

Stateful Protocol Composition and Typing

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

February 6, 2026

*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	7
2	Preliminaries and Intruder Model	11
2.1	Miscellaneous Lemmata	11
2.2	Protocol Messages as (First-Order) Terms	16
2.3	Definitions and Properties Related to Substitutions and Unification	22
2.4	Dolev-Yao Intruder Model	45
3	The Typing Result for Non-Stateful Protocols	57
3.1	Strands and Symbolic Intruder Constraints	57
3.2	The Lazy Intruder	76
3.3	The Typed Model	79
3.4	The Typing Result	92
4	The Typing Result for Stateful Protocols	105
4.1	Stateful Strands	105
4.2	Extending the Typing Result to Stateful Constraints	124
5	The Parallel Composition Result for Non-Stateful Protocols	133
5.1	Labeled Strands	133
5.2	Parallel Compositionality of Security Protocols	136
6	The Stateful Protocol Composition Result	145
6.1	Labeled Stateful Strands	145
6.2	Stateful Protocol Compositionality	155
7	Examples	169
7.1	Proving Type-Flaw Resistance of the TLS Handshake Protocol	169
7.2	The Keyserver Example	173

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 ??) 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, 5] and [1, chapter 5].

Overall, the structure of this document follows the theory dependencies (see ??): 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.

1 Introduction

,ü,

,ü,,ü,

2 Preliminaries and Intruder Model

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

2.1 Miscellaneous Lemmata

```
theory Miscellaneous
  imports Main "HOL-Library.Sublist" "HOL-Library.Infinite_Set" "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 in_set_zip_swap: "(x,y) ∈ set (zip xs ys) ⟷ (y,x) ∈ set (zip ys xs)"
⟨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 list_all2_in_set_ex:
  assumes P: "list_all2 P xs ys"
  and x: "x ∈ set xs"
  shows "∃y ∈ set ys. P x y"
⟨proof⟩
```

```
lemma list_all2_in_set_ex':
  assumes P: "list_all2 P xs ys"
  and y: "y ∈ set ys"
  shows "∃x ∈ set xs. P x y"
⟨proof⟩
```

```
lemma list_all2_sym:
  assumes "∧x y. P x y ⟹ P y x"
  and "list_all2 P xs ys"
  shows "list_all2 P ys xs"
⟨proof⟩
```

```

lemma map_upt_index_eq:
  assumes "j < length xs"
  shows "(map (λi. xs ! is i) [0..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  $\subseteq$  set xs"
<proof>

```

2.1.3 List: prefixes, suffixes

```

lemma suffix_Cons': "suffix [x] (y#ys)  $\implies$  suffix [x] ys  $\vee$  (y = x  $\wedge$  ys = [])"
<proof>

```

```

lemma prefix_Cons': "prefix (x#xs) (x#ys)  $\implies$  prefix xs ys"
<proof>

```

```

lemma prefix_map: "prefix xs (map f ys)  $\implies$   $\exists$ zs. prefix zs ys  $\wedge$  map f zs = xs"
<proof>

```

```

lemma concat_mono_prefix: "prefix xs ys  $\implies$  prefix (concat xs) (concat ys)"
<proof>

```

```

lemma concat_map_mono_prefix: "prefix xs ys  $\implies$  prefix (concat (map f xs)) (concat (map f ys))"
<proof>

```

```

lemma length_prefix_ex:
  assumes "n  $\leq$  length xs"
  shows " $\exists$ ys zs. xs = ys@zs  $\wedge$  length ys = n"
<proof>

```

```

lemma length_prefix_ex':
  assumes "n < length xs"
  shows " $\exists$ ys zs. xs = ys@xs ! n#zs  $\wedge$  length ys = n"
<proof>

```

```

lemma length_prefix_ex2:
  assumes "i < length xs" "j < length xs" "i < j"
  shows " $\exists$ ys zs vs. xs = ys@xs ! i#zs@xs ! j#vs  $\wedge$  length ys = i  $\wedge$  length zs = j - i - 1"
<proof>

```

```

lemma prefix_prefix_inv:
  assumes xs: "prefix xs (ys@zs)"
  and x: "suffix [x] xs"
  shows "prefix xs ys  $\vee$  x  $\in$  set zs"
<proof>

```

```

lemma prefix_snoc_obtain:
  assumes xs: "prefix (xs@[x]) (ys@zs)"
  and ys: " $\neg$ prefix (xs@[x]) ys"
  obtains vs where "xs@[x] = ys@vs@[x]" "prefix (vs@[x]) zs"
<proof>

```

```

lemma prefix_snoc_in_iff: "x  $\in$  set xs  $\longleftrightarrow$  ( $\exists$ B. prefix (B@[x]) xs)"
<proof>

```

2.1.4 List: products

```

lemma product_lists_Cons:
  "x#xs  $\in$  set (product_lists (y#ys))  $\longleftrightarrow$  (xs  $\in$  set (product_lists ys)  $\wedge$  x  $\in$  set y)"
<proof>

```

```

lemma product_lists_in_set_nth:
  assumes "xs  $\in$  set (product_lists ys)"
  shows " $\forall$ i<length ys. xs ! i  $\in$  set (ys ! i)"
<proof>

```

```

lemma product_lists_in_set_nth':
  assumes " $\forall$ i<length xs. ys ! i  $\in$  set (xs ! i)"

```

```

    and "length xs = length ys"
    shows "ys ∈ set (product_lists xs)"
  ⟨proof⟩

```

2.1.5 Other Lemmata

```

lemma finite_ballI:
  "∀l ∈ {}. P l" "P x ⇒ ∀l ∈ xs. P l ⇒ ∀l ∈ insert x xs. P l"
  ⟨proof⟩

```

```

lemma list_set_ballI:
  "∀l ∈ set []. P l" "P x ⇒ ∀l ∈ set xs. P l ⇒ ∀l ∈ set (x#xs). P l"
  ⟨proof⟩

```

```

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..

```

```

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⟩

lemma ex_list_obtain:
  assumes ts: "∧ t. t ∈ set ts ⇒ ∃ s. P t s"
  obtains ss where "length ss = length ts" "∀ i < length ss. P (ts ! i) (ss ! i)"
⟨proof⟩

lemma length_1_conv[iff]:
  "(length ts = 1) = (∃ a. ts = [a])"
⟨proof⟩

lemma length_2_conv[iff]:
  "(length ts = 2) = (∃ a b. ts = [a,b])"
⟨proof⟩

lemma length_3_conv[iff]:
  "(length ts = 3) ↔ (∃ a b c. ts = [a,b,c])"
⟨proof⟩

lemma Max_nat_finite_le:
  assumes "finite M"
  and "∧ m. m ∈ M ⇒ f m ≤ (n::nat)"
  shows "Max (insert 0 (f ` M)) ≤ n"
⟨proof⟩

lemma Max_nat_finite_lt:
  assumes "finite M"
  and "M ≠ {}"
  and "∧ m. m ∈ M ⇒ f m < (n::nat)"
  shows "Max (f ` M) < n"
⟨proof⟩

lemma ex_finite_disj_nat_inj:
  fixes N N'::"nat set"
  assumes N: "finite N"
  and N': "finite N'"
  shows "∃ M::nat set. ∃ δ::nat ⇒ nat. inj δ ∧ δ ` N = M ∧ M ∩ N' = {}"
⟨proof⟩

```

2.1.6 Infinite Paths in Relations as Mappings from Naturals to States

```

context
begin

```

```

private fun rel_chain_fun::"nat  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  ('a  $\times$  'a) set  $\Rightarrow$  '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)  $\in$  r)"

lemma infinite_chain_intro:
  fixes r:: "('a  $\times$  'a) set"
  assumes " $\forall (a,b) \in r. \exists c. (b,c) \in r$ " "r  $\neq$  {}"
  shows " $\exists f. \forall i::nat. (f i, f (Suc i)) \in r$ "
<proof>

end

lemma infinite_chain_intro':
  fixes r:: "('a  $\times$  'a) set"
  assumes base: " $\exists b. (x,b) \in r$ " and step: " $\forall b. (x,b) \in r^+ \longrightarrow (\exists c. (b,c) \in r)$ "
  shows " $\exists f. \forall i::nat. (f i, f (Suc i)) \in r$ "
<proof>

lemma infinite_chain_mono:
  assumes "S  $\subseteq$  T" " $\exists f. \forall i::nat. (f i, f (Suc i)) \in S$ "
  shows " $\exists f. \forall i::nat. (f i, f (Suc i)) \in T$ "
<proof>

end

