subst}$ $\mathcal{I}_\tau$ ∧ ($\mathcal{I}_\tau \models \langle \mathcal{A}$, Var⟩) ∧ wt_{subst} $\mathcal{I}_\tau$ ∧ wf_{trms} (subst_range $\mathcal{I}_\tau$)"
⟨proof⟩

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} {} $\mathcal{A}$" "fv_{st} $\mathcal{A}$ ∩ $bvars_{st}$ $\mathcal{A}$ = {}" "tfr_{st} $\mathcal{A}$" "wf_{trms} ($trms_{st}$ $\mathcal{A}$)"
and "Ana_invar_subst (ik_{st} $\mathcal{A}$ ∪ $assignment\_rhs_{st}$ $\mathcal{A}$)"
and "$interpretation_{subst}$ $\mathcal{I}$" "$\mathcal{I} \models \langle \mathcal{A} \rangle$"
shows "$\exists \mathcal{I}_\tau$. $interpretation_{subst}$ $\mathcal{I}_\tau$ ∧ ($\mathcal{I}_\tau \models \langle \mathcal{A} \rangle$) ∧ wt_{subst} $\mathcal{I}_\tau$ ∧ wf_{trms} (subst_range $\mathcal{I}_\tau$)"
⟨proof⟩

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 "·sst" 51) where
  "S ·sst ϑ ≡ map (λx. x ·sstp ϑ) S"

fun dbupdsst::"('f,'v) stateful_strand ⇒ ('f,'v) subst ⇒ ('f,'v) dbstate ⇒ ('f,'v) dbstate"
where
  "dbupdsst [] I D = D"
| "dbupdsst (Insert t s#A) I D = dbupdsst A I (insert ((t,s) ·p I) D)"
| "dbupdsst (Delete t s#A) I D = dbupdsst A I (D - {((t,s) ·p I)})"
| "dbupdsst (_#A) I D = dbupdsst 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) ·p I) D)"
| "db'sst (Delete t s#A) I D = db'sst A I (List.removeAll ((t,s) ·p I) D)"
| "db'sst (_#A) I D = db'sst A I D"

definition dbsst where
  "dbsst S I ≡ db'sst S I []"

fun setopssstp where
  "setopssstp (Insert t s) = {(t,s)}"
| "setopssstp (Delete t s) = {(t,s)}"
| "setopssstp (InSet _ t s) = {(t,s)}"
| "setopssstp (NegChecks _ _ F') = set F'"
| "setopssstp _ = {}"
```

The set-operations of a stateful strand

```
definition setopssst where
  "setopssst S ≡ ⋃(setopssstp ` set S)"

fun setops_listsstp where
  "setops_listsstp (Insert t s) = [(t,s)]"
| "setops_listsstp (Delete t s) = [(t,s)]"
| "setops_listsstp (InSet _ t s) = [(t,s)]"
| "setops_listsstp (NegChecks _ _ F') = F'"
| "setops_listsstp _ = []"
```

The set-operations of a stateful strand (list variant)

```
definition setops_listsst where
  "setops_listsst S ≡ remdups (concat (map setops_listsstp S))"
```

4.1.2 Small Lemmata

```
lemma trms_listsst_is_trmssst: "trmssst S = set (trms_listsst S)"
⟨proof⟩

lemma setops_listsst_is_setopssst: "setopssst S = set (setops_listsst S)"
⟨proof⟩

lemma fv_listsstp_is_fvsstp: "fvsstp a = set (fv_listsstp a)"
⟨proof⟩

lemma fv_listsst_is_fvsst: "fvsst S = set (fv_listsst S)"
⟨proof⟩

lemma trmssstp_finite[simp]: "finite (trmssstp x)"
⟨proof⟩

lemma trmssst_finite[simp]: "finite (trmssst S)"
⟨proof⟩
```

lemma vars$_{sstp}$_finite[simp]: "finite (vars$_{sstp}$ x)"
⟨*proof*⟩

lemma vars$_{sst}$_finite[simp]: "finite (vars$_{sst}$ S)"
⟨*proof*⟩

lemma fv$_{sstp}$_finite[simp]: "finite (fv$_{sstp}$ x)"
⟨*proof*⟩

lemma fv$_{sst}$_finite[simp]: "finite (fv$_{sst}$ S)"
⟨*proof*⟩

lemma bvars$_{sstp}$_finite[simp]: "finite (set (bvars$_{sstp}$ x))"
⟨*proof*⟩

lemma bvars$_{sst}$_finite[simp]: "finite (bvars$_{sst}$ S)"
⟨*proof*⟩

lemma subst_sst_nil[simp]: "[] $\cdot_{sst}$ δ = []"
⟨*proof*⟩

lemma db$_{sst}$_nil[simp]: "db$_{sst}$ [] $\mathcal{I}$ = []"
⟨*proof*⟩

lemma ik$_{sst}$_nil[simp]: "ik$_{sst}$ [] = {}"
⟨*proof*⟩

lemma ik$_{sst}$_append[simp]: "ik$_{sst}$ (A@B) = ik$_{sst}$ A ∪ ik$_{sst}$ B"
⟨*proof*⟩

lemma ik$_{sst}$_subst: "ik$_{sst}$ (A $\cdot_{sst}$ δ) = ik$_{sst}$ A $\cdot_{set}$ δ"
⟨*proof*⟩

lemma db$_{sst}$_set_is_dbupd$_{sst}$: "set (db'$_{sst}$ A I D) = dbupd$_{sst}$ A I (set D)" (**is** "?A = ?B")
⟨*proof*⟩

lemma dbupd$_{sst}$_no_upd:
assumes "∀ a ∈ set A. ¬is_Insert a ∧ ¬is_Delete a"
shows "dbupd$_{sst}$ A I D = D"
⟨*proof*⟩

lemma db$_{sst}$_no_upd:
assumes "∀ a ∈ set A. ¬is_Insert a ∧ ¬is_Delete a"
shows "db'$_{sst}$ A I D = D"
⟨*proof*⟩

lemma db$_{sst}$_no_upd_append:
assumes "∀ b ∈ set B. ¬is_Insert b ∧ ¬is_Delete b"
shows "db'$_{sst}$ A = db'$_{sst}$ (A@B)"
⟨*proof*⟩

lemma db$_{sst}$_append:
"db'$_{sst}$ (A@B) I D = db'$_{sst}$ B I (db'$_{sst}$ A I D)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
  ⟨proof⟩
```

```
lemma db_sst_filter:
  "db'_sst A I D = db'_sst (filter is_Update A) I D"
⟨proof⟩
```

```
lemma subst_sst_cons: "a#A ·_sst δ = (a ·_sstp δ)#(A ·_sst δ)"
⟨proof⟩
```

```
lemma subst_sst_snoc: "A@[a] ·_sst δ = (A ·_sst δ)@[a ·_sstp δ]"
⟨proof⟩
```

```
lemma subst_sst_append[simp]: "A@B ·_sst δ = (A ·_sst δ)@(B ·_sst δ)"
⟨proof⟩
```

```
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)"
⟨proof⟩
```

```
lemma sst_vars_disj_cons[simp]: "fv_sst (a#A) ∩ bvars_sst (a#A) = {} ⟹ fv_sst A ∩ bvars_sst A = {}"
⟨proof⟩
```

```
lemma fv_sst_cons_subset[simp]: "fv_sst A ⊆ fv_sst (a#A)"
⟨proof⟩
```

```
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"
⟨proof⟩
```

```
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⟩
```

```
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⟩

lemma bvars$_{sst}$_cons_subset: "bvars$_{sst}$ A ⊆ bvars$_{sst}$ (a#A)"
⟨proof⟩

lemma bvars$_{sstp}$_subst: "bvars$_{sstp}$ (a ·$_{sstp}$ δ) = bvars$_{sstp}$ a"
⟨proof⟩

lemma bvars$_{sst}$_subst: "bvars$_{sst}$ (A ·$_{sst}$ δ) = bvars$_{sst}$ A"
⟨proof⟩

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"
```
⟨proof⟩

lemma bvars$_{sstp}$_NegChecks: "¬is_NegChecks a ⟹ bvars$_{sstp}$ a = []"
⟨proof⟩

lemma bvars$_{sst}$_NegChecks: "bvars$_{sst}$ A = bvars$_{sst}$ (filter is_NegChecks A)"
⟨proof⟩

lemma vars$_{sst}$_append[simp]: "vars$_{sst}$ (A@B) = vars$_{sst}$ A ∪ vars$_{sst}$ B"
⟨proof⟩

lemma vars$_{sst}$_Nil[simp]: "vars$_{sst}$ [] = {}"
⟨proof⟩

lemma vars$_{sst}$_Cons: "vars$_{sst}$ (a#A) = vars$_{sstp}$ a ∪ vars$_{sst}$ A"
⟨proof⟩

lemma fv$_{sst}$_Cons: "fv$_{sst}$ (a#A) = fv$_{sstp}$ a ∪ fv$_{sst}$ A"
⟨proof⟩

lemma bvars$_{sst}$_Cons: "bvars$_{sst}$ (a#A) = set (bvars$_{sstp}$ a) ∪ bvars$_{sst}$ A"
⟨proof⟩

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"
```
⟨proof⟩

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⟩

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⟩

lemma $vars_{sstp}$_NegCheck[simp]:
"$vars_{sstp}$ ($\forall X\langle\vee\neq$: F $\vee\notin$: G$\rangle$) = set X $\cup$ fv_{pairs} F $\cup$ fv_{pairs} G"
⟨proof⟩

lemma $bvars_{sstp}$_NegCheck[simp]:
"$bvars_{sstp}$ ($\forall X\langle\vee\neq$: F $\vee\notin$: G$\rangle$) = X"
"set ($bvars_{sstp}$ ($\forall$ []$\langle\vee\neq$: F $\vee\notin$: G$\rangle$)) = {}"
⟨proof⟩

lemma fv_{sstp}_NegCheck[simp]:
"fv_{sstp} ($\forall X\langle\vee\neq$: F $\vee\notin$: G$\rangle$) = fv_{pairs} F $\cup$ fv_{pairs} G - set X"
"fv_{sstp} ($\forall$ []$\langle\vee\neq$: F $\vee\notin$: G$\rangle$) = fv_{pairs} F $\cup$ fv_{pairs} G"
"fv_{sstp} (⟨t != s⟩) = fv t $\cup$ fv s"
"fv_{sstp} (⟨t not in s⟩) = fv t $\cup$ fv s"
⟨proof⟩

lemma fv_{sst}_append[simp]: "fv_{sst} (A@B) = fv_{sst} A $\cup$ fv_{sst} B"
⟨proof⟩

lemma $bvars_{sst}$_append[simp]: "$bvars_{sst}$ (A@B) = $bvars_{sst}$ A $\cup$ $bvars_{sst}$ B"
⟨proof⟩

lemma fv_{sstp}_is_subterm_$trms_{sstp}$:
assumes "x $\in$ fv_{sstp} a"
shows "Var x $\in$ $subterms_{set}$ ($trms_{sstp}$ a)"
⟨proof⟩

lemma fv_{sst}_is_subterm_$trms_{sst}$: "x $\in$ fv_{sst} A $\Longrightarrow$ Var x $\in$ $subterms_{set}$ ($trms_{sst}$ A)"
⟨proof⟩

lemma var_subterm_$trms_{sstp}$_is_$vars_{sstp}$:
assumes "Var x $\in$ $subterms_{set}$ ($trms_{sstp}$ a)"
shows "x $\in$ $vars_{sstp}$ a"
⟨proof⟩

lemma var_subterm_$trms_{sst}$_is_$vars_{sst}$: "Var x $\in$ $subterms_{set}$ ($trms_{sst}$ A) $\Longrightarrow$ x $\in$ $vars_{sst}$ A"
⟨proof⟩

lemma var_$trms_{sst}$_is_$vars_{sst}$: "Var x $\in$ $trms_{sst}$ A $\Longrightarrow$ x $\in$ $vars_{sst}$ A"
⟨proof⟩

lemma ik_{sst}_$trms_{sst}$_subset: "ik_{sst} A $\subseteq$ $trms_{sst}$ A"
⟨proof⟩

lemma var_subterm_ik_{sst}_is_$vars_{sst}$: "Var x $\in$ $subterms_{set}$ (ik_{sst} A) $\Longrightarrow$ x $\in$ $vars_{sst}$ A"
⟨proof⟩

lemma var_subterm_ik_{sst}_is_fv_{sst}:
assumes "Var x $\in$ $subterms_{set}$ (ik_{sst} A)"
shows "x $\in$ fv_{sst} A"
⟨proof⟩

lemma fv_ik_{sst}_is_fv_{sst}:
assumes "x $\in$ fv_{set} (ik_{sst} A)"
shows "x $\in$ fv_{sst} A"
⟨proof⟩

lemma fv_$trms_{sst}$_subset:
"fv_{set} ($trms_{sst}$ S) $\subseteq$ $vars_{sst}$ S"
"fv_{sst} S $\subseteq$ fv_{set} ($trms_{sst}$ S)"
⟨proof⟩

lemma *fv_ik_subset_fv_sst'[simp]: "fv$_{set}$ (ik$_{sst}$ S) ⊆ fv$_{sst}$ S"*
⟨*proof*⟩

lemma *fv_ik_subset_vars_sst'[simp]: "fv$_{set}$ (ik$_{sst}$ S) ⊆ vars$_{sst}$ S"*
⟨*proof*⟩

lemma *ik$_{sst}$_var_is_fv: "Var x ∈ subterms$_{set}$ (ik$_{sst}$ A) ⟹ x ∈ fv$_{sst}$ A"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *vars$_{sst}$_subst_cases:*
  assumes *"x ∈ vars$_{sst}$ (S ·$_{sst}$ ϑ)"*
  shows *"x ∈ vars$_{sst}$ S ∨ x ∈ fv$_{set}$ (ϑ ‘ vars$_{sst}$ S)"*
  ⟨*proof*⟩

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'"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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)"
⟨*proof*⟩

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⟩

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⟩

lemma *trms$_{sst}$_nil[simp]:*