```

2.2 Protocol Messages as (First-Order) Terms

```

theory Messages
  imports Miscellaneous "First_Order_Terms.Term"
begin

```

2.2.1 Term-related definitions: subterms and free variables

```

abbreviation "the_Fun  $\equiv$  un_Fun1"
lemmas the_Fun_def = un_Fun1_def

fun subterms:: "('a, 'b) term  $\Rightarrow$  ('a, 'b) terms" where
  "subterms (Var x) = {Var x}"
| "subterms (Fun f T) = {Fun f T}  $\cup$  ( $\bigcup t \in \text{set } T. \text{subterms } t$ )"

abbreviation subterm_eq (infix <math>\sqsubseteq</math> 50) where "t'  $\sqsubseteq$  t  $\equiv$  (t'  $\in$  subterms t)"
abbreviation subterm (infix <math>\sqsubset</math> 50) where "t'  $\sqsubset$  t  $\equiv$  (t'  $\sqsubseteq$  t  $\wedge$  t'  $\neq$  t)"

abbreviation "subterms_set M  $\equiv$   $\bigcup$  (subterms ` M)"
abbreviation subterm_eq_set (infix <math>\sqsubseteq_{\text{set}}</math> 50) where "t  $\sqsubseteq_{\text{set}}$  M  $\equiv$  (t  $\in$  subterms_set M)"

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

fun fv_set where "fv_set M =  $\bigcup$  (fv ` M)"

abbreviation fv_pair where "fv_pair p  $\equiv$  case p of (t,t')  $\Rightarrow$  fv t  $\cup$  fv t'"

fun fv_pairs where "fv_pairs F =  $\bigcup$  (fv_pair ` set F)"

abbreviation ground where "ground M  $\equiv$  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 (FOG) = fv_list_pairs FO@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>

lemma `distinct_fv_list_Fun_param`:

```
assumes f: "distinct (fv_list (Fun f ts))"
```

```
and t: "t ∈ set ts"
```

```
shows "distinct (fv_list t)"
```

<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`: " $M \subseteq N \implies \text{fv}_{\text{set}} M \subseteq \text{fv}_{\text{set}} N$ "
 $\langle \text{proof} \rangle$

lemma `subterms_set_mono`: " $M \subseteq N \implies \text{subterms}_{\text{set}} M \subseteq \text{subterms}_{\text{set}} N$ "
 $\langle \text{proof} \rangle$

lemma `ground_empty[simp]`: "`ground {}`"
 $\langle \text{proof} \rangle$

lemma `ground_subset`: " $M \subseteq N \implies \text{ground } N \implies \text{ground } M$ "
 $\langle \text{proof} \rangle$

lemma `fv_map_fv_set`: " $\bigcup (\text{set } (\text{map } \text{fv } L)) = \text{fv}_{\text{set}} (\text{set } L)$ "
 $\langle \text{proof} \rangle$

lemma `fv_set_union`: " $\text{fv}_{\text{set}} (M \cup N) = \text{fv}_{\text{set}} M \cup \text{fv}_{\text{set}} N$ "
 $\langle \text{proof} \rangle$

lemma `finite_subset_Union`:
 fixes $A::\text{'a set}$ and $f::\text{'a} \implies \text{'b set}$
 assumes "`finite` $(\bigcup a \in A. f a)$ "
 shows " $\exists B. \text{finite } B \wedge B \subseteq A \wedge (\bigcup b \in B. f b) = (\bigcup a \in A. f a)$ "
 $\langle \text{proof} \rangle$

lemma `inv_set_fv`: "`finite` $M \implies \bigcup (\text{set } (\text{map } \text{fv } (\text{inv set } M))) = \text{fv}_{\text{set}} M$ "
 $\langle \text{proof} \rangle$

lemma `ground_subterm`: "`fv` $t = \{\}$ $\implies t' \sqsubseteq t \implies \text{fv } t' = \{\}$ " $\langle \text{proof} \rangle$

lemma `empty_fv_not_var`: "`fv` $t = \{\}$ $\implies t \neq \text{Var } x$ " $\langle \text{proof} \rangle$

lemma `empty_fv_exists_fun`: "`fv` $t = \{\}$ $\implies \exists f X. t = \text{Fun } f X$ " $\langle \text{proof} \rangle$

lemma `vars_iff_subtermeq`: " $x \in \text{fv } t \iff \text{Var } x \sqsubseteq t$ " $\langle \text{proof} \rangle$

lemma `vars_iff_subtermeq_set`: " $x \in \text{fv}_{\text{set}} M \iff \text{Var } x \in \text{subterms}_{\text{set}} M$ "
 $\langle \text{proof} \rangle$

lemma `vars_if_subtermeq_set`: " $\text{Var } x \in \text{subterms}_{\text{set}} M \implies x \in \text{fv}_{\text{set}} M$ "
 $\langle \text{proof} \rangle$

lemma `subtermeq_set_if_vars`: " $x \in \text{fv}_{\text{set}} M \implies \text{Var } x \in \text{subterms}_{\text{set}} M$ "
 $\langle \text{proof} \rangle$

lemma `vars_iff_subterm_or_eq`: " $x \in \text{fv } t \iff \text{Var } x \sqsubset t \vee \text{Var } x = t$ "
 $\langle \text{proof} \rangle$

lemma `var_is_subterm`: " $x \in \text{fv } t \implies \text{Var } x \in \text{subterms } t$ "
 $\langle \text{proof} \rangle$

lemma `subterm_is_var`: " $\text{Var } x \in \text{subterms } t \implies x \in \text{fv } t$ "
 $\langle \text{proof} \rangle$

lemma `no_var_subterm`: " $\neg t \sqsubset \text{Var } v$ " $\langle \text{proof} \rangle$

lemma `fun_if_subterm`: " $t \sqsubset u \implies \exists f X. u = \text{Fun } f X$ " $\langle \text{proof} \rangle$

lemma `subtermeq_vars_subset`: " $M \sqsubseteq N \implies \text{fv } M \subseteq \text{fv } N$ " $\langle \text{proof} \rangle$

lemma `fv_subterms[simp]`: " $\text{fv}_{\text{set}} (\text{subterms } t) = \text{fv } t$ "
 $\langle \text{proof} \rangle$

lemma `fv_subterms_set[simp]`: " $\text{fv}_{\text{set}} (\text{subterms}_{\text{set}} M) = \text{fv}_{\text{set}} M$ "

<proof>

lemma *fv_subset*: " $t \in M \implies \text{fv } t \subseteq \text{fv}_{\text{set}} M$ "

<proof>

lemma *fv_subset_subterms*: " $t \in \text{subterms}_{\text{set}} M \implies \text{fv } t \subseteq \text{fv}_{\text{set}} M$ "

<proof>

lemma *subterms_finite[simp]*: "*finite* (subterms *t*)" *<proof>*

lemma *subterms_union_finite*: "*finite* *M* \implies *finite* ($\bigcup t \in M. \text{subterms } t$)"

<proof>

lemma *subterms_subset*: " $t' \sqsubseteq t \implies \text{subterms } t' \subseteq \text{subterms } t$ "

<proof>

lemma *subterms_subset_set*: " $M \subseteq \text{subterms } t \implies \text{subterms}_{\text{set}} M \subseteq \text{subterms } t$ "

<proof>

lemma *subset_subterms_Union[simp]*: " $M \subseteq \text{subterms}_{\text{set}} M$ " *<proof>*

lemma *in_subterms_Union*: " $t \in M \implies t \in \text{subterms}_{\text{set}} M$ " *<proof>*

lemma *in_subterms_subset_Union*: " $t \in \text{subterms}_{\text{set}} M \implies \text{subterms } t \subseteq \text{subterms}_{\text{set}} M$ "

<proof>

lemma *subterm_param_split*:

assumes " $t \sqsubset \text{Fun } f X$ "

shows " $\exists \text{pre } x \text{ suf}. t \sqsubseteq x \wedge X = \text{pre}@x\#\text{suf}$ "

<proof>

lemma *ground_iff_no_vars*: "*ground* ($M :: ('a, 'b)$ terms) \longleftrightarrow ($\forall v. \text{Var } v \notin (\bigcup m \in M. \text{subterms } m)$)"

<proof>

lemma *index_Fun_subterms_subset[simp]*: " $i < \text{length } T \implies \text{subterms } (T ! i) \subseteq \text{subterms } (\text{Fun } f T)$ "

<proof>

lemma *index_Fun_fv_subset[simp]*: " $i < \text{length } T \implies \text{fv } (T ! i) \subseteq \text{fv } (\text{Fun } f T)$ "

<proof>

lemma *subterms_union_ground*:

assumes "*ground* *M*"

shows "*ground* ($\text{subterms}_{\text{set}} M$)"

<proof>

lemma *Var_subtermeq*: " $t \sqsubseteq \text{Var } v \implies t = \text{Var } v$ " *<proof>*

lemma *subtermeq_imp_funs_term_subset*: " $s \sqsubseteq t \implies \text{funs_term } s \subseteq \text{funs_term } t$ "

<proof>

lemma *subterms_const*: " $\text{subterms } (\text{Fun } f []) = \{\text{Fun } f []\}$ " *<proof>*

lemma *subterm_subtermeq_neq*: " $\llbracket t \sqsubset u; u \sqsubseteq v \rrbracket \implies t \neq v$ "

<proof>

lemma *subtermeq_subterm_neq*: " $\llbracket t \sqsubseteq u; u \sqsubset v \rrbracket \implies t \neq v$ "

<proof>

lemma *subterm_size_lt*: " $x \sqsubset y \implies \text{size } x < \text{size } y$ "

<proof>

lemma *in_subterms_eq*: " $\llbracket x \in \text{subterms } y; y \in \text{subterms } x \rrbracket \implies \text{subterms } x = \text{subterms } y$ "

<proof>

lemma *Fun_param_size_lt*:

" $t \in \text{set } ts \implies \text{size } t < \text{size } (\text{Fun } f \text{ } ts)$ "

<proof>

lemma *Fun_zip_size_lt*:

assumes " $(t,s) \in \text{set } (\text{zip } ts \text{ } ss)$ "

shows " $\text{size } t < \text{size } (\text{Fun } f \text{ } ts)$ "

and " $\text{size } s < \text{size } (\text{Fun } g \text{ } ss)$ "

<proof>

lemma *Fun_gt_params*: " $\text{Fun } f \text{ } X \notin (\bigcup x \in \text{set } X. \text{subterms } x)$ "

<proof>

lemma *params_subterms[simp]*: " $\text{set } X \subseteq \text{subterms } (\text{Fun } f \text{ } X)$ " *<proof>*

lemma *params_subterms_Union[simp]*: " $\text{subterms}_{\text{set}} (\text{set } X) \subseteq \text{subterms } (\text{Fun } f \text{ } X)$ " *<proof>*

lemma *Fun_subterm_inside_params*: " $t \sqsubseteq \text{Fun } f \text{ } X \iff t \in (\bigcup x \in (\text{set } X). \text{subterms } x)$ "

<proof>

lemma *Fun_param_is_subterm*: " $x \in \text{set } X \implies x \sqsubseteq \text{Fun } f \text{ } X$ "

<proof>

lemma *Fun_param_in_subterms*: " $x \in \text{set } X \implies x \in \text{subterms } (\text{Fun } f \text{ } X)$ "

<proof>

lemma *Fun_not_in_param*: " $x \in \text{set } X \implies \neg \text{Fun } f \text{ } X \sqsubseteq x$ "

<proof>

lemma *Fun_ex_if_subterm*: " $t \sqsubseteq s \implies \exists f \text{ } T. \text{Fun } f \text{ } T \sqsubseteq s \wedge t \in \text{set } T$ "

<proof>

lemma *const_subterm_obtain*:

assumes " $\text{fv } t = \{\}$ "

obtains c where " $\text{Fun } c \text{ } [] \sqsubseteq t$ "

<proof>

lemma *const_subterm_obtain'*: " $\text{fv } t = \{\} \implies \exists c. \text{Fun } c \text{ } [] \sqsubseteq t$ "

<proof>

lemma *subterms_singleton*:

assumes " $(\exists v. t = \text{Var } v) \vee (\exists f. t = \text{Fun } f \text{ } [])$ "

shows " $\text{subterms } t = \{t\}$ "

<proof>

lemma *subtermeq_Var_const*:

assumes " $s \sqsubseteq t$ "

shows " $t = \text{Var } v \implies s = \text{Var } v$ " " $t = \text{Fun } f \text{ } [] \implies s = \text{Fun } f \text{ } []$ "

<proof>

lemma *subterms_singleton'*:

assumes " $\text{subterms } t = \{t\}$ "

shows " $(\exists v. t = \text{Var } v) \vee (\exists f. t = \text{Fun } f \text{ } [])$ "

<proof>

lemma *funs_term_subterms_eq[simp]*:

" $(\bigcup s \in \text{subterms } t. \text{funs_term } s) = \text{funs_term } t$ "

" $(\bigcup s \in \text{subterms}_{\text{set}} M. \text{funs_term } s) = \bigcup (\text{funs_term } \text{` } M)$ "

<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 funs_term_eq_FunI:
  assumes "length T = length S" "∧i. i < length T ⇒ funs_term (T ! i) = funs_term (S ! i)"
  shows "funs_term (Fun f T) = funs_term (Fun f 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⟩

lemma in_Fun_fv_iff_in_args_nth_fv:
  "x ∈ fv (Fun f ts) ↔ (∃i < length ts. x ∈ fv (ts ! i))"
  (is "?A ↔ ?B")
⟨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

```

theory More_Unification
  imports Messages "First_Order_Terms.Unification"
begin

```

2.3.1 Substitutions

```

abbreviation subst_apply_list (infix <·list> 51) where
  "T ·list ∅ ≡ map (λt. t · ∅) T"

```

```

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

```

```

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

```

```

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

```

```

abbreviation subst_more_general_than (infixl <≤o> 50) where
  "σ ≤o ∅ ≡ ∃γ. ∅ = σ ∘s γ"

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma subst_support_def: "∅ supports τ ⟷ τ = ∅ ∘s τ"
⟨proof⟩

```

```

lemma subst_supportD: "∅ supports δ ⟹ ∅ ≤o 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 ·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?]: $"v \notin \text{fv } t \implies t \cdot s(v := u) = t \cdot s"$
 <proof>

lemma subst_idx_map:
 assumes $"\forall i \in \text{set } I. i < \text{length } T"$
 shows $"(\text{map } (!) T) I \cdot_{\text{list}} \delta = \text{map } (!) (\text{map } (\lambda t. t \cdot \delta) T) I"$
 <proof>

lemma subst_idx_map':
 assumes $"\forall i \in \text{fv}_{\text{set}} (\text{set } K). i < \text{length } T"$
 shows $"(K \cdot_{\text{list}} (!) T) \cdot_{\text{list}} \delta = K \cdot_{\text{list}} (!) (\text{map } (\lambda t. t \cdot \delta) T)"$ (is $"?A = ?B"$)
 <proof>

lemma subst_remove_var: $"v \notin \text{fv } s \implies v \notin \text{fv } (t \cdot \text{Var}(v := s))"$
 <proof>

lemma subst_set_map: $"x \in \text{set } X \implies x \cdot s \in \text{set } (\text{map } (\lambda x. x \cdot s) X)"$
 <proof>