```
"trmssst [] = {}"
```
⟨proof⟩

lemma *trms$_{sst}$_mono:*

```
"set M ⊆ set N ⟹ trmssst M ⊆ trmssst N"
```
⟨proof⟩

lemma *trms$_{sst}$_in:*
assumes "t ∈ trms$_{sst}$ S"
shows "∃a ∈ set S. t ∈ trms$_{sstp}$ a"
⟨proof⟩

lemma *trms$_{sst}$_cons:* "trms$_{sst}$ (a#A) = trms$_{sstp}$ a ∪ trms$_{sst}$ A"
⟨proof⟩

lemma *trms$_{sst}$_append[simp]:* "trms$_{sst}$ (A@B) = trms$_{sst}$ A ∪ trms$_{sst}$ B"
⟨proof⟩

lemma *trms$_{sstp}$_subst:*
assumes "set (bvars$_{sstp}$ a) ∩ subst_domain ϑ = {}"
shows "trms$_{sstp}$ (a ·$_{sstp}$ ϑ) = trms$_{sstp}$ a ·$_{set}$ ϑ"
⟨proof⟩

lemma *trms$_{sstp}$_subst':*
assumes "¬is_NegChecks a"
shows "trms$_{sstp}$ (a ·$_{sstp}$ ϑ) = trms$_{sstp}$ a ·$_{set}$ ϑ"
⟨proof⟩

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⟩

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⟩

lemma *trms$_{sst}$_subst:*
assumes "bvars$_{sst}$ S ∩ subst_domain ϑ = {}"
shows "trms$_{sst}$ (S ·$_{sst}$ ϑ) = trms$_{sst}$ S ·$_{set}$ ϑ"
⟨proof⟩

lemma *trms$_{sst}$_subst_cons:*

```
"trmssst (a#A ·sst δ) = trmssstp (a ·sstp δ) ∪ trmssst (A ·sst δ)"
```

⟨proof⟩

lemma (**in** intruder_model) wf$_{trms}$_trms$_{sstp}$_subst:
  assumes "wf$_{trms}$ (trms$_{sstp}$ a $\cdot_{set}$ δ)"
  shows "wf$_{trms}$ (trms$_{sstp}$ (a $\cdot_{sstp}$ δ))"
  ⟨proof⟩

lemma trms$_{sst}$_fv_vars$_{sst}$_subset: "t $\in$ trms$_{sst}$ A $\Longrightarrow$ fv t $\subseteq$ vars$_{sst}$ A"
⟨proof⟩

lemma trms$_{sst}$_fv_subst_subset:
  assumes "t $\in$ trms$_{sst}$ S" "subst_domain ϑ $\cap$ bvars$_{sst}$ S = {}"
  shows "fv (t $\cdot$ ϑ) $\subseteq$ vars$_{sst}$ (S $\cdot_{sst}$ ϑ)"
⟨proof⟩

lemma trms$_{sst}$_fv_subst_subset':
  assumes "t $\in$ subterms$_{set}$ (trms$_{sst}$ S)" "fv t $\cap$ bvars$_{sst}$ S = {}" "fv (t $\cdot$ ϑ) $\cap$ bvars$_{sst}$ S = {}"
  shows "fv (t $\cdot$ ϑ) $\subseteq$ fv$_{sst}$ (S $\cdot_{sst}$ ϑ)"
⟨proof⟩

lemma trms$_{sstp}$_funs_term_cases:
  assumes "t $\in$ trms$_{sstp}$ (s $\cdot_{sstp}$ ϑ)" "f $\in$ funs_term t"
  shows "($\exists$u $\in$ trms$_{sstp}$ s. f $\in$ funs_term u) $\vee$ ($\exists$x $\in$ fv$_{sstp}$ s. f $\in$ funs_term (ϑ x))"
  ⟨proof⟩

lemma trms$_{sst}$_funs_term_cases:
  assumes "t $\in$ trms$_{sst}$ (S $\cdot_{sst}$ ϑ)" "f $\in$ funs_term t"
  shows "($\exists$u $\in$ trms$_{sst}$ S. f $\in$ funs_term u) $\vee$ ($\exists$x $\in$ fv$_{sst}$ S. f $\in$ funs_term (ϑ x))"
⟨proof⟩

lemma fv$_{sst}$_is_subterm_trms$_{sst}$_subst:
  assumes "x $\in$ fv$_{sst}$ T"
    and "bvars$_{sst}$ T $\cap$ subst_domain ϑ = {}"
  shows "ϑ x $\in$ subterms$_{set}$ (trms$_{sst}$ (T $\cdot_{sst}$ ϑ))"
⟨proof⟩

lemma fv$_{sst}$_subst_fv_subset:
  assumes "x $\in$ fv$_{sst}$ S" "x $\notin$ bvars$_{sst}$ S" "fv (ϑ x) $\cap$ bvars$_{sst}$ S = {}"
  shows "fv (ϑ x) $\subseteq$ fv$_{sst}$ (S $\cdot_{sst}$ ϑ)"
⟨proof⟩

lemma (**in** intruder_model) wf$_{trms}$_trms$_{sst}$_subst:
  assumes "wf$_{trms}$ (trms$_{sst}$ A $\cdot_{set}$ δ)"
  shows "wf$_{trms}$ (trms$_{sst}$ (A $\cdot_{sst}$ δ))"
  ⟨proof⟩

lemma fv$_{sst}$_subst_obtain_var:
  assumes "x $\in$ fv$_{sst}$ (S $\cdot_{sst}$ δ)"
  shows "$\exists$y $\in$ fv$_{sst}$ S. x $\in$ fv (δ y)"
  ⟨proof⟩

lemma fv$_{sst}$_subst_subset_range_vars_if_subset_domain:
  assumes "fv$_{sst}$ S $\subseteq$ subst_domain σ"
  shows "fv$_{sst}$ (S $\cdot_{sst}$ σ) $\subseteq$ range_vars σ"
⟨proof⟩

lemma fv$_{sst}$_in_fv_trms$_{sst}$: "x $\in$ fv$_{sst}$ S $\Longrightarrow$ x $\in$ fv$_{set}$ (trms$_{sst}$ S)"
⟨proof⟩

lemma stateful_strand_step_subst_comp:
  assumes "range_vars δ $\cap$ set (bvars$_{sstp}$ x) = {}"
  shows "x $\cdot_{sstp}$ δ $\circ_s$ ϑ = (x $\cdot_{sstp}$ δ) $\cdot_{sstp}$ ϑ"
⟨proof⟩

lemma *stateful_strand_subst_comp:*
assumes "range_vars δ $\cap$ bvars$_{sst}$ S = {}"
shows "S $\cdot_{sst}$ δ $\circ_s$ ϑ = (S $\cdot_{sst}$ δ) $\cdot_{sst}$ ϑ"
⟨proof⟩

lemma *subst_apply_bvars_disj_NegChecks:*
assumes "set X $\cap$ subst_domain ϑ = {}"
shows "NegChecks X F G $\cdot_{sstp}$ ϑ = NegChecks X (F $\cdot_{pairs}$ ϑ) (G $\cdot_{pairs}$ ϑ)"
⟨proof⟩

lemma *subst_apply_NegChecks_no_bvars[simp]:*
"$\forall$ []⟨$\vee\neq$: F $\vee\notin$: F'⟩ $\cdot_{sstp}$ ϑ = $\forall$ []⟨$\vee\neq$: (F $\cdot_{pairs}$ ϑ) $\vee\notin$: (F' $\cdot_{pairs}$ ϑ)⟩"
"$\forall$ []⟨$\vee\neq$: [] $\vee\notin$: F'⟩ $\cdot_{sstp}$ ϑ = $\forall$ []⟨$\vee\neq$: [] $\vee\notin$: (F' $\cdot_{pairs}$ ϑ)⟩"
"$\forall$ []⟨$\vee\neq$: F $\vee\notin$: []⟩ $\cdot_{sstp}$ ϑ = $\forall$ []⟨$\vee\neq$: (F $\cdot_{pairs}$ ϑ) $\vee\notin$: []⟩"
"$\forall$ []⟨$\vee\neq$: [] $\vee\notin$: [(t,s)]⟩ $\cdot_{sstp}$ ϑ = $\forall$ []⟨$\vee\neq$: [] $\vee\notin$: ([(t $\cdot$ ϑ,s $\cdot$ ϑ)])⟩" (**is** ?A)
"$\forall$ []⟨$\vee\neq$: [(t,s)] $\vee\notin$: []⟩ $\cdot_{sstp}$ ϑ = $\forall$ []⟨$\vee\neq$: ([(t $\cdot$ ϑ,s $\cdot$ ϑ)]) $\vee\notin$: []⟩" (**is** ?B)
⟨proof⟩

lemma *setops$_{sst}$_mono:*
"set M $\subseteq$ set N $\Longrightarrow$ setops$_{sst}$ M $\subseteq$ setops$_{sst}$ N"
⟨proof⟩

lemma *setops$_{sst}$_nil[simp]:* "setops$_{sst}$ [] = {}"
⟨proof⟩

lemma *setops$_{sst}$_cons[simp]:* "setops$_{sst}$ (a#A) = setops$_{sstp}$ a $\cup$ setops$_{sst}$ A"
⟨proof⟩

lemma *setops$_{sst}$_cons_subset[simp]:* "setops$_{sst}$ A $\subseteq$ setops$_{sst}$ (a#A)"
⟨proof⟩

lemma *setops$_{sst}$_append:* "setops$_{sst}$ (A@B) = setops$_{sst}$ A $\cup$ setops$_{sst}$ B"
⟨proof⟩

lemma *setops$_{sstp}$_member_iff:*
"(t,s) $\in$ setops$_{sstp}$ x $\longleftrightarrow$
(x = Insert t s $\vee$ x = Delete t s $\vee$ ($\exists$ac. x = InSet ac t s) $\vee$
($\exists$X F F'. x = NegChecks X F F' $\wedge$ (t,s) $\in$ set F'))"
⟨proof⟩

lemma *setops$_{sst}$_member_iff:*
"(t,s) $\in$ setops$_{sst}$ A $\longleftrightarrow$
(Insert t s $\in$ set A $\vee$ Delete t s $\in$ set A $\vee$ ($\exists$ac. InSet ac t s $\in$ set A) $\vee$
($\exists$X F F'. NegChecks X F F' $\in$ set A $\wedge$ (t,s) $\in$ set F'))"
(**is** "?P $\longleftrightarrow$?Q")
⟨proof⟩

lemma *setops$_{sstp}$_subst:*
assumes "set (bvars$_{sstp}$ a) $\cap$ subst_domain ϑ = {}"
shows "setops$_{sstp}$ (a $\cdot_{sstp}$ ϑ) = setops$_{sstp}$ a $\cdot_{pset}$ ϑ"
⟨proof⟩

lemma *setops$_{sstp}$_subst':*
assumes "$\neg$is_NegChecks a"
shows "setops$_{sstp}$ (a $\cdot_{sstp}$ ϑ) = setops$_{sstp}$ a $\cdot_{pset}$ ϑ"
⟨proof⟩

lemma *setops$_{sstp}$_subst'':*
fixes t::"('a,'b) term $\times$ ('a,'b) term" **and** δ::"('a,'b) subst"
assumes t: "t $\in$ setops$_{sstp}$ (b $\cdot_{sstp}$ δ)"
shows "$\exists$s $\in$ setops$_{sstp}$ b. t = s $\cdot_p$ rm_vars (set (bvars$_{sstp}$ b)) δ"
⟨proof⟩

lemma *setops$_{sst}$_subst:*
  assumes "bvars$_{sst}$ S ∩ subst_domain ϑ = {}"
  shows "setops$_{sst}$ (S $\cdot_{sst}$ ϑ) = setops$_{sst}$ S $\cdot_{pset}$ ϑ"
⟨*proof*⟩

lemma *setops$_{sst}$_subst':*
  fixes p::"('a,'b) term × ('a,'b) term" **and** δ::"('a,'b) subst"
  assumes "p ∈ setops$_{sst}$ (S $\cdot_{sst}$ δ)"
  shows "∃s ∈ setops$_{sst}$ S. ∃X. set X ⊆ bvars$_{sst}$ S ∧ p = s $\cdot_p$ rm_vars (set X) δ"
⟨*proof*⟩

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 $\circ_s$ ℐ) ≠ snd f · (δ $\circ_s$ ℐ)) F ∨
           list_ex (λf. f $\cdot_p$ (δ $\circ_s$ 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) $\cdot_p$ ℐ) D; S⟧$_s$ ℐ)"
| "⟦M; D; Delete t s#S⟧$_s$ = (λℐ. ⟦M; D - {(t,s) $\cdot_p$ ℐ}; S⟧$_s$ ℐ)"
| "⟦M; D; InSet _ t s#S⟧$_s$ = (λℐ. (t,s) $\cdot_p$ 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** "$\models_s$" 91) **where** "ℐ $\models_s$ 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"
⟨*proof*⟩

lemma *strand_sem_ik_mono_stateful:*
  "⟦M; D; A⟧$_s$ ℐ ⟹ ⟦M ∪ M'; D; A⟧$_s$ ℐ"
⟨*proof*⟩

lemma *strand_sem_append_stateful:*
  "⟦M; D; A@B⟧$_s$ ℐ ⟷ ⟦M; D; A⟧$_s$ ℐ ∧ ⟦M ∪ (ik$_{sst}$ A $\cdot_{set}$ ℐ); dbupd$_{sst}$ A ℐ D; B⟧$_s$ ℐ"
  (**is** "?P ⟷ ?Q ∧ ?R")
⟨*proof*⟩

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⟩

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 $\cdot_p$ (δ $\circ_s$ ℐ) ∉ D - D'"
    and "negchecks_model ℐ D' X F F'"
  shows "negchecks_model ℐ D X F F'"
⟨proof⟩

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 $\circ_s$ ϑ) D X F F' ⟷ negchecks_model ϑ D X (F $\cdot_{pairs}$ δ) (F' $\cdot_{pairs}$ δ)"
⟨proof⟩

lemma *strand_sem_subst_stateful:*
  fixes δ::"('fun,'var) subst"
  assumes "(subst_domain δ ∪ range_vars δ) ∩ bvars$_{sst}$ S = {}"
  shows "⟦M; D; S⟧$_s$ (δ $\circ_s$ ϑ) ⟷ ⟦M; D; S $\cdot_{sst}$ δ⟧$_s$ ϑ"
⟨proof⟩

end

4.1.4 Well-Formedness Lemmata

lemma *wfvarsocc$_{sst}$_subset_wfrestrictedvars$_{sst}$[simp]:*
  "wfvarsoccs$_{sst}$ S ⊆ wfrestrictedvars$_{sst}$ S"
⟨proof⟩

lemma *wfvarsoccs$_{sst}$_append:* "wfvarsoccs$_{sst}$ (S@S') = wfvarsoccs$_{sst}$ S ∪ wfvarsoccs$_{sst}$ S'"
⟨proof⟩

lemma *wfrestrictedvars$_{sst}$_union[simp]:*
  "wfrestrictedvars$_{sst}$ (S@T) = wfrestrictedvars$_{sst}$ S ∪ wfrestrictedvars$_{sst}$ T"
⟨proof⟩

lemma *wfrestrictedvars$_{sst}$_singleton:*
  "wfrestrictedvars$_{sst}$ [s] = wfrestrictedvars$_{sstp}$ s"
⟨proof⟩

lemma *wf$_{sst}$_prefix[dest]:* "wf'$_{sst}$ V (S@S') ⟹ wf'$_{sst}$ V S"
⟨proof⟩

lemma *wf$_{sst}$_vars_mono:* "wf'$_{sst}$ V S ⟹ wf'$_{sst}$ (V ∪ W) S"
⟨proof⟩

lemma *wf$_{sst}$I[intro]:* "wfrestrictedvars$_{sst}$ S ⊆ V ⟹ wf'$_{sst}$ V S"
⟨proof⟩

lemma *wf$_{sst}$I'[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"
```
⟨proof⟩