lemma subst_set_idx_map:
 assumes $"\forall i \in I. i < \text{length } T"$
 shows $"(!) T \setminus I \cdot_{\text{set}} \delta = (!) (\text{map } (\lambda t. t \cdot \delta) T) \setminus I"$ (is $"?A = ?B"$)
 <proof>

lemma subst_set_idx_map':
 assumes $"\forall i \in \text{fv}_{\text{set}} K. i < \text{length } T"$
 shows $"K \cdot_{\text{set}} (!) T \cdot_{\text{set}} \delta = K \cdot_{\text{set}} (!) (\text{map } (\lambda t. t \cdot \delta) T)"$ (is $"?A = ?B"$)
 <proof>

lemma subst_term_list_obtain:
 assumes $"\forall i < \text{length } T. \exists s. P (T ! i) s \wedge S ! i = s \cdot \delta"$
 and $"\text{length } T = \text{length } S"$
 shows $"\exists U. \text{length } T = \text{length } U \wedge (\forall i < \text{length } T. P (T ! i) (U ! i)) \wedge S = \text{map } (\lambda u. u \cdot \delta) U"$
 <proof>

lemma subst_mono: $"t \sqsubseteq u \implies t \cdot s \sqsubseteq u \cdot s"$
 <proof>

lemma subst_mono_fv: $"x \in \text{fv } t \implies s x \sqsubseteq t \cdot s"$
 <proof>

lemma subst_mono_neq:
 assumes $"t \sqsubset u"$
 shows $"t \cdot s \sqsubset u \cdot s"$
 <proof>

lemma subst_no_occs[dest]: $"\neg \text{Var } v \sqsubseteq t \implies t \cdot \text{Var}(v := s) = t"$
 <proof>

lemma var_comp[simp]: $"\sigma \circ_s \text{Var} = \sigma"$ $"\text{Var} \circ_s \sigma = \sigma"$
 <proof>

lemma subst_comp_all: $"M \cdot_{\text{set}} (\delta \circ_s \vartheta) = (M \cdot_{\text{set}} \delta) \cdot_{\text{set}} \vartheta"$
 <proof>

lemma subst_all_mono: $"M \sqsubseteq M' \implies M \cdot_{\text{set}} s \sqsubseteq M' \cdot_{\text{set}} s"$
 <proof>

lemma subst_comp_set_image: $"(\delta \circ_s \vartheta) \setminus X = \delta \setminus X \cdot_{\text{set}} \vartheta"$
 <proof>

lemma subst_ground_ident[dest?]: $"\text{fv } t = \{\} \implies t \cdot s = t"$
 <proof>

lemma subst_ground_ident_compose:

"fv (σ x) = {} \implies ($\sigma \circ_s \vartheta$) x = σ x"

"fv (t \cdot σ) = {} \implies t \cdot ($\sigma \circ_s \vartheta$) = t \cdot σ "

<proof>

lemma subst_all_ground_ident[dest?]: "ground M \implies M \cdot_{set} s = M"

<proof>

lemma subst_cong: "[$\sigma = \sigma'$; $\vartheta = \vartheta'$] \implies ($\sigma \circ_s \vartheta$) = ($\sigma' \circ_s \vartheta'$)"

<proof>

lemma subst_mgt_bot[simp]: "Var \preceq_o ϑ "

<proof>

lemma subst_mgt_refl[simp]: " $\vartheta \preceq_o \vartheta$ "

<proof>

lemma subst_mgt_trans: "[$\vartheta \preceq_o \delta$; $\delta \preceq_o \sigma$] \implies $\vartheta \preceq_o \sigma$ "

<proof>

lemma subst_mgt_comp: " $\vartheta \preceq_o \vartheta \circ_s \delta$ "

<proof>

lemma subst_mgt_comp': " $\vartheta \circ_s \delta \preceq_o \sigma \implies \vartheta \preceq_o \sigma$ "

<proof>

lemma var_self: "($\lambda w.$ 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 ϑ = (Var v) \cdot σ) \implies ϑ = σ "

<proof>

lemma subst_all_empty[simp]: "{} \cdot_{set} ϑ = {}"

<proof>

lemma subst_all_insert: "(insert t M) \cdot_{set} δ = insert (t \cdot δ) (M \cdot_{set} δ)"

<proof>

lemma subst_apply_fv_subset: "fv t \subseteq V \implies fv (t \cdot δ) \subseteq fv_{set} ($\delta \setminus V$)"

<proof>

lemma subst_apply_fv_empty:

assumes "fv t = {}"

shows "fv (t \cdot σ) = {}"

<proof>

lemma subst_compose_fv:

assumes "fv (ϑ x) = {}"

shows "fv (($\vartheta \circ_s \sigma$) x) = {}"

<proof>

lemma subst_compose_fv':

fixes ϑ σ :: "('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 δ) = fv_{set} ($\delta \setminus$ fv t)"

<proof>

lemma `subst_apply_fv_unfold_set`: " $fv_{set} (\delta \setminus fv_{set} (set\ ts)) = fv_{set} (set\ ts \cdot_{set} \delta)$ "
 $\langle proof \rangle$

lemma `subst_apply_fv_unfold'`: " $fv (t \cdot \delta) = (\bigcup v \in fv\ t. fv (\delta\ v))$ "
 $\langle proof \rangle$

lemma `subst_apply_fv_union`: " $fv_{set} (\delta \setminus V) \cup fv (t \cdot \delta) = fv_{set} (\delta \setminus (V \cup fv\ t))$ "
 $\langle proof \rangle$

lemma `fv_set_subst`:
 $fv_{set} (M \cdot_{set} \vartheta) = fv_{set} (\vartheta \setminus fv_{set} M)$
 $\langle proof \rangle$

lemma `subst_list_set_fv`:
 $fv_{set} (set (ts \cdot_{list} \vartheta)) = fv_{set} (\vartheta \setminus fv_{set} (set\ ts))$
 $\langle proof \rangle$

lemma `subst_elimI[intro]`: " $(\bigwedge t. v \notin fv (t \cdot \sigma)) \implies subst_elim\ \sigma\ v$ "
 $\langle proof \rangle$

lemma `subst_elimI'[intro]`: " $(\bigwedge w. v \notin fv (Var\ w \cdot \vartheta)) \implies subst_elim\ \vartheta\ v$ "
 $\langle proof \rangle$

lemma `subst_elimD[dest]`: " $subst_elim\ \sigma\ v \implies v \notin fv (t \cdot \sigma)$ "
 $\langle proof \rangle$

lemma `subst_elimD'[dest]`: " $subst_elim\ \sigma\ v \implies \sigma\ v \neq Var\ v$ "
 $\langle proof \rangle$

lemma `subst_elimD''[dest]`: " $subst_elim\ \sigma\ v \implies v \notin fv (\sigma\ w)$ "
 $\langle proof \rangle$

lemma `subst_elim_rm_vars_dest[dest]`:
 $subst_elim (\sigma :: ('a, 'b)\ subst)\ v \implies v \notin vs \implies subst_elim (rm_vars\ vs\ \sigma)\ v$
 $\langle proof \rangle$

lemma `occs_subst_elim`: " $\neg Var\ v \sqsubseteq t \implies subst_elim (Var(v := t))\ v \vee (Var(v := t)) = Var$ "
 $\langle proof \rangle$

lemma `occs_subst_elim'`: " $\neg Var\ v \sqsubseteq t \implies subst_elim (Var(v := t))\ v$ "
 $\langle proof \rangle$

lemma `subst_elim_comp`: " $subst_elim\ \vartheta\ v \implies subst_elim (\delta \circ_s \vartheta)\ v$ "
 $\langle proof \rangle$

lemma `var_subst_idem`: " $subst_idem\ Var$ "
 $\langle proof \rangle$

lemma `var_upd_subst_idem`:
 $assumes\ \neg Var\ v \sqsubseteq t\ shows\ subst_idem (Var(v := t))$
 $\langle proof \rangle$

lemma `zip_map_subst`:
 $zip\ xs\ (xs \cdot_{list} \delta) = map (\lambda t. (t, t \cdot \delta))\ xs$
 $\langle proof \rangle$

lemma `map2_map_subst`:
 $map2\ f\ xs\ (xs \cdot_{list} \delta) = map (\lambda t. f\ t (t \cdot \delta))\ xs$
 $\langle proof \rangle$