lemma *wf$_{sst}$_append_exec:* "wf'$_{sst}$ V (S@S') ⟹ wf'$_{sst}$ (V ∪ wfvarsoccs$_{sst}$ S) S'"
⟨proof⟩

lemma *wf$_{sst}$_append:*
  "wf'$_{sst}$ X S $\Longrightarrow$ wf'$_{sst}$ Y T $\Longrightarrow$ wf'$_{sst}$ (X $\cup$ Y) (S@T)"
$\langle$*proof*$\rangle$

lemma *wf$_{sst}$_append_suffix:*
  "wf'$_{sst}$ V S $\Longrightarrow$ wfrestrictedvars$_{sst}$ S' $\subseteq$ wfrestrictedvars$_{sst}$ S $\cup$ V $\Longrightarrow$ wf'$_{sst}$ V (S@S')"
$\langle$*proof*$\rangle$

lemma *wf$_{sst}$_append_suffix':*
  assumes "wf'$_{sst}$ V S"
    and "$\bigcup$((λx. case x of
            Receive t ⇒ fv t
          | Equality Assign _ t' ⇒ fv t'
          | Insert t t' ⇒ fv t $\cup$ fv t'
          | _ ⇒ {}) ` set S') $\subseteq$ wfvarsoccs$_{sst}$ S $\cup$ V"
  shows "wf'$_{sst}$ V (S@S')"
$\langle$*proof*$\rangle$

lemma *wf$_{sst}$_subst_apply:*
  "wf'$_{sst}$ V S $\Longrightarrow$ wf'$_{sst}$ (fv$_{set}$ (δ ` V)) (S $\cdot_{sst}$ δ)"
$\langle$*proof*$\rangle$

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': "$\bigwedge$f T δ K M. Ana (Fun f T) = (K,M) $\Longrightarrow$ Ana (Fun f T · δ) = (K $\cdot_{list}$ δ,M $\cdot_{list}$ δ)"*
begin

lemma *Ana_invar_subst'[simp]: "Ana_invar_subst $\mathcal{S}$"*
$\langle$*proof*$\rangle$

definition *pair* **where**
  "pair d ≡ case d of (t,t') ⇒ Fun Pair [t,t']"

fun *tr$_{pairs}$::*
  "(('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) (trpairs F' D))) (tr A D)"
```

Type-flaw resistance of stateful constraint steps

```
fun tfrsstp where
  "tfrsstp (Equality _ t t') = ((∃δ. Unifier δ t t') ⟶ Γ t = Γ t')"
| "tfrsstp (NegChecks X F F') = (
    (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)))"
| "tfrsstp _ = True"
```

Type-flaw resistance of stateful constraints

```
definition tfrsst where "tfrsst S ≡ tfrset (trmssst S ∪ pair ` setopssst S) ∧ list_all tfrsstp S"
```

4.2.1 Small Lemmata

```
lemma pair_in_pair_image_iff:
  "pair (s,t) ∈ pair ` P ⟷ (s,t) ∈ P"
⟨proof⟩

lemma subst_apply_pairs_pair_image_subst:
  "pair ` set (F ·pairs ϑ) = pair ` set F ·set ϑ"
⟨proof⟩

lemma Ana_subst_subterms_cases:
  fixes ϑ::"('fun,'var) subst"
  assumes t: "t ∈ subtermsset (M ·set ϑ)"
    and s: "s ∈ set (snd (Ana t))"
  shows "(∃u ∈ subtermsset M. t = u · ϑ ∧ s ∈ set (snd (Ana u)) ·set ϑ) ∨ (∃x ∈ fvset M. t ⊑ ϑ x)"
⟨proof⟩

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⟩

lemma fun_pair_eq[dest]: "pair d = pair d' ⟹ d = d'"
⟨proof⟩

lemma fun_pair_subst: "pair d · δ = pair (d ·p δ)"
⟨proof⟩

lemma fun_pair_subst_set: "pair ` M ·set δ = pair ` (M ·pset δ)"
```

⟨*proof*⟩

lemma *fun_pair_eq_subst: "pair d · δ = pair d' · ϑ ⟷ d $\cdot_p$ δ = d' $\cdot_p$ ϑ"*
⟨*proof*⟩

lemma *setops$_{sst}$_pair_image_cons[simp]:*
  "pair ‘ setops$_{sst}$ (x#S) = pair ‘ setops$_{sstp}$ x ∪ pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (send⟨t⟩#S) = pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (receive⟨t⟩#S) = pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (⟨ac: t ≐ t'⟩#S) = pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (insert⟨t,s⟩#S) = {pair (t,s)} ∪ pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (delete⟨t,s⟩#S) = {pair (t,s)} ∪ pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (⟨ac: t ∈ s⟩#S) = {pair (t,s)} ∪ pair ‘ setops$_{sst}$ S"
  "pair ‘ setops$_{sst}$ (∀X⟨∨≠: F ∨∉: G⟩#S) = pair ‘ set G ∪ pair ‘ setops$_{sst}$ S"
⟨*proof*⟩

lemma *setops$_{sst}$_pair_image_subst_cons[simp]:*
  "pair ‘ setops$_{sst}$ (x#S $\cdot_{sst}$ ϑ) = pair ‘ setops$_{sstp}$ (x $\cdot_{sstp}$ ϑ) ∪ pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (send⟨t⟩#S $\cdot_{sst}$ ϑ) = pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (receive⟨t⟩#S $\cdot_{sst}$ ϑ) = pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (⟨ac: t ≐ t'⟩#S $\cdot_{sst}$ ϑ) = pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (insert⟨t,s⟩#S $\cdot_{sst}$ ϑ) = {pair (t,s) · ϑ} ∪ pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (delete⟨t,s⟩#S $\cdot_{sst}$ ϑ) = {pair (t,s) · ϑ} ∪ pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (⟨ac: t ∈ s⟩#S $\cdot_{sst}$ ϑ) = {pair (t,s) · ϑ} ∪ pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
  "pair ‘ setops$_{sst}$ (∀X⟨∨≠: F ∨∉: G⟩#S $\cdot_{sst}$ ϑ) =
    pair ‘ set (G $\cdot_{pairs}$ rm_vars (set X) ϑ) ∪ pair ‘ setops$_{sst}$ (S $\cdot_{sst}$ ϑ)"
⟨*proof*⟩

lemma *setops$_{sst}$_are_pairs: "t ∈ pair ‘ setops$_{sst}$ A ⟹ ∃s s'. t = pair (s,s')"*
⟨*proof*⟩

lemma *fun_pair_wf$_{trm}$: "wf$_{trm}$ t ⟹ wf$_{trm}$ t' ⟹ wf$_{trm}$ (pair (t,t'))"*
⟨*proof*⟩

lemma *wf$_{trms}$_pairs: "wf$_{trms}$ (trms$_{pairs}$ F) ⟹ wf$_{trms}$ (pair ‘ set F)"*
⟨*proof*⟩

lemma *tfr$_{sst}$_Nil[simp]: "tfr$_{sst}$ []"*
⟨*proof*⟩

lemma *tfr$_{sst}$_append: "tfr$_{sst}$ (A@B) ⟹ tfr$_{sst}$ A"*
⟨*proof*⟩

lemma *tfr$_{sst}$_append': "tfr$_{sst}$ (A@B) ⟹ tfr$_{sst}$ B"*
⟨*proof*⟩

lemma *tfr$_{sst}$_cons: "tfr$_{sst}$ (a#A) ⟹ tfr$_{sst}$ A"*
⟨*proof*⟩

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 $\cdot_{sstp}$ ϑ)"*
⟨*proof*⟩

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 $\cdot_{sst}$ ϑ)"*
⟨*proof*⟩

lemma *tr$_{pairs}$_empty_case:*
  assumes *"tr$_{pairs}$ F D = []"*

shows "D = []" "F ≠ []"
⟨*proof*⟩

lemma tr_{pairs}_elem_length_eq:
assumes "G ∈ set (tr_{pairs} F D)"
shows "length G = length F"
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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)"
⟨*proof*⟩

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*⟩

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*⟩

lemma tr_{pairs}_trms_subset:
"G ∈ set (tr_{pairs} F D) ⟹ $trms_{pairs}$ G ⊆ pair ‘ set F ∪ pair ‘ set D"
⟨*proof*⟩

lemma tr_{pairs}_trms_subset':
"⋃($trms_{pairs}$ ‘ set (tr_{pairs} F D)) ⊆ pair ‘ set F ∪ pair ‘ set D"
⟨*proof*⟩

lemma tr_trms_subset:
"A' ∈ set (tr A D) ⟹ $trms_{st}$ A' ⊆ $trms_{sst}$ A ∪ pair ‘ $setops_{sst}$ A ∪ pair ‘ set D"
⟨*proof*⟩

lemma tr_{pairs}_vars_subset:
"G ∈ set (tr_{pairs} F D) ⟹ fv_{pairs} G ⊆ fv_{pairs} F ∪ fv_{pairs} D"
⟨*proof*⟩

lemma tr_{pairs}_vars_subset': "⋃(fv_{pairs} ‘ set (tr_{pairs} F D)) ⊆ fv_{pairs} F ∪ fv_{pairs} D"
⟨*proof*⟩

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*⟩

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' = {}"

⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *wf_fun_pair_negchecks_map:*
  assumes *"wf$_{st}$ X A"*
  shows *"wf$_{st}$ X (map (λG. ∀Y⟨∨≠: (F@G)⟩$_{st}$) M@A)"*
⟨*proof*⟩

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)) $\doteq$ (pair d)⟩$_{st}$) Di)@*
       (map (λd. ∀[]⟨∨≠: [(pair (t,s), pair d)]⟩$_{st}$) [d←D. d ∉ set Di])@A)"
⟨*proof*⟩

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"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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*⟩

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")*
⟨*proof*⟩

lemma *setops_subterm_trms:*
  assumes *t: "t ∈ pair ‘ setops$_{sst}$ S"*
    and *s: "s ⊏ t"*
  shows *"s ∈ subterms$_{set}$ (trms$_{sst}$ S)"*
⟨*proof*⟩

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*⟩

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*⟩

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⟩
```

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⟩ lemma tr_wf_trms:
  assumes "A' ∈ set (tr A [])" "wf_trms (trms_sst A)"
  shows "wf_trms (trms_st A')"
⟨proof⟩

lemma tr_wf:
  assumes "A' ∈ set (tr A [])"
    and "wf_sst A"
    and "wf_trms (trms_sst A)"
  shows "wf_st {} A'"
    and "wf_trms (trms_st A')"
    and "fv_st A' ∩ bvars_st A' = {}"
⟨proof⟩ lemma tr_tfr_sstp:
  assumes "A' ∈ set (tr A D)" "list_all tfr_sstp A"
  and "fv_sst A ∩ bvars_sst A = {}" (is "?P0 A D")
  and "∀(t,s) ∈ set D. (fv t ∪ fv s) ∩ bvars_sst A = {}" (is "?P1 A D")
  and "∀t ∈ pair ` setops_sst A ∪ pair ` set D. ∀t' ∈ pair ` setops_sst A ∪ pair ` set D.
          (∃δ. Unifier δ t t') ⟶ Γ t = Γ t'" (is "?P3 A D")
  shows "list_all tfr_stp A'"
⟨proof⟩

lemma tr_tfr:
  assumes "A' ∈ set (tr A [])" and "tfr_sst A" and "fv_sst A ∩ bvars_sst A = {}"
  shows "tfr_st A'"
⟨proof⟩ lemma fun_pair_ineqs:
  assumes "d ·_p δ ·_p ϑ ≠ d' ·_p ℐ"
  shows "pair d · δ · ϑ ≠ pair d' · ℐ"
⟨proof⟩ 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⟩ 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⟩ 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⟩ 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⟩

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⟩

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⟩

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⟩

lemma tr_sem_equiv:
  assumes "fv_sst A ∩ bvars_sst A = {}" and "interpretation_subst ℐ"
  shows "ℐ ⊨_s A ⟷ (∃A' ∈ set (tr A []). (ℐ ⊨ ⟨A'⟩))"
⟨proof⟩

theorem stateful_typing_result:
  assumes "wf_sst 𝒜"
    and "tfr_sst 𝒜"
    and "wf_trms (trms_sst 𝒜)"
    and "interpretation_subst ℐ"
    and "ℐ ⊨_s 𝒜"
  obtains ℐ_τ
    where "interpretation_subst ℐ_τ"
    and "ℐ_τ ⊨_s 𝒜"
    and "wt_subst ℐ_τ"
    and "wf_trms (subst_range ℐ_τ)"
⟨proof⟩

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'*
⟨*proof*⟩

lemma *pair_code:*
"pair d = pair' Pair d"
⟨*proof*⟩

lemma *tfr$_{sstp}$_is_comp_tfr$_{sstp}$*: "tfr$_{sstp}$ a = comp_tfr$_{sstp}$ Γ Pair a"
⟨*proof*⟩

lemma *tfr$_{sst}$_if_comp_tfr$_{sst}$*:
assumes "comp_tfr$_{sst}$ arity Ana Γ Pair M S"
shows "tfr$_{sst}$ S"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨proof⟩

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)"
⟨proof⟩

lemma proj_in_setD:
  assumes a: "a ∈ set (proj l A)"
  obtains k b where "a = (k, b)" "k = (ln l) ∨ k = ⋆"
⟨proof⟩

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)"
⟨proof⟩

lemma unlabel_nil[simp]: "unlabel [] = []"
⟨proof⟩

lemma unlabel_mono: "set A ⊆ set B ⟹ set (unlabel A) ⊆ set (unlabel B)"
⟨proof⟩

lemma unlabel_in: "(l,x) ∈ set A ⟹ x ∈ set (unlabel A)"
⟨proof⟩

lemma unlabel_mem_has_label: "x ∈ set (unlabel A) ⟹ ∃l. (l,x) ∈ set A"
⟨proof⟩

lemma proj_nil[simp]: "proj n [] = []" "proj_unl n [] = []"
⟨proof⟩

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]"
⟨proof⟩

lemma unlabel_nil_only_if_nil[simp]: "unlabel A = [] ⟹ A = []"
⟨proof⟩

lemma unlabel_Cons[simp]:
  "unlabel ((l,a)#A) = a#unlabel A"
  "unlabel (b#A) = snd b#unlabel A"
⟨proof⟩

lemma unlabel_append[simp]: "unlabel (A@B) = unlabel A@unlabel B"
⟨proof⟩

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"
⟨proof⟩

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"
⟨proof⟩
```

lemma *proj_unl_cons[simp]:*
"proj_unl l ((ln l, a)#A) = a#proj_unl l A"
"l $\neq$ l' $\Longrightarrow$ proj_unl l' ((ln l, a)#A) = proj_unl l' A"
"proj_unl l (($\star$, a)#A) = a#proj_unl l A"
⟨*proof*⟩

lemma *trms_unlabel_proj[simp]:*
"trms$_{stp}$ (snd (ln l, x)) $\subseteq$ trms_proj$_{lst}$ l [(ln l, x)]"
⟨*proof*⟩

lemma *trms_unlabel_star[simp]:*
"trms$_{stp}$ (snd ($\star$, x)) $\subseteq$ trms_proj$_{lst}$ l [($\star$, x)]"
⟨*proof*⟩

lemma *trms$_{lst}$_union[simp]:* "trms$_{lst}$ A = ($\bigcup$l. trms_proj$_{lst}$ l A)"
⟨*proof*⟩

lemma *trms$_{lst}$_append[simp]:* "trms$_{lst}$ (A@B) = trms$_{lst}$ A $\cup$ trms$_{lst}$ B"
⟨*proof*⟩

lemma *trms_proj$_{lst}$_append[simp]:* "trms_proj$_{lst}$ l (A@B) = trms_proj$_{lst}$ l A $\cup$ trms_proj$_{lst}$ l B"
⟨*proof*⟩

lemma *trms_proj$_{lst}$_subset[simp]:*
"trms_proj$_{lst}$ l A $\subseteq$ trms_proj$_{lst}$ l (A@B)"
"trms_proj$_{lst}$ l B $\subseteq$ trms_proj$_{lst}$ l (A@B)"
⟨*proof*⟩

lemma *trms$_{lst}$_subset[simp]:*
"trms$_{lst}$ A $\subseteq$ trms$_{lst}$ (A@B)"
"trms$_{lst}$ B $\subseteq$ trms$_{lst}$ (A@B)"
⟨*proof*⟩

lemma *vars$_{lst}$_union:* "vars$_{lst}$ A = ($\bigcup$l. vars_proj$_{lst}$ l A)"
⟨*proof*⟩

lemma *unlabel_Cons_inv:*
"unlabel A = b#B $\Longrightarrow$ $\exists$A'. ($\exists$n. A = (ln n, b)#A') $\vee$ A = ($\star$, b)#A'"
⟨*proof*⟩

lemma *unlabel_snoc_inv:*
"unlabel A = B@[b] $\Longrightarrow$ $\exists$A'. ($\exists$n. A = A'@[(ln n, b)]) $\vee$ A = A'@[($\star$, b)]"
⟨*proof*⟩

lemma *proj_idem[simp]:* "proj l (proj l A) = proj l A"
⟨*proof*⟩

lemma *proj_ik$_{st}$_is_proj_rcv_set:*
"ik$_{st}$ (proj_unl n A) = {t. (ln n, Receive t) $\in$ set A $\vee$ ($\star$, Receive t) $\in$ set A} "
⟨*proof*⟩

lemma *unlabel_ik$_{st}$_is_rcv_set:*
"ik$_{st}$ (unlabel A) = {t | l t. (l, Receive t) $\in$ set A}"
⟨*proof*⟩

lemma *proj_ik_union_is_unlabel_ik:*
"ik$_{st}$ (unlabel A) = ($\bigcup$l. ik$_{st}$ (proj_unl l A))"
⟨*proof*⟩

lemma *proj_ik_append[simp]:*
"ik$_{st}$ (proj_unl l (A@B)) = ik$_{st}$ (proj_unl l A) $\cup$ ik$_{st}$ (proj_unl l B)"
⟨*proof*⟩

lemma *proj_ik_append_subst_all:*
  "ik_{st} (proj_unl l (A@B)) $\cdot_{set}$ I = (ik_{st} (proj_unl l A) $\cdot_{set}$ I) ∪ (ik_{st} (proj_unl l B) $\cdot_{set}$ I)"
⟨*proof*⟩

lemma *ik_proj_subset[simp]:* "ik_{st} (proj_unl n A) ⊆ $trms\_proj_{lst}$ n A"
⟨*proof*⟩

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)"
⟨*proof*⟩

5.1.3 Lemmata: Well-formedness

lemma *$wfvarsoccs_{st}$_proj_union:*
  "$wfvarsoccs_{st}$ (unlabel A) = (⋃l. $wfvarsoccs_{st}$ (proj_unl l A))"
⟨*proof*⟩

lemma *wf_if_wf_proj:*
  assumes "∀l. wf_{st} V (proj_unl l A)"
  shows "wf_{st} V (unlabel A)"
⟨*proof*⟩

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. {} $\vdash_c$ 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"
⟨proof⟩

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⟩

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⟩

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;
                  homogeneous_lst (Fun f X) 𝒜 Sec;
                  Fun f X ∈ GSMP (trms_lst 𝒜)
                  ⟧ ⟹ P M (Fun f X)"
          "⋀t K T t_i. ⟦⟨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; t_i ∈ set T⟧ ⟹ P M t_i"
        shows "P M t"
⟨proof⟩

lemma ideduct_hom_mono:
```