2.3.2 Lemmata: Domain and Range of Substitutions

lemma `range_vars_alt_def`: "range_vars s \equiv 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) \implies finite (range_vars σ)"
 ⟨proof⟩

lemma `finite_subst_img_if_finite_dom'`: "finite (subst_domain σ) \implies finite (subst_range σ)"
 ⟨proof⟩

lemma `subst_img_alt_def`: "subst_range s = {t. $\exists v. s\ v = t \wedge t \neq \text{Var } v$ }"
 ⟨proof⟩

lemma `subst_fv_img_alt_def`: "range_vars s = ($\bigcup t \in \{t. \exists v. s\ v = t \wedge t \neq \text{Var } v\}. \text{fv } t$)"
 ⟨proof⟩

lemma `subst_domI[intro]`: " $\sigma\ v \neq \text{Var } v \implies v \in \text{subst_domain } \sigma$ "
 ⟨proof⟩

lemma `subst_imgI[intro]`: " $\sigma\ v \neq \text{Var } v \implies \sigma\ v \in \text{subst_range } \sigma$ "
 ⟨proof⟩

lemma `subst_fv_imgI[intro]`: " $\sigma\ v \neq \text{Var } v \implies \text{fv } (\sigma\ v) \subseteq \text{range_vars } \sigma$ "
 ⟨proof⟩

lemma `subst_eqI'`:
 assumes "t \cdot δ = t \cdot ϑ " "subst_domain δ = subst_domain ϑ " "subst_domain $\delta \subseteq \text{fv } t$ "
 shows " δ = ϑ "
 ⟨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 $\delta \subseteq \text{range Var}$ "
 shows "is_Fun t = is_Fun (t \cdot δ)"
 ⟨proof⟩

lemma `subst_fv_dom_img_subset`: "fv t \subseteq subst_domain $\vartheta \implies \text{fv } (t \cdot \vartheta) \subseteq \text{range_vars } \vartheta$ "
 ⟨proof⟩

lemma `subst_fv_dom_img_subset_set`: "fv_{set} M \subseteq subst_domain $\vartheta \implies \text{fv}_{\text{set}} (M \cdot_{\text{set}} \vartheta) \subseteq \text{range_vars } \vartheta$ "
 ⟨proof⟩

lemma `subst_fv_dom_ground_if_ground_img`:
 assumes "fv t \subseteq subst_domain s" "ground (subst_range s)"
 shows "fv (t \cdot s) = {}"

<proof>

lemma *subst_fv_dom_ground_if_ground_img'*:

assumes " $fv\ t \subseteq subst_domain\ s$ " " $\wedge x. x \in subst_domain\ s \implies fv\ (s\ x) = \{\}$ "
 shows " $fv\ (t \cdot s) = \{\}$ "

<proof>

lemma *subst_fv_unfold*: " $fv\ (t \cdot s) = (fv\ t - subst_domain\ s) \cup fv_{set}\ (s \setminus (fv\ t \cap subst_domain\ s))$ "

<proof>

lemma *subst_fv_unfold_ground_img*: " $range_vars\ s = \{\} \implies fv\ (t \cdot s) = fv\ t - subst_domain\ s$ "

<proof>

lemma *subst_img_update*:

" $[\sigma\ v = Var\ v; t \neq Var\ v] \implies range_vars\ (\sigma(v := t)) = range_vars\ \sigma \cup fv\ t$ "

<proof>

lemma *subst_dom_update1*: " $v \notin subst_domain\ \sigma \implies subst_domain\ (\sigma(v := Var\ v)) = subst_domain\ \sigma$ "

<proof>

lemma *subst_dom_update2*: " $t \neq Var\ v \implies subst_domain\ (\sigma(v := t)) = insert\ v\ (subst_domain\ \sigma)$ "

<proof>

lemma *subst_dom_update3*: " $t = Var\ v \implies subst_domain\ (\sigma(v := t)) = subst_domain\ \sigma - \{v\}$ "

<proof>

lemma *var_not_in_subst_dom[elim]*: " $v \notin subst_domain\ s \implies s\ v = Var\ v$ "

<proof>

lemma *subst_dom_vars_in_subst[elim]*: " $v \in subst_domain\ s \implies s\ v \neq Var\ v$ "

<proof>

lemma *subst_not_dom_fixed*: " $[[v \in fv\ t; v \notin subst_domain\ s] \implies v \in fv\ (t \cdot s)]$ " *<proof>*

lemma *subst_not_img_fixed*: " $[[v \in fv\ (t \cdot s); v \notin range_vars\ s] \implies v \in fv\ t]$ "

<proof>

lemma *ground_range_vars[intro]*: " $ground\ (subst_range\ s) \implies range_vars\ s = \{\}$ "

<proof>

lemma *ground_subst_no_var[intro]*: " $ground\ (subst_range\ s) \implies x \notin range_vars\ s$ "

<proof>

lemma *ground_img_obtain_fun*:

assumes " $ground\ (subst_range\ s)$ " " $x \in subst_domain\ s$ "

obtains $f\ T$ where " $s\ x = Fun\ f\ T$ " " $Fun\ f\ T \in subst_range\ s$ " " $fv\ (Fun\ f\ T) = \{\}$ "

<proof>

lemma *ground_term_subst_domain_fv_subset*:

" $fv\ (t \cdot \delta) = \{\} \implies fv\ t \subseteq subst_domain\ \delta$ "

<proof>

lemma *ground_subst_range_empty_fv*:

" $ground\ (subst_range\ \vartheta) \implies x \in subst_domain\ \vartheta \implies fv\ (\vartheta\ x) = \{\}$ "

<proof>

lemma *subst_Var_notin_img*: " $x \notin range_vars\ s \implies t \cdot s = Var\ x \implies t = Var\ x$ "

<proof>

lemma *fv_in_subst_img*: " $[[s\ v = t; t \neq Var\ v] \implies fv\ t \subseteq range_vars\ s]$ "

<proof>

lemma *empty_dom_iff_empty_subst*: " $subst_domain\ \vartheta = \{\} \iff \vartheta = Var$ " *<proof>*

lemma `subst_dom_cong`: " $(\bigwedge v t. \vartheta v = t \implies \delta v = t) \implies \text{subst_domain } \vartheta \subseteq \text{subst_domain } \delta$ "
 $\langle \text{proof} \rangle$

lemma `subst_img_cong`: " $(\bigwedge v t. \vartheta v = t \implies \delta v = t) \implies \text{range_vars } \vartheta \subseteq \text{range_vars } \delta$ "
 $\langle \text{proof} \rangle$

lemma `subst_dom_elim`: " $\text{subst_domain } s \cap \text{range_vars } s = \{\} \implies \text{fv } (t \cdot s) \cap \text{subst_domain } s = \{\}$ "
 $\langle \text{proof} \rangle$

lemma `subst_dom_insert_finite`: " $\text{finite } (\text{subst_domain } s) = \text{finite } (\text{subst_domain } (s(v := t)))$ "
 $\langle \text{proof} \rangle$

lemma `trm_subst_disj`: " $t \cdot \vartheta = t \implies \text{fv } t \cap \text{subst_domain } \vartheta = \{\}$ "
 $\langle \text{proof} \rangle$

declare `subst_apply_term_ident`[intro]

lemma `trm_subst_ident'`[intro]: " $v \notin \text{subst_domain } \vartheta \implies (\text{Var } v) \cdot \vartheta = \text{Var } v$ "
 $\langle \text{proof} \rangle$

lemma `trm_subst_ident''`[intro]: " $(\bigwedge x. x \in \text{fv } t \implies \vartheta x = \text{Var } x) \implies t \cdot \vartheta = t$ "
 $\langle \text{proof} \rangle$