"⟦⟨M; 𝒜; Sec⟩ $\vdash_{hom}$ t; M ⊆ M'⟧ ⟹ ⟨M'; 𝒜; Sec⟩ $\vdash_{hom}$ t"
⟨proof⟩

5.2.4 Lemmata: GSMP

lemma *GSMP_disjoint_empty[simp]:*
"GSMP_disjoint {} A Sec" "GSMP_disjoint A {} Sec"
⟨proof⟩

lemma *GSMP_mono:*
assumes "N ⊆ M"
shows "GSMP N ⊆ GSMP M"
⟨proof⟩

lemma *GSMP_SMP_mono:*
assumes "SMP N ⊆ SMP M"
shows "GSMP N ⊆ GSMP M"
⟨proof⟩

lemma *GSMP_subterm:*
assumes "t ∈ GSMP M" "t' ⊑ t"
shows "t' ∈ GSMP M"
⟨proof⟩

lemma *GSMP_subterms:* "subterms_{set} (GSMP M) = GSMP M"
⟨proof⟩

lemma *GSMP_Ana_key:*
assumes "t ∈ GSMP M" "Ana t = (K,T)" "k ∈ set K"
shows "k ∈ GSMP M"
⟨proof⟩

lemma *GSMP_append[simp]:* "GSMP (trms_{lst} (A@B)) = GSMP (trms_{lst} A) ∪ GSMP (trms_{lst} B)"
⟨proof⟩

lemma *GSMP_union:* "GSMP (A ∪ B) = GSMP A ∪ GSMP B"
⟨proof⟩

lemma *GSMP_Union:* "GSMP (trms_{lst} A) = (⋃l. GSMP (trms_proj_{lst} l A))"
⟨proof⟩

lemma *in_GSMP_in_proj:* "t ∈ GSMP (trms_{lst} A) ⟹ ∃n. t ∈ GSMP (trms_proj_{lst} n A)"
⟨proof⟩

lemma *in_proj_in_GSMP:* "t ∈ GSMP (trms_proj_{lst} n A) ⟹ t ∈ GSMP (trms_{lst} A)"
⟨proof⟩

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. {} $\vdash_c$ m}"
⟨proof⟩

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 ∨ {} $\vdash_c$ t"
⟨proof⟩

lemma *GSMP_wt_subst_subset:*
assumes "t ∈ GSMP (M $\cdot_{set}$ ℐ)" "wt_{subst} ℐ" "wf_{trms} (subst_range ℐ)"
shows "t ∈ GSMP M"
⟨proof⟩

lemma *GSMP_wt_substI:*

assumes *"t ∈ M" "wt$_{subst}$ I" "wf$_{trms}$ (subst_range I)" "interpretation$_{subst}$ I"*
shows *"t · I ∈ GSMP M"*
⟨*proof*⟩

lemma *GSMP_disjoint_subset:*
assumes *"GSMP_disjoint L R S" "L' ⊆ L" "R' ⊆ R"*
shows *"GSMP_disjoint L' R' S"*
⟨*proof*⟩

lemma *GSMP_disjoint_fst_specific_not_snd_specific:*
assumes *"GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec" "l ≠ l'"*
and *"proj_specific l m $\mathcal{A}$ Sec"*
shows *"¬proj_specific l' m $\mathcal{A}$ Sec"*
⟨*proof*⟩

lemma *GSMP_disjoint_snd_specific_not_fst_specific:*
assumes *"GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"*
and *"proj_specific l' m $\mathcal{A}$ Sec"*
shows *"¬proj_specific l m $\mathcal{A}$ Sec"*
⟨*proof*⟩

lemma *GSMP_disjoint_intersection_not_specific:*
assumes *"GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"*
and *"t ∈ Sec ∨ {} ⊢$_c$ t"*
shows *"¬proj_specific l t $\mathcal{A}$ Sec" "¬proj_specific l t $\mathcal{A}$ Sec"*
⟨*proof*⟩

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*⟩

lemma *declassified_proj_ik_subset: "declassified$_{lst}$ A I ⊆ ik$_{st}$ (proj_unl n A) ·$_{set}$ I"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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*⟩

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}"*
⟨*proof*⟩

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}"*
⟨*proof*⟩

lemma *declassified_proj_eq: "declassified$_{lst}$ A I = declassified$_{lst}$ (proj n A) I"*
⟨*proof*⟩

lemma *declassified_append: "declassified$_{lst}$ (A@B) I = declassified$_{lst}$ A I ∪ declassified$_{lst}$ B I"*

⟨*proof*⟩

```
lemma declassified_prefix_subset: "prefix A B ⟹ declassified_lst A I ⊆ declassified_lst B I"
```
⟨*proof*⟩

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'"
```
⟨*proof*⟩

```
lemma heterogeneous_secrets_anti_mono:
  assumes "heterogeneous_lst t 𝒜 Sec" "Sec' ⊆ Sec"
  shows "heterogeneous_lst t 𝒜 Sec'"
```
⟨*proof*⟩

```
lemma homogeneous_secrets_mono:
  assumes "homogeneous_lst t 𝒜 Sec'" "Sec' ⊆ Sec"
  shows "homogeneous_lst t 𝒜 Sec"
```
⟨*proof*⟩

```
lemma heterogeneous_supterm:
  assumes "heterogeneous_lst t 𝒜 Sec" "t ⊑ t'"
  shows "heterogeneous_lst t' 𝒜 Sec"
```
⟨*proof*⟩

```
lemma homogeneous_subterm:
  assumes "homogeneous_lst t 𝒜 Sec" "t' ⊑ t"
  shows "homogeneous_lst t' 𝒜 Sec"
```
⟨*proof*⟩

```
lemma proj_specific_subterm:
  assumes "t ⊑ t'" "proj_specific l t' 𝒜 Sec"
  shows "proj_specific l t 𝒜 Sec ∨ t ∈ Sec ∨ {} ⊢c t"
```
⟨*proof*⟩

```
lemma heterogeneous_term_is_Fun:
  assumes "heterogeneous_lst t A S" shows "∃f T. t = Fun f T"
```
⟨*proof*⟩

```
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*⟩

```
lemma deduct_synth_homogeneous:
  assumes "{} ⊢c t"
  shows "homogeneous_lst t 𝒜 Sec"
```
⟨*proof*⟩

```
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*⟩

```
lemma homogeneous_is_not_proj_specific:
  assumes "homogeneous_lst m 𝒜 Sec"
  shows "∃l::'lbl. ¬proj_specific l m 𝒜 Sec"
```
⟨*proof*⟩

lemma *secrets_are_homogeneous:*
  assumes "$\forall$s $\in$ Sec. P s $\longrightarrow$ ($\forall$s' $\in$ subterms s. {} $\vdash_c$ s' $\vee$ s' $\in$ Sec)" "s $\in$ Sec" "P s"
  shows "homogeneous$_{lst}$ s $\mathcal{A}$ Sec"
⟨*proof*⟩

lemma *GSMP_is_homogeneous:*
  assumes $\mathcal{A}$: "$\forall$l l'. l $\neq$ l' $\longrightarrow$ GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"
  and t: "t $\in$ GSMP (trms$_{lst}$ $\mathcal{A}$)" "t $\notin$ Sec"
  shows "homogeneous$_{lst}$ t $\mathcal{A}$ Sec"
⟨*proof*⟩

lemma *GSMP_intersection_is_homogeneous:*
  assumes $\mathcal{A}$: "$\forall$l l'. l $\neq$ l' $\longrightarrow$ GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"
    and t: "t $\in$ GSMP (trms_proj$_{lst}$ l $\mathcal{A}$) $\cap$ GSMP (trms_proj$_{lst}$ l' $\mathcal{A}$)" "l $\neq$ l'"
  shows "homogeneous$_{lst}$ t $\mathcal{A}$ Sec"
⟨*proof*⟩

lemma *GSMP_is_homogeneous':*
  assumes $\mathcal{A}$: "$\forall$l l'. l $\neq$ l' $\longrightarrow$ GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"
  and t: "t $\in$ GSMP (trms$_{lst}$ $\mathcal{A}$)"
        "t $\notin$ Sec - $\bigcup${GSMP (trms_proj$_{lst}$ l1 $\mathcal{A}$) $\cap$ GSMP (trms_proj$_{lst}$ l2 $\mathcal{A}$) | l1 l2. l1 $\neq$ l2}"
  shows "homogeneous$_{lst}$ t $\mathcal{A}$ Sec"
⟨*proof*⟩

lemma *declassified_secrets_are_homogeneous:*
  assumes $\mathcal{A}$: "$\forall$l l'. l $\neq$ l' $\longrightarrow$ GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"
    and $\mathcal{I}$: "wt$_{subst}$ $\mathcal{I}$" "wf$_{trms}$ (subst_range $\mathcal{I}$)" "interpretation$_{subst}$ $\mathcal{I}$"
    and s: "s $\in$ declassified$_{lst}$ $\mathcal{A}$ $\mathcal{I}$"
  shows "homogeneous$_{lst}$ s $\mathcal{A}$ Sec"
⟨*proof*⟩

lemma *Ana_keys_homogeneous:*
  assumes $\mathcal{A}$: "$\forall$l l'. l $\neq$ l' $\longrightarrow$ GSMP_disjoint (trms_proj$_{lst}$ l $\mathcal{A}$) (trms_proj$_{lst}$ l' $\mathcal{A}$) Sec"
  and t: "t $\in$ GSMP (trms$_{lst}$ $\mathcal{A}$)"
  and k: "Ana t = (K,T)" "k $\in$ set K"
        "k $\notin$ Sec - $\bigcup${GSMP (trms_proj$_{lst}$ l1 $\mathcal{A}$) $\cap$ GSMP (trms_proj$_{lst}$ l2 $\mathcal{A}$) | l1 l2. l1 $\neq$ l2}"
  shows "homogeneous$_{lst}$ k $\mathcal{A}$ Sec"
⟨*proof*⟩

5.2.7 Lemmata: Intruder Deduction Equivalences

lemma *deduct_if_hom_deduct:* "⟨M;A;S⟩ $\vdash_{hom}$ m $\Longrightarrow$ M $\vdash$ m"
⟨*proof*⟩

lemma *hom_deduct_if_hom_ik:*
  assumes "⟨M;A;Sec⟩ $\vdash_{hom}$ m" "$\forall$m $\in$ M. homogeneous$_{lst}$ m A Sec $\wedge$ m $\in$ GSMP (trms$_{lst}$ A)"
  shows "homogeneous$_{lst}$ m A Sec $\wedge$ m $\in$ GSMP (trms$_{lst}$ A)"
⟨*proof*⟩

lemma *deduct_hom_if_synth:*
  assumes hom: "homogeneous$_{lst}$ m $\mathcal{A}$ Sec" "m $\in$ GSMP (trms$_{lst}$ $\mathcal{A}$)"
  and m: "M $\vdash_c$ m"
  shows "⟨M; $\mathcal{A}$; Sec⟩ $\vdash_{hom}$ m"
⟨*proof*⟩

lemma *hom_deduct_if_deduct:*
  assumes $\mathcal{A}$: "par_comp $\mathcal{A}$ Sec"
  and M: "$\forall$m$\in$M. homogeneous$_{lst}$ m $\mathcal{A}$ Sec $\wedge$ m $\in$ GSMP (trms$_{lst}$ $\mathcal{A}$)"
  and m: "M $\vdash$ m" "m $\in$ GSMP (trms$_{lst}$ $\mathcal{A}$)"
shows "⟨M; $\mathcal{A}$; Sec⟩ $\vdash_{hom}$ m"
⟨*proof*⟩

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⟩
```

```
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)"
⟨proof⟩
```

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⟩
```

```
theorem par_comp_constr_typed:
  assumes 𝒜: "par_comp 𝒜 Sec"
  and ℐ: "ℐ ⊨ ⟨unlabel 𝒜⟩" "interpretation_{subst} ℐ" "wt_{subst} ℐ" "wf_{trms} (subst_range ℐ)"
  shows "(∀l. (ℐ ⊨ ⟨proj_unl l 𝒜⟩)) ∨ (∃𝒜'. prefix 𝒜' 𝒜 ∧ (strand_leaks_{lst} 𝒜' Sec ℐ))"
⟨proof⟩
```

```
theorem par_comp_constr:
  assumes 𝒜: "par_comp 𝒜 Sec" "typing_cond (unlabel 𝒜)"
  and ℐ: "ℐ ⊨ ⟨unlabel 𝒜⟩" "interpretation_{subst} ℐ"
  shows "∃ℐ_τ. interpretation_{subst} ℐ_τ ∧ wt_{subst} ℐ_τ ∧ wf_{trms} (subst_range ℐ_τ) ∧ (ℐ_τ ⊨ ⟨unlabel 𝒜⟩) ∧
             ((∀l. (ℐ_τ ⊨ ⟨proj_unl l 𝒜⟩)) ∨ (∃𝒜'. prefix 𝒜' 𝒜 ∧ (strand_leaks_{lst} 𝒜' Sec ℐ_τ)))"
⟨proof⟩
```

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 []"*
⟨*proof*⟩

lemma *par_comp_prot_impl_par_comp:*
  assumes *"par_comp_prot 𝒫 Sec" "𝒜 ∈ 𝒫"*
  shows *"par_comp 𝒜 Sec"*
⟨*proof*⟩

lemma *typing_cond_prot_impl_typing_cond:*
  assumes *"typing_cond_prot 𝒫" "𝒜 ∈ 𝒫"*
  shows *"typing_cond (unlabel 𝒜)"*
⟨*proof*⟩

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*⟩

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⟩

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 db_{lsst} **where** "db_{lsst} S ≡ db_{sst} (unlabel S)"
abbreviation db'_{lsst} **where** "db'_{lsst} S ≡ db'_{sst} (unlabel S)"

abbreviation $trms_{lsst}$ **where** "$trms_{lsst}$ S ≡ $trms_{sst}$ (unlabel S)"
abbreviation trms_proj$_{lsst}$ **where** "trms_proj$_{lsst}$ n S ≡ $trms_{sst}$ (proj_unl n S)"

abbreviation $vars_{lsst}$ **where** "$vars_{lsst}$ S ≡ $vars_{sst}$ (unlabel S)"
abbreviation vars_proj$_{lsst}$ **where** "vars_proj$_{lsst}$ n S ≡ $vars_{sst}$ (proj_unl n S)"

abbreviation $bvars_{lsst}$ **where** "$bvars_{lsst}$ S ≡ $bvars_{sst}$ (unlabel S)"
abbreviation fv_{lsst} **where** "fv_{lsst} S ≡ fv_{sst} (unlabel S)"

Labeled set-operations

fun $setops_{lsstp}$ **where**
"$setops_{lsstp}$ (i,insert⟨t,s⟩) = {(i,t,s)}"
| "$setops_{lsstp}$ (i,delete⟨t,s⟩) = {(i,t,s)}"
| "$setops_{lsstp}$ (i,⟨_: t ∈ s⟩) = {(i,t,s)}"
| "$setops_{lsstp}$ (i,∀_⟨∨≠: _ ∨∉: F'⟩) = ((λ(t,s). (i,t,s)) ` set F')"
| "$setops_{lsstp}$ _ = {}"

definition $setops_{lsst}$ **where**
"$setops_{lsst}$ S ≡ ⋃($setops_{lsstp}$ ` set S)"

6.1.2 Minor Lemmata

lemma *subst_lsst_nil[simp]:* "[] $\cdot_{lsst}$ δ = []"
⟨*proof*⟩

lemma *subst_lsst_cons:* "a#A $\cdot_{lsst}$ δ = (a $\cdot_{lsstp}$ δ)#(A $\cdot_{lsst}$ δ)"
⟨*proof*⟩

lemma *subst_lsst_singleton:* "[(l,s)] $\cdot_{lsst}$ δ = [(l,s $\cdot_{sstp}$ δ)]"
⟨*proof*⟩

lemma *subst_lsst_append:* "A@B $\cdot_{lsst}$ δ = (A $\cdot_{lsst}$ δ)@(B $\cdot_{lsst}$ δ)"
⟨*proof*⟩

lemma *subst_lsst_append_inv:*
assumes "A $\cdot_{lsst}$ δ = B1@B2"
shows "∃A1 A2. A = A1@A2 ∧ A1 $\cdot_{lsst}$ δ = B1 ∧ A2 $\cdot_{lsst}$ δ = B2"
⟨*proof*⟩

lemma *subst_lsst_member[intro]:* "x ∈ set A ⟹ x $\cdot_{lsstp}$ δ ∈ set (A $\cdot_{lsst}$ δ)"
⟨*proof*⟩

lemma *subst_lsst_unlabel_cons:* "unlabel ((l,b)#A $\cdot_{lsst}$ ϑ) = (b $\cdot_{sstp}$ ϑ)#(unlabel (A $\cdot_{lsst}$ ϑ))"
⟨*proof*⟩

lemma *subst_lsst_unlabel:* "unlabel (A $\cdot_{lsst}$ δ) = unlabel A $\cdot_{sst}$ δ"
⟨*proof*⟩

lemma *subst_lsst_unlabel_member[intro]:*
assumes "x ∈ set (unlabel A)"
shows "x $\cdot_{sstp}$ δ ∈ set (unlabel (A $\cdot_{lsst}$ δ))"
⟨*proof*⟩

lemma *subst_lsst_prefix:*
assumes "prefix B (A $\cdot_{lsst}$ ϑ)"
shows "∃C. C $\cdot_{lsst}$ ϑ = B ∧ prefix C A"
⟨*proof*⟩

lemma *$dual_{lsst}$_nil[simp]:* "$dual_{lsst}$ [] = []"

⟨proof⟩

lemma dual$_{lsst}$_Cons[simp]:
  "dual$_{lsst}$ ((l,send⟨t⟩)#A) = (l,receive⟨t⟩)#(dual$_{lsst}$ A)"
  "dual$_{lsst}$ ((l,receive⟨t⟩)#A) = (l,send⟨t⟩)#(dual$_{lsst}$ A)"
  "dual$_{lsst}$ ((l,⟨a: t $\doteq$ s⟩)#A) = (l,⟨a: t $\doteq$ s⟩)#(dual$_{lsst}$ A)"
  "dual$_{lsst}$ ((l,insert⟨t,s⟩)#A) = (l,insert⟨t,s⟩)#(dual$_{lsst}$ A)"
  "dual$_{lsst}$ ((l,delete⟨t,s⟩)#A) = (l,delete⟨t,s⟩)#(dual$_{lsst}$ A)"
  "dual$_{lsst}$ ((l,⟨a: t ∈ s⟩)#A) = (l,⟨a: t ∈ s⟩)#(dual$_{lsst}$ A)"
  "dual$_{lsst}$ ((l,∀X⟨∨≠: F ∨∉: G⟩)#A) = (l,∀X⟨∨≠: F ∨∉: G⟩)#(dual$_{lsst}$ A)"
⟨proof⟩

lemma dual$_{lsst}$_append[simp]: "dual$_{lsst}$ (A@B) = dual$_{lsst}$ A@dual$_{lsst}$ B"
⟨proof⟩

lemma dual$_{lsstp}$_subst: "dual$_{lsstp}$ (s $\cdot_{lsstp}$ δ) = (dual$_{lsstp}$ s) $\cdot_{lsstp}$ δ"
⟨proof⟩

lemma dual$_{lsst}$_subst: "dual$_{lsst}$ (S $\cdot_{lsst}$ δ) = (dual$_{lsst}$ S) $\cdot_{lsst}$ δ"
⟨proof⟩

lemma dual$_{lsst}$_subst_unlabel: "unlabel (dual$_{lsst}$ (S $\cdot_{lsst}$ δ)) = unlabel (dual$_{lsst}$ S) $\cdot_{sst}$ δ"
⟨proof⟩

lemma dual$_{lsst}$_subst_cons: "dual$_{lsst}$ (a#A $\cdot_{lsst}$ σ) = (dual$_{lsstp}$ a $\cdot_{lsstp}$ σ)#(dual$_{lsst}$ (A $\cdot_{lsst}$ σ))"
⟨proof⟩

lemma dual$_{lsst}$_subst_append: "dual$_{lsst}$ (A@B $\cdot_{lsst}$ σ) = (dual$_{lsst}$ A@dual$_{lsst}$ B) $\cdot_{lsst}$ σ"
⟨proof⟩

lemma dual$_{lsst}$_subst_snoc: "dual$_{lsst}$ (A@[a] $\cdot_{lsst}$ σ) = (dual$_{lsst}$ A $\cdot_{lsst}$ σ)@[dual$_{lsstp}$ a $\cdot_{lsstp}$ σ]"
⟨proof⟩

lemma dual$_{lsst}$_memberD:
  assumes "(l,a) ∈ set (dual$_{lsst}$ A)"
  shows "∃b. (l,b) ∈ set A ∧ dual$_{lsstp}$ (l,b) = (l,a)"
  ⟨proof⟩

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⟩

lemma dual$_{lsst}$_self_inverse: "dual$_{lsst}$ (dual$_{lsst}$ A) = A"
⟨proof⟩

lemma vars$_{sst}$_unlabel_dual$_{lsst}$_eq: "vars$_{lsst}$ (dual$_{lsst}$ A) = vars$_{lsst}$ A"
⟨proof⟩

lemma fv$_{sst}$_unlabel_dual$_{lsst}$_eq: "fv$_{lsst}$ (dual$_{lsst}$ A) = fv$_{lsst}$ A"
⟨proof⟩

lemma bvars$_{sst}$_unlabel_dual$_{lsst}$_eq: "bvars$_{lsst}$ (dual$_{lsst}$ A) = bvars$_{lsst}$ A"
⟨proof⟩

lemma vars$_{sst}$_unlabel_Cons: "vars$_{lsst}$ ((l,b)#A) = vars$_{sstp}$ b ∪ vars$_{lsst}$ A"
⟨proof⟩

lemma fv$_{sst}$_unlabel_Cons: "fv$_{lsst}$ ((l,b)#A) = fv$_{sstp}$ b ∪ fv$_{lsst}$ A"
⟨proof⟩

```
lemma bvars_sst_unlabel_Cons: "bvars_lsst ((l,b)#A) = set (bvars_sstp b) ∪ bvars_lsst A"
⟨proof⟩

lemma bvars_lsst_subst: "bvars_lsst (A ·_lsst δ) = bvars_lsst A"
⟨proof⟩

lemma dual_lsst_member:
  assumes "(l,x) ∈ set A"
    and "¬is_Receive x" "¬is_Send x"
  shows "(l,x) ∈ set (dual_lsst A)"
⟨proof⟩

lemma dual_lsst_unlabel_member:
  assumes "x ∈ set (unlabel A)"
    and "¬is_Receive x" "¬is_Send x"
  shows "x ∈ set (unlabel (dual_lsst A))"
⟨proof⟩