lemma `set_subst_ident`: " $\text{fv}_{\text{set}} M \cap \text{subst_domain } \vartheta = \{\} \implies M \cdot_{\text{set}} \vartheta = M$ "
 $\langle \text{proof} \rangle$

lemma `trm_subst_ident_subterms`[intro]:
" $\text{fv } t \cap \text{subst_domain } \vartheta = \{\} \implies \text{subterms } t \cdot_{\text{set}} \vartheta = \text{subterms } t$ "
 $\langle \text{proof} \rangle$

lemma `trm_subst_ident_subterms'`[intro]:
" $v \notin \text{fv } t \implies \text{subterms } t \cdot_{\text{set}} \text{Var}(v := s) = \text{subterms } t$ "
 $\langle \text{proof} \rangle$

lemma `const_mem_subst_cases`:
assumes " $\text{Fun } c [] \in M \cdot_{\text{set}} \vartheta$ "
shows " $\text{Fun } c [] \in M \vee \text{Fun } c [] \in \vartheta \setminus \text{fv}_{\text{set}} M$ "
 $\langle \text{proof} \rangle$

lemma `const_mem_subst_cases'`:
assumes " $\text{Fun } c [] \in M \cdot_{\text{set}} \vartheta$ "
shows " $\text{Fun } c [] \in M \vee \text{Fun } c [] \in \text{subst_range } \vartheta$ "
 $\langle \text{proof} \rangle$

lemma `fv_subterms_substI`[intro]: " $y \in \text{fv } t \implies \vartheta y \in \text{subterms } t \cdot_{\text{set}} \vartheta$ "
 $\langle \text{proof} \rangle$

lemma `fv_subterms_subst_eq`[simp]: " $\text{fv}_{\text{set}} (\text{subterms } (t \cdot \vartheta)) = \text{fv}_{\text{set}} (\text{subterms } t \cdot_{\text{set}} \vartheta)$ "
 $\langle \text{proof} \rangle$

lemma `fv_subterms_set_subst`: " $\text{fv}_{\text{set}} (\text{subterms}_{\text{set}} M \cdot_{\text{set}} \vartheta) = \text{fv}_{\text{set}} (\text{subterms}_{\text{set}} (M \cdot_{\text{set}} \vartheta))$ "
 $\langle \text{proof} \rangle$

lemma `fv_subterms_set_subst'`: " $\text{fv}_{\text{set}} (\text{subterms}_{\text{set}} M \cdot_{\text{set}} \vartheta) = \text{fv}_{\text{set}} (M \cdot_{\text{set}} \vartheta)$ "
 $\langle \text{proof} \rangle$

lemma `fv_subst_subset`: " $x \in \text{fv } t \implies \text{fv } (\vartheta x) \subseteq \text{fv } (t \cdot \vartheta)$ "
 $\langle \text{proof} \rangle$

lemma `fv_subst_subset'`: " $\text{fv } s \subseteq \text{fv } t \implies \text{fv } (s \cdot \vartheta) \subseteq \text{fv } (t \cdot \vartheta)$ "
 $\langle \text{proof} \rangle$

```

lemma fv_subst_obtain_var:
  fixes  $\delta :: ('a, 'b) \text{subst}$ 
  assumes " $x \in \text{fv } (t \cdot \delta)$ "
  shows " $\exists y \in \text{fv } t. x \in \text{fv } (\delta y)$ "
<proof>

lemma set_subst_all_ident: " $\text{fv}_{\text{set}} (M \cdot_{\text{set}} \vartheta) \cap \text{subst\_domain } \delta = \{\} \implies M \cdot_{\text{set}} (\vartheta \circ_s \delta) = M \cdot_{\text{set}} \vartheta$ "
<proof>

lemma subterms_subst:
  " $\text{subterms } (t \cdot d) = (\text{subterms } t \cdot_{\text{set}} d) \cup \text{subterms}_{\text{set}} (d \setminus (\text{fv } t \cap \text{subst\_domain } d))$ "
<proof>

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

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

lemma subterms_subst_subterm:
  fixes  $\vartheta :: ('a, 'b) \text{subst}$ 
  assumes " $\forall x \in \text{fv } a. (\exists f. \vartheta x = \text{Fun } f []) \vee (\exists y. \vartheta x = \text{Var } y)$ "
  and " $b \in \text{subterms } (a \cdot \vartheta)$ "
  shows " $\exists c \in \text{subterms } a. c \cdot \vartheta = b$ "
<proof>

lemma subterms_subst_subset: " $\text{subterms } t \cdot_{\text{set}} \sigma \subseteq \text{subterms } (t \cdot \sigma)$ "
<proof>

lemma subterms_subst_subset': " $\text{subterms}_{\text{set}} M \cdot_{\text{set}} \sigma \subseteq \text{subterms}_{\text{set}} (M \cdot_{\text{set}} \sigma)$ "
<proof>

lemma subterms_set_subst:
  fixes  $\vartheta :: ('a, 'b) \text{subst}$ 
  assumes " $t \in \text{subterms}_{\text{set}} (M \cdot_{\text{set}} \vartheta)$ "
  shows " $t \in \text{subterms}_{\text{set}} M \cdot_{\text{set}} \vartheta \vee (\exists x \in \text{fv}_{\text{set}} M. t \in \text{subterms } (\vartheta x))$ "
<proof>

lemma rm_vars_dom: " $\text{subst\_domain } (\text{rm\_vars } V s) = \text{subst\_domain } s - V$ "
<proof>

lemma rm_vars_dom_subset: " $\text{subst\_domain } (\text{rm\_vars } V s) \subseteq \text{subst\_domain } s$ "
<proof>

lemma rm_vars_dom_eq':
  " $\text{subst\_domain } (\text{rm\_vars } (\text{UNIV} - V) s) = \text{subst\_domain } s \cap V$ "
<proof>

lemma rm_vars_dom_eqI:
  assumes " $t \cdot \delta = t \cdot \vartheta$ "
  shows " $\text{subst\_domain } (\text{rm\_vars } (\text{UNIV} - \text{fv } t) \delta) = \text{subst\_domain } (\text{rm\_vars } (\text{UNIV} - \text{fv } t) \vartheta)$ "
<proof>

lemma rm_vars_img: " $\text{subst\_range } (\text{rm\_vars } V s) = s \setminus \text{subst\_domain } (\text{rm\_vars } V s)$ "
<proof>

lemma rm_vars_img_subset: " $\text{subst\_range } (\text{rm\_vars } V s) \subseteq \text{subst\_range } s$ "

```

<proof>

lemma *rm_vars_img_fv_subset*: "range_vars (rm_vars V s) \subseteq range_vars s"

<proof>

lemma *rm_vars_fv_obtain*:

assumes "x \in fv (t · rm_vars X ϑ) - X"

shows " $\exists y \in$ fv t - X. x \in fv (rm_vars X ϑ y)"

<proof>

lemma *rm_vars_apply*: "v \in subst_domain (rm_vars V s) \implies (rm_vars V s) v = s v"

<proof>

lemma *rm_vars_apply'*: "subst_domain $\delta \cap$ vs = {} \implies rm_vars vs δ = δ "

<proof>

lemma *rm_vars_ident*: "fv t \cap vs = {} \implies t · (rm_vars vs ϑ) = t · ϑ "

<proof>

lemma *rm_vars_fv_subset*: "fv (t · rm_vars X ϑ) \subseteq fv t \cup fv (t · ϑ)"

<proof>

lemma *rm_vars_fv_disj*:

assumes "fv t \cap X = {}" "fv (t · ϑ) \cap X = {}"

shows "fv (t · rm_vars X ϑ) \cap 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 \cap X = {}" "X \cap range_vars ϑ = {}"

shows "fv (t · rm_vars X ϑ) \cap X = {}"

<proof>

lemma *subst_apply_dom_ident*: "t · ϑ = t \implies subst_domain $\delta \subseteq$ subst_domain $\vartheta \implies$ 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 $\delta \cap$ 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 $\delta \cap$ vs = {}"

shows "t · rm_vars vs ($\delta \circ_s \vartheta$) = t · (rm_vars vs $\delta \circ_s$ 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 ∘s σ) v = ϑ v"
⟨proof⟩

lemma ident_comp_subst_trm_if_disj': "fv (ϑ v) ∩ subst_domain σ = {} ⇒ (ϑ ∘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:*

" $x \notin \text{subst_domain } \vartheta1 \implies (\vartheta1 \circ_s \vartheta2) x = \vartheta2 x$ "

<proof>

lemma *subst_dom_comp_eq:*

assumes " $\text{subst_domain } \vartheta \cap \text{range_vars } \sigma = \{\}$ "

shows " $\text{subst_domain } (\vartheta \circ_s \sigma) = \text{subst_domain } \vartheta \cup \text{subst_domain } \sigma$ "

<proof>

lemma *subst_img_comp_subset[simp]:*

" $\text{range_vars } (\vartheta1 \circ_s \vartheta2) \subseteq \text{range_vars } \vartheta1 \cup \text{range_vars } \vartheta2$ "

<proof>

lemma *subst_img_comp_subset':*

assumes " $t \in \text{subst_range } (\vartheta1 \circ_s \vartheta2)$ "

shows " $t \in \text{subst_range } \vartheta2 \vee (\exists t' \in \text{subst_range } \vartheta1. t = t' \cdot \vartheta2)$ "

<proof>

lemma *subst_img_comp_subset'':*

" $\text{subterms}_{\text{set}} (\text{subst_range } (\vartheta1 \circ_s \vartheta2)) \subseteq$

$\text{subterms}_{\text{set}} (\text{subst_range } \vartheta2) \cup ((\text{subterms}_{\text{set}} (\text{subst_range } \vartheta1)) \cdot_{\text{set}} \vartheta2)$ "

<proof>

lemma *subst_img_comp_subset''':*

" $\text{subterms}_{\text{set}} (\text{subst_range } (\vartheta1 \circ_s \vartheta2)) - \text{range Var} \subseteq$

$\text{subterms}_{\text{set}} (\text{subst_range } \vartheta2) - \text{range Var} \cup ((\text{subterms}_{\text{set}} (\text{subst_range } \vartheta1) - \text{range Var}) \cdot_{\text{set}} \vartheta2)$ "

<proof>

lemma *subst_img_comp_subset_const:*

assumes " $\text{Fun } c [] \in \text{subst_range } (\vartheta1 \circ_s \vartheta2)$ "

shows " $\text{Fun } c [] \in \text{subst_range } \vartheta2 \vee \text{Fun } c [] \in \text{subst_range } \vartheta1 \vee$
 $(\exists x. \text{Var } x \in \text{subst_range } \vartheta1 \wedge \vartheta2 x = \text{Fun } c [])$ "

<proof>

lemma *subst_img_comp_subset_const':*

fixes $\delta \tau :: ('f, 'v) \text{subst}$

assumes " $(\delta \circ_s \tau) x = \text{Fun } c []$ "

shows " $\delta x = \text{Fun } c [] \vee (\exists z. \delta x = \text{Var } z \wedge \tau z = \text{Fun } c [])$ "

<proof>

lemma *subst_img_comp_subset_ground:*

assumes " $\text{ground } (\text{subst_range } \vartheta1)$ "

shows " $\text{subst_range } (\vartheta1 \circ_s \vartheta2) \subseteq \text{subst_range } \vartheta1 \cup \text{subst_range } \vartheta2$ "

<proof>

lemma *subst_fv_dom_img_single:*

assumes " $v \notin \text{fv } t$ " " $\sigma v = t$ " " $\bigwedge w. v \neq w \implies \sigma w = \text{Var } w$ "

shows " $\text{subst_domain } \sigma = \{v\}$ " " $\text{range_vars } \sigma = \text{fv } t$ "

<proof>

lemma *subst_comp_upd1:*

" $\vartheta(v := t) \circ_s \sigma = (\vartheta \circ_s \sigma)(v := t \cdot \sigma)$ "

<proof>

lemma *subst_comp_upd2:*

assumes " $v \notin \text{subst_domain } s$ " " $v \notin \text{range_vars } s$ "

shows " $s(v := t) = s \circ_s (\text{Var}(v := t))$ "

<proof>

lemma *ground_subst_dom_iff_img:*

" $\text{ground } (\text{subst_range } \sigma) \implies x \in \text{subst_domain } \sigma \iff \sigma x \in \text{subst_range } \sigma$ "

<proof>

lemma *finite_dom_subst_exists:*

"finite $S \implies \exists \sigma :: ('f, 'v)$ subst. subst_domain $\sigma = S$ "

<proof>

lemma *subst_inj_is_bij_betw_dom_img_if_ground_img:*

assumes "ground (subst_range σ)"

shows "inj $\sigma \longleftrightarrow$ bij_betw σ (subst_domain σ) (subst_range σ)" (is "?A \longleftrightarrow ?B")

<proof>

lemma *bij_finite_ground_subst_exists:*

assumes "finite ($S :: 'v$ set)" "infinite ($U :: ('f, 'v)$ term set)" "ground U "

shows " $\exists \sigma :: ('f, 'v)$ subst. subst_domain $\sigma = S$

\wedge bij_betw σ (subst_domain σ) (subst_range σ)

\wedge subst_range $\sigma \subseteq U$ "

<proof>

lemma *bij_finite_const_subst_exists:*

assumes "finite ($S :: 'v$ set)" "finite ($T :: 'f$ set)" "infinite ($U :: 'f$ set)"

shows " $\exists \sigma :: ('f, 'v)$ subst. subst_domain $\sigma = S$

\wedge bij_betw σ (subst_domain σ) (subst_range σ)

\wedge subst_range $\sigma \subseteq (\lambda c. \text{Fun } c \text{ []}) \setminus (U - T)$ "

<proof>

lemma *bij_finite_const_subst_exists':*

assumes "finite ($S :: 'v$ set)" "finite ($T :: ('f, 'v)$ terms)" "infinite ($U :: 'f$ set)"

shows " $\exists \sigma :: ('f, 'v)$ subst. subst_domain $\sigma = S$

\wedge bij_betw σ (subst_domain σ) (subst_range σ)

\wedge subst_range $\sigma \subseteq ((\lambda c. \text{Fun } c \text{ []}) \setminus U) - T$ "

<proof>

lemma *bij_betw_iteI:*

assumes "bij_betw f A B " "bij_betw g C D " " $A \cap C = \{\}$ " " $B \cap D = \{\}$ "

shows "bij_betw ($\lambda x. \text{if } x \in A \text{ then } f \ x \text{ else } g \ x$) ($A \cup C$) ($B \cup D$)"

<proof>

lemma *subst_comp_split:*

assumes "subst_domain $\vartheta \cap$ range_vars $\vartheta = \{\}$ "

shows " $\vartheta = (\text{rm_vars (subst_domain } \vartheta - V) \vartheta) \circ_s (\text{rm_vars } V \vartheta)$ " (is ?P)

and " $\vartheta = (\text{rm_vars } V \vartheta) \circ_s (\text{rm_vars (subst_domain } \vartheta - V) \vartheta)$ " (is ?Q)

<proof>

lemma *subst_comp_eq_if_disjoint_vars:*

assumes "(subst_domain $\delta \cup$ range_vars $\delta) \cap$ (subst_domain $\gamma \cup$ range_vars $\gamma) = \{\}$ "

shows " $\gamma \circ_s \delta = \delta \circ_s \gamma$ "

<proof>

lemma *subst_eq_if_disjoint_vars_ground:*

fixes $\xi \delta :: ('f, 'v)$ subst"

assumes "subst_domain $\delta \cap$ subst_domain $\xi = \{\}$ " "ground (subst_range ξ)" "ground (subst_range δ)"

shows " $t \cdot \delta \cdot \xi = t \cdot \xi \cdot \delta$ "

<proof>

lemma *subst_img_bound:* "subst_domain $\delta \cup$ range_vars $\delta \subseteq$ fv $t \implies$ range_vars $\delta \subseteq$ fv ($t \cdot \delta$)"