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⟩

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))"
⟨proof⟩

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⟩

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"
  ⟨proof⟩

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⟩
```

lemma $trms_{sst}$_unlabel_dual$_{lsst}$_eq: "$trms_{lsst}$ (dual$_{lsst}$ A) = $trms_{lsst}$ A"
⟨proof⟩

lemma $trms_{sst}$_unlabel_subst_cons:
"$trms_{lsst}$ ((l,b)#A $\cdot_{lsst}$ δ) = $trms_{sstp}$ (b $\cdot_{sstp}$ δ) ∪ $trms_{lsst}$ (A $\cdot_{lsst}$ δ)"
⟨proof⟩

lemma $trms_{sst}$_unlabel_subst:
assumes "$bvars_{lsst}$ S ∩ subst_domain ϑ = {}"
shows "$trms_{lsst}$ (S $\cdot_{lsst}$ ϑ) = $trms_{lsst}$ S $\cdot_{set}$ ϑ"
⟨proof⟩

lemma $trms_{sst}$_unlabel_subst':
fixes t::"('a,'b) term" **and** δ::"('a,'b) subst"
assumes "t ∈ $trms_{lsst}$ (S $\cdot_{lsst}$ δ)"
shows "∃s ∈ $trms_{lsst}$ S. ∃X. set X ⊆ $bvars_{lsst}$ S ∧ t = s · rm_vars (set X) δ"
⟨proof⟩

lemma $trms_{sst}$_unlabel_subst'':
fixes t::"('a,'b) term" **and** δ ϑ::"('a,'b) subst"
assumes "t ∈ $trms_{lsst}$ (S $\cdot_{lsst}$ δ) $\cdot_{set}$ ϑ"
shows "∃s ∈ $trms_{lsst}$ S. ∃X. set X ⊆ $bvars_{lsst}$ S ∧ t = s · rm_vars (set X) δ $\circ_s$ ϑ"
⟨proof⟩

lemma $trms_{sst}$_unlabel_dual_subst_cons:
"$trms_{lsst}$ (dual$_{lsst}$ (a#A $\cdot_{lsst}$ σ)) = ($trms_{sstp}$ (snd a $\cdot_{sstp}$ σ)) ∪ ($trms_{lsst}$ (dual$_{lsst}$ (A $\cdot_{lsst}$ σ)))"
⟨proof⟩

lemma dual$_{lsst}$_funs_term:
"⋃(funs_term ` ($trms_{sst}$ (unlabel (dual$_{lsst}$ S)))) = ⋃(funs_term ` ($trms_{sst}$ (unlabel S)))"
⟨proof⟩

lemma dual$_{lsst}$_db$_{lsst}$:
"db'$_{lsst}$ (dual$_{lsst}$ A) = db'$_{lsst}$ A"
⟨proof⟩

lemma db$_{sst}$_unlabel_append:
"db'$_{lsst}$ (A@B) I D = db'$_{lsst}$ B I (db'$_{lsst}$ A I D)"
⟨proof⟩

lemma db$_{sst}$_dual$_{lsst}$:
"db'$_{sst}$ (unlabel (dual$_{lsst}$ (T $\cdot_{lsst}$ δ))) $\mathcal{I}$ D = db'$_{sst}$ (unlabel (T $\cdot_{lsst}$ δ)) $\mathcal{I}$ D"
⟨proof⟩

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)"
⟨proof⟩

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))"
⟨proof⟩

lemma proj_subst: "proj l (A $\cdot_{lsst}$ δ) = proj l A $\cdot_{lsst}$ δ"
⟨proof⟩

lemma proj_set_subset[simp]:
"set (proj n A) ⊆ set A"
⟨proof⟩

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)"
```
⟨proof⟩

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"
```
⟨proof⟩

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"
```
⟨proof⟩

lemma *proj_filter:* "proj i [d←D. d ∉ set Di] = [d←proj i D. d ∉ set Di]"
⟨proof⟩

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"
```
⟨proof⟩

lemma *proj_dual$_{lsst}$:*
"proj l (dual$_{lsst}$ A) = dual$_{lsst}$ (proj l A)"
⟨proof⟩

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⟩

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) = []"
```
⟨proof⟩

lemma *dbproj_Cons:*
```
"dbproj i ((i,d)#D) = (i,d)#dbproj i D"
"i ≠ j ⟹ dbproj j ((i,d)#D) = dbproj j D"
```
⟨proof⟩

lemma *dbproj_subset[simp]:*
"set (unlabel (dbproj i D)) ⊆ set (unlabel D)"
⟨proof⟩

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⟩

lemma *dbproj_subseq_subset:*
assumes "Di ∈ set (subseqs (dbproj i D))"
shows "set Di ⊆ set D"
⟨proof⟩

lemma *dbproj_subseq_in_subseqs:*
assumes "Di ∈ set (subseqs (dbproj i D))"

```
  shows "Di ∈ set (subseqs D)"
⟨proof⟩

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⟩

lemma unlabel_subseqsD:
  assumes "A ∈ set (subseqs (unlabel B))"
  shows "∃C ∈ set (subseqs B). unlabel C = A"
⟨proof⟩

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]"
⟨proof⟩

lemma subseqs_mem_dbproj:
  assumes "Di ∈ set (subseqs D)" "list_all (λd. fst d = i) Di"
  shows "Di ∈ set (subseqs (dbproj i D))"
⟨proof⟩

lemma unlabel_subst: "unlabel S ·sst δ = unlabel (S ·lsst δ)"
⟨proof⟩

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"
⟨proof⟩

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"
⟨proof⟩

lemma labeled_stateful_strand_subst_comp:
  assumes "range_vars δ ∩ bvarslsst S = {}"
  shows "S ·lsst δ ∘s ϑ = (S ·lsst δ) ·lsst ϑ"
⟨proof⟩

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)"
⟨proof⟩

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⟩

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)
⟨proof⟩

lemma trmssst_unlabel_suffix_subset:
  "trmssst (unlabel B) ⊆ trmssst (unlabel (A@B))"
  "trmssst (proj_unl n B) ⊆ trmssst (proj_unl n (A@B))"
⟨proof⟩
```

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*⟩

lemma *setops*$_{lsst}$*_nil[simp]:*
  "setops$_{lsst}$ *[] = {}"*
⟨*proof*⟩

lemma *setops*$_{lsst}$*_cons[simp]:*
  "setops$_{lsst}$ *(x#S) = setops*$_{lsstp}$ *x ∪ setops*$_{lsst}$ *S"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *setops*$_{lsst}$*_prefix_subset:*
  "setops$_{lsst}$ *A ⊆ setops*$_{lsst}$ *(A@B)"*
  "setops$_{lsst}$ *(proj n A) ⊆ setops*$_{lsst}$ *(proj n (A@B))"*
⟨*proof*⟩

lemma *setops*$_{lsst}$*_suffix_subset:*
  "setops$_{lsst}$ *B ⊆ setops*$_{lsst}$ *(A@B)"*
  "setops$_{lsst}$ *(proj n B) ⊆ setops*$_{lsst}$ *(proj n (A@B))"*
⟨*proof*⟩

lemma *setops*$_{lsst}$*_mono:*
  "set M ⊆ set N ⟹ setops$_{lsst}$ *M ⊆ setops*$_{lsst}$ *N"*
⟨*proof*⟩

lemma *trms*$_{sst}$*_unlabel_subset_if_no_label:*
  "¬list_ex (is_LabelN l) A ⟹ trms$_{lsst}$ *(proj l A) ⊆ trms*$_{lsst}$ *(proj l' A)"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *setops*$_{lsst}$*_proj_subset_if_no_label:*
  "¬list_ex (is_LabelN l) A ⟹ setops$_{lsst}$ *(proj l A) ⊆ setops*$_{lsst}$ *(proj l' A)"*
⟨*proof*⟩

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⟩

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⟩

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"
⟨proof⟩

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"
⟨proof⟩

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⟩

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")
  ⟨proof⟩

lemma setops_lsst_dual_lsst_eq:
  "setops_lsst (dual_lsst A) = setops_lsst A"
⟨proof⟩

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⟩

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⟩
```

```
lemma (in stateful_typed_model) fv_pair_fv_pairs_subset:
  assumes "d ∈ set D"
  shows "fv (pair (snd d)) ⊆ fv_pairs (unlabel D)"
⟨proof⟩
```

```
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"
⟨proof⟩
```

```
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")
⟨proof⟩
```

```
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 "fv_pairs (map snd D) ∩ set X = {}"
  shows "⟦M; map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' (map snd D))⟧_d ℐ ⟷
         ⟦M; map (λG. ∀X⟨∨≠: (F@G)⟩_st) (tr_pairs F' (map snd (dbproj i D)))⟧_d ℐ"
⟨proof⟩
```

```
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"
⟨proof⟩
```

```
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⟩
```

```
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))"
⟨proof⟩
```

```
lemma (in stateful_typed_model) setops_sst_unlabel_dual_lsst_eq:
  "setops_sst (unlabel (dual_lsst A)) = setops_sst (unlabel A)"
⟨proof⟩
```

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')"
⟨proof⟩

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 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←dbproj i D. d ∉ set Di])@B)
                           (tr_pc A [d←D. d ∉ set Di]))
                  (subseqs (dbproj i D))))"
| "tr_pc ((i,⟨ac: t ∈ s⟩)#A) D =
    concat (map (λB. map (λd. (i,⟨ac: (pair (t,s)) ≐ (pair (snd d))⟩_st)#B) (dbproj i D)) (tr_pc A D))"
| "tr_pc ((i,∀X⟨∨≠: F ∨∉: F' ⟩)#A) D =
    map ((@) (map (λG. (i,∀X⟨∨≠: (F@G)⟩_st)) (tr_pairs F' (map snd (dbproj i D))))) (tr_pc A D)"
```

6.2.3 Small Lemmata

```
lemma par_comp_lsst_nil:
  assumes "ground Sec" "∀s ∈ Sec. ∀s'∈subterms s. {} ⊢_c s' ∨ s' ∈ Sec"
  shows "par_comp_lsst [] Sec"
⟨proof⟩

lemma par_comp_lsst_subset:
  assumes A: "par_comp_lsst A Sec"
    and BA: "set B ⊆ set A"
  shows "par_comp_lsst B Sec"
⟨proof⟩

lemma par_comp_lsst_split:
  assumes "par_comp_lsst (A@B) Sec"
  shows "par_comp_lsst A Sec" "par_comp_lsst B Sec"
⟨proof⟩

lemma par_comp_lsst_proj:
  assumes "par_comp_lsst A Sec"
  shows "par_comp_lsst (proj n A) Sec"
⟨proof⟩

lemma par_comp_lsst_dual_lsst:
  assumes A: "par_comp_lsst A S"
  shows "par_comp_lsst (dual_lsst A) S"
⟨proof⟩

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⟩

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))"
⟨proof⟩

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⟩

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⟩
```

lemma *setops$_{lsst}$_wt_instance_ex:*
assumes *B: "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a $\cdot_{lsstp}$ 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 $\cdot_p$ δ ∧ wt$_{subst}$ δ ∧ wf$_{trms}$ (subst_range δ)"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *tr_par_declassified_eq:*
"B ∈ set (tr$_{pc}$ A D) ⟹ declassified$_{lst}$ B I = declassified$_{lsst}$ A I"
⟨*proof*⟩

lemma *tr_par_ik_eq:*
assumes *"B ∈ set (tr$_{pc}$ A D)"*
shows *"ik$_{st}$ (unlabel B) = ik$_{sst}$ (unlabel A)"*
⟨*proof*⟩

lemma *tr_par_deduct_iff:*
assumes *"B ∈ set (tr$_{pc}$ A D)"*
shows *"ik$_{st}$ (unlabel B) $\cdot_{set}$ I ⊢ t ⟷ ik$_{sst}$ (unlabel A) $\cdot_{set}$ I ⊢ t"*
⟨*proof*⟩

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*⟩

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' = {}"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *tr_par_wf_trms:*
assumes *"A' ∈ set (tr$_{pc}$ A [])" "wf$_{trms}$ (trms$_{sst}$ (unlabel A))"*
shows *"wf$_{trms}$ (trms$_{lst}$ A')"*
⟨*proof*⟩

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*⟩

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' = {}"*

⟨proof⟩

```
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⟩

```
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⟩

```
lemma tr_par_proj:
  assumes "B ∈ set (tr_{pc} A D)"
  shows "proj n B ∈ set (tr_{pc} (proj n A) (proj n D))"
```
⟨proof⟩

```
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⟩

```
lemma tr_par_preserves_par_comp:
  assumes "par_comp_{lsst} A Sec" "A' ∈ set (tr_{pc} A [])"
  shows "par_comp A' Sec"
```
⟨proof⟩

```
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⟩

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)) (tr_pairs F' (map snd D)))) (tr'_pc A D)"
```

Part 1

private lemma *tr'_par_iff_unlabel_tr:*
```
  assumes "∀(i,p) ∈ setops_lsst A ∪ set D.
           ∀(j,q) ∈ setops_lsst 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*⟩

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*⟩

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*⟩

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")
```
⟨*proof*⟩

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*⟩

6.2.7 Theorem: The Stateful Compositionality Result, on the Protocol Level

abbreviation wf_{lsst} **where**
```
  "wf_lsst V 𝒜 ≡ wf'_sst V (unlabel 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 𝒮 ≡ (∀𝒜 ∈ 𝒮. wf_lsst {} 𝒜) ∧ (∀𝒜 ∈ 𝒮. ∀𝒜' ∈ 𝒮. fv_lsst 𝒜 ∩ bvars_lsst 𝒜' = {})"

definition wf_lssts'::
  "('fun,'var,'lbl) labeled_stateful_strand set ⇒ ('fun,'var,'lbl) labeled_stateful_strand ⇒ bool"
where
  "wf_lssts' 𝒮 𝒜 ≡ (∀𝒜' ∈ 𝒮. wf'_sst (wfrestrictedvars_lsst 𝒜) (unlabel 𝒜')) ∧
                   (∀𝒜' ∈ 𝒮. ∀𝒜'' ∈ 𝒮. fv_lsst 𝒜' ∩ bvars_lsst 𝒜'' = {}) ∧
                   (∀𝒜' ∈ 𝒮. fv_lsst 𝒜' ∩ bvars_lsst 𝒜 = {}) ∧
                   (∀𝒜' ∈ 𝒮. fv_lsst 𝒜 ∩ bvars_lsst 𝒜' = {})"

definition typing_cond_prot_stateful where
  "typing_cond_prot_stateful 𝒫 ≡
    wf_lssts 𝒫 ∧
    tfr_set (⋃(trms_lsst ` 𝒫) ∪ pair ` ⋃(setops_sst ` unlabel ` 𝒫)) ∧
    wf_trms (⋃(trms_lsst ` 𝒫)) ∧
    (∀S ∈ 𝒫. list_all tfr_sstp (unlabel S))"

definition par_comp_prot_stateful where
  "par_comp_prot_stateful 𝒫 Sec ≡
    (∀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) ∧
    ground Sec ∧ (∀s ∈ Sec. ∀s' ∈ subterms s. {} ⊢_c s' ∨ s' ∈ Sec) ∧
    (∀(i,p) ∈ ⋃𝒜 ∈ 𝒫. setops_lsst 𝒜. ∀(j,q) ∈ ⋃𝒜 ∈ 𝒫. setops_lsst 𝒜.
      (∃δ. Unifier δ (pair p) (pair q)) ⟶ i = j) ∧
    typing_cond_prot_stateful 𝒫"

definition component_secure_prot_stateful where
  "component_secure_prot_stateful 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 ⊨_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]))))))"

definition component_leaks_stateful where
  "component_leaks_stateful 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_lsst 𝒜' ℐ_τ. (ℐ_τ ⊨_s (proj_unl n 𝒜'@[Send t]))))"

definition unsat_stateful where
  "unsat_stateful 𝒜 ≡ (∀ℐ. interpretation_subst ℐ ⟶ ¬(ℐ ⊨_s unlabel 𝒜))"

lemma wf_lssts_eqs_wf_lssts'[simp]: "wf_lssts S = wf_lssts' S []"
⟨proof⟩

lemma par_comp_prot_impl_par_comp_stateful:
  assumes "par_comp_prot_stateful 𝒫 Sec" "𝒜 ∈ 𝒫"
  shows "par_comp_lsst 𝒜 Sec"
⟨proof⟩

lemma typing_cond_prot_impl_typing_cond_stateful:
  assumes "typing_cond_prot_stateful 𝒫" "𝒜 ∈ 𝒫"
  shows "typing_cond_sst (unlabel 𝒜)"
⟨proof⟩

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⟩

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 (fvset (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_complsst where
  "comp_par_complsst 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_listsst B@map (pair' pair_fun) (setops_listsst B));
      pr = λl. MP0 (proj_unl l A)
  in length L > 1 ∧
     list_all (wftrm' arity) (MP0 (unlabel A)) ∧
     list_all (wftrm' arity) C ∧
     has_all_wt_instances_of Γ (subtermsset (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) ∈ setopslsst A. ∀(j,q) ∈ setopslsst 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
⟨proof⟩

lemma GSMP_disjoint_if_comp_GSMP_disjoint:
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}}"
  assumes AB'_wf: "list_all (wftrm' arity) A'" "list_all (wftrm' arity) B'"
    and C_wf: "list_all (wftrm' 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})"
⟨proof⟩

lemma par_complsst_if_comp_par_complsst:
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (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*⟩

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*⟩

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⟩

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)"
⟨proof⟩

lemma assm2: "Ana (Fun f T) = (K, M) ⟹ set M ⊆ set T"
⟨proof⟩

lemma assm6: "0 < arity f ⟹ public f" ⟨proof⟩

global_interpretation im: intruder_model arity public Ana
  defines wf_trm = "im.wf_trm"
    and wf_trms = "im.wf_trms"
⟨proof⟩
```

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" ⟨proof⟩
```

lemma *assm8: "0 < arity f* $\Longrightarrow$ Γ *(Fun f X) = TComp f (map* Γ *X)"* ⟨*proof*⟩

lemma *assm9: "infinite {c.* Γ *(Fun c []) = TAtom a* $\wedge$ *public c}"*
⟨*proof*⟩

lemma *assm10: "TComp f T* $\sqsubseteq$ Γ *t* $\Longrightarrow$ *arity f > 0"*
⟨*proof*⟩

lemma *assm11: "im.wf*$_{trm}$ *(*Γ *(Var x))"*
⟨*proof*⟩

lemma *assm12: "*Γ *(Var (*τ*, n)) =* Γ *(Var (*τ*, m))"*
⟨*proof*⟩

lemma *Ana_const: "arity c = 0* $\Longrightarrow$ *Ana (Fun c T) = ([],[])"*
⟨*proof*⟩

lemma *Ana_keys_subterm: "Ana t = (K,T)* $\Longrightarrow$ *k* $\in$ *set K* $\Longrightarrow$ *k* $\sqsubset$ *t"*
⟨*proof*⟩

global_interpretation *tm: typed_model' arity public Ana* Γ
⟨*proof*⟩

7.1.5 TLS example: Proving type-flaw resistance

abbreviation Γ_v*_clientHello* **where**
"Γ_v*_clientHello* $\equiv$
TComp clientHello [TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce]"

abbreviation Γ_v*_serverHello* **where**
"Γ_v*_serverHello* $\equiv$
TComp serverHello [TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce]"

abbreviation Γ_v*_pub* **where**
"Γ_v*_pub* $\equiv$ *TComp pub [TAtom PrivKey]"*

abbreviation Γ_v*_x509* **where**
"Γ_v*_x509* $\equiv$ *TComp x509 [TAtom Agent,* Γ_v*_pub]"*

abbreviation Γ_v*_sign* **where**
"Γ_v*_sign* $\equiv$ *TComp sign [TAtom PrivKey,* Γ_v*_x509]"*

abbreviation Γ_v*_serverCert* **where**
"Γ_v*_serverCert* $\equiv$ *TComp serverCert [*Γ_v*_sign]"*

abbreviation Γ_v*_pmsForm* **where**
"Γ_v*_pmsForm* $\equiv$ *TComp pmsForm [TAtom SymKey]"*

abbreviation Γ_v*_crypt* **where**
"Γ_v*_crypt* $\equiv$ *TComp crypt [*Γ_v*_pub,* Γ_v*_pmsForm]"*

abbreviation Γ_v*_clientKeyExchange* **where**
"Γ_v*_clientKeyExchange* $\equiv$
TComp clientKeyExchange [Γ_v*_crypt]"*

abbreviation Γ_v*_HSMsgs* **where**
"Γ_v*_HSMsgs* $\equiv$ *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})"*
⟨*proof*⟩

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}"
⟨proof⟩

instance ex_atom::finite
⟨proof⟩

lemma ex_lbl_UNIV:
  "(UNIV::ex_lbl set) = {Label1, Label2}"
⟨proof⟩

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)"
⟨proof⟩

lemma assm2: "Ana (Fun f T) = (K, M) ⟹ set M ⊆ set T"
⟨proof⟩

lemma assm6: "0 < arity f ⟹ public f" ⟨proof⟩

global_interpretation im: intruder_model arity public Ana
  defines wf_trm = "im.wf_trm"
⟨proof⟩

type_synonym ex_strand_step = "(ex_fun,ex_var) strand_step"
type_synonym ex_strand = "(ex_fun,ex_var) strand"
```

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 (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" ⟨proof⟩

lemma assm8: "0 < arity f ⟹ Γ (Fun f X) = TComp f (map Γ X)" ⟨proof⟩

lemma assm9: "infinite {c. Γ (Fun c []) = TAtom a ∧ public c}"
⟨proof⟩

lemma assm10: "TComp f T ⊑ Γ t ⟹ arity f > 0"
```

⟨proof⟩

lemma *assm11: "im.wf$_{trm}$ (Γ (Var x))"*
⟨proof⟩

lemma *assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))"*
⟨proof⟩

lemma *Ana_const: "arity c = 0 ⟹ Ana (Fun c T) = ([], [])"*
⟨proof⟩

lemma *Ana_subst': "Ana (Fun f T) = (K,M) ⟹ Ana (Fun f T · δ) = (K ·$_{list}$ δ,M ·$_{list}$ δ)"*
⟨proof⟩

global_interpretation *tm: typed_model' arity public Ana Γ*
⟨proof⟩

Locale interpretation: labeled stateful typed model

global_interpretation *stm: labeled_stateful_typed_model' arity public Ana Γ tuple* **1 2**
⟨proof⟩

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⟩

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^2_1~~
⟨**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^3_1~~
⟨⋆, ⟨PK 0 in beginauthset 0 (A 0) (A 1)⟩⟩,
⟨⋆, ⟨PK 0 in endauthset 0 (A 0) (A 1)⟩⟩,

~~rule R^4_1~~
⟨⋆, receive⟨PK 0⟩⟩,
⟨⋆, receive⟨invkeymsg (PK 0)⟩⟩,

~~rule R^5_1~~
⟨**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^6_1~~
⟨**1**, select⟨PK 0, ringset (A 0)⟩⟩,
⟨**1**, delete⟨PK 0, ringset (A 0)⟩⟩,

~~rule R^7_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 R^8_1~~
~~nothing new~~

~~rule R^9_1~~
⟨**1**, send⟨PK 0⟩⟩,

~~rule R^10_1~~
⟨**1**, send⟨Fun (attack 0) []⟩⟩,

~~constraint steps from the second protocol (duplicate steps are ignored)~~
~~rule R^2_1~~
⟨**2**, send⟨invkeymsg (PK 0)⟩⟩,
⟨⋆, ⟨PK 0 in validset (A 0) (A 1)⟩⟩,
⟨**2**, receive⟨Fun (attack 1) []⟩⟩,

~~rule R^2_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 R^3_2~~
⟨⋆, ⟨PK 0 in beginauthset 1 (A 0) (A 1)⟩⟩,
⟨⋆, ⟨PK 0 in endauthset 1 (A 0) (A 1)⟩⟩,

~~rule R^4_2~~
⟨⋆, receive⟨PK 0⟩⟩,
⟨⋆, receive⟨invkeymsg (PK 0)⟩⟩,

~~rule R^5_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 R^6_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 R^7_2~~
⟨**2**, receive⟨pwmsg (A 0) (A 1)⟩⟩,

~~rule R^8_2~~
~~nothing new~~

~~rule R^9_2~~
⟨**2**, insert⟨PK 0, pubkeysset (A 0)⟩⟩,

~~rule R^10_2~~
⟨**2**, send⟨Fun (attack 1) []⟩⟩
]"

theorem "stm.tfr$_{sst}$ (unlabel S'$_{ks}$)"
⟨*proof*⟩

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"

$\langle proof \rangle$

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.