<proof>

lemma *subst_all_fv_subset:* "fv $t \subseteq$ fv_{set} $M \implies$ fv ($t \cdot \vartheta$) \subseteq fv_{set} ($M \cdot_{\text{set}} \vartheta$)"

<proof>

lemma *subst_support_if_mgt_subst_idem:*

assumes " $\vartheta \preceq_o \delta$ " "subst_idem ϑ "

shows " ϑ supports δ "

<proof>

lemma *subst_support_iff_mgt_if_subst_idem*:

assumes "subst_idem ϑ "

shows " $\vartheta \preceq_o \delta \longleftrightarrow \vartheta$ supports δ "

<proof>

lemma *subst_support_comp*:

fixes $\vartheta \delta \mathcal{I} :: ('a, 'b)$ subst"

assumes " ϑ supports \mathcal{I} " " δ supports \mathcal{I} "

shows " $(\vartheta \circ_s \delta)$ supports \mathcal{I} "

<proof>

lemma *subst_support_comp'*:

fixes $\vartheta \delta \sigma :: ('a, 'b)$ subst"

assumes " ϑ supports δ "

shows " ϑ supports $(\delta \circ_s \sigma)$ " " σ supports $\delta \implies \vartheta$ supports $(\sigma \circ_s \delta)$ "

<proof>

lemma *subst_support_comp_split*:

fixes $\vartheta \delta \mathcal{I} :: ('a, 'b)$ subst"

assumes " $(\vartheta \circ_s \delta)$ supports \mathcal{I} "

shows "subst_domain $\vartheta \cap$ range_vars $\vartheta = \{\}$ $\implies \vartheta$ supports \mathcal{I} "

and "subst_domain $\vartheta \cap$ subst_domain $\delta = \{\}$ $\implies \delta$ supports \mathcal{I} "

<proof>

lemma *subst_idem_support*: "subst_idem $\vartheta \implies \vartheta$ supports $\vartheta \circ_s \delta$ "

<proof>

lemma *subst_idem_iff_self_support*: "subst_idem $\vartheta \longleftrightarrow \vartheta$ supports ϑ "

<proof>

lemma *subterm_subst_neq*: " $t \sqsubseteq t' \implies t \cdot s \neq t' \cdot s$ "

<proof>

lemma *fv_Fun_subst_neq*: " $x \in \text{fv} (\text{Fun } f \ T) \implies \sigma \ x \neq \text{Fun } f \ T \cdot \sigma$ "

<proof>

lemma *subterm_subst_unfold*:

assumes " $t \sqsubseteq s \cdot \vartheta$ "

shows " $(\exists s'. s' \sqsubseteq s \wedge t = s' \cdot \vartheta) \vee (\exists x \in \text{fv } s. t \sqsubseteq \vartheta \ x)$ "

<proof>

lemma *subterm_subst_img_subterm*:

assumes " $t \sqsubseteq s \cdot \vartheta$ " " $\bigwedge s'. s' \sqsubseteq s \implies t \neq s' \cdot \vartheta$ "

shows " $\exists w \in \text{fv } s. t \sqsubseteq \vartheta \ w$ "

<proof>

lemma *subterm_subst_not_img_subterm*:

assumes " $t \sqsubseteq s \cdot \mathcal{I}$ " " $\neg(\exists w \in \text{fv } s. t \sqsubseteq \mathcal{I} \ w)$ "

shows " $\exists f \ T. \text{Fun } f \ T \sqsubseteq s \wedge t = \text{Fun } f \ T \cdot \mathcal{I}$ "

<proof>

lemma *subst_apply_img_var*:

assumes " $v \in \text{fv} (t \cdot \delta)$ " " $v \notin \text{fv } t$ "

obtains w where " $w \in \text{fv } t$ " " $v \in \text{fv} (\delta \ w)$ "

<proof>

lemma *subst_apply_img_var'*:

assumes " $x \in \text{fv} (t \cdot \delta)$ " " $x \notin \text{fv } t$ "

shows " $\exists y \in \text{fv } t. x \in \text{fv} (\delta \ y)$ "

<proof>

lemma nth_map_subst:

fixes $\vartheta::('f, 'v) \text{ subst}$ and $T::('f, 'v) \text{ term list}$ and $i::\text{nat}$
 shows " $i < \text{length } T \implies (\text{map } (\lambda t. t \cdot \vartheta) T) ! i = (T ! i) \cdot \vartheta$ "

<proof>

lemma subst_subterm:

assumes " $\text{Fun } f T \sqsubseteq t \cdot \vartheta$ "
 shows " $(\exists S. \text{Fun } f S \sqsubseteq t \wedge \text{Fun } f S \cdot \vartheta = \text{Fun } f T) \vee$
 $(\exists s \in \text{subst_range } \vartheta. \text{Fun } f T \sqsubseteq s)$ "

<proof>

lemma subst_subterm':

assumes " $\text{Fun } f T \sqsubseteq t \cdot \vartheta$ "
 shows " $\exists S. \text{length } S = \text{length } T \wedge (\text{Fun } f S \sqsubseteq t \vee (\exists s \in \text{subst_range } \vartheta. \text{Fun } f S \sqsubseteq s))$ "

<proof>

lemma subst_subterm'':

assumes " $s \in \text{subterms } (t \cdot \vartheta)$ "
 shows " $(\exists u \in \text{subterms } t. s = u \cdot \vartheta) \vee s \in \text{subterms}_{\text{set}} (\text{subst_range } \vartheta)$ "

<proof>

lemma fv_ground_subst_compose:

assumes " $\text{subst_domain } \delta = \text{subst_domain } \sigma$ "
 and " $\text{range_vars } \delta = \{\}$ " " $\text{range_vars } \sigma = \{\}$ "
 shows " $\text{fv } (t \cdot \delta \circ_s \vartheta) = \text{fv } (t \cdot \sigma \circ_s \vartheta)$ "

<proof>

2.3.3 More Small Lemmata

lemma funs_term_subst: " $\text{funs_term } (t \cdot \vartheta) = \text{funs_term } t \cup (\bigcup x \in \text{fv } t. \text{funs_term } (\vartheta x))$ "

<proof>

lemma fv_set_subst_img_eq:

assumes " $X \cap (\text{subst_domain } \delta \cup \text{range_vars } \delta) = \{\}$ "
 shows " $\text{fv}_{\text{set}} (\delta \cdot (Y - X)) = \text{fv}_{\text{set}} (\delta \cdot Y) - X$ "

<proof>

lemma subst_Fun_index_eq:

assumes " $i < \text{length } T$ " " $\text{Fun } f T \cdot \delta = \text{Fun } g T' \cdot \delta$ "
 shows " $T ! i \cdot \delta = T' ! i \cdot \delta$ "

<proof>

lemma fv_exists_if_unifiable_and_neq:

fixes $t t'::('a, 'b) \text{ term}$ and $\delta \vartheta::('a, 'b) \text{ subst}$
 assumes " $t \neq t'$ " " $t \cdot \vartheta = t' \cdot \vartheta$ "
 shows " $\text{fv } t \cup \text{fv } t' \neq \{\}$ "

<proof>

lemma const_subterm_subst: " $\text{Fun } c [] \sqsubseteq t \implies \text{Fun } c [] \sqsubseteq t \cdot \sigma$ "

<proof>

lemma const_subterm_subst_var_obtain:

assumes " $\text{Fun } c [] \sqsubseteq t \cdot \sigma$ " " $\neg \text{Fun } c [] \sqsubseteq t$ "
 obtains x where " $x \in \text{fv } t$ " " $\text{Fun } c [] \sqsubseteq \sigma x$ "

<proof>

lemma const_subterm_subst_cases:

assumes " $\text{Fun } c [] \sqsubseteq t \cdot \sigma$ "
 shows " $\text{Fun } c [] \sqsubseteq t \vee (\exists x \in \text{fv } t. x \in \text{subst_domain } \sigma \wedge \text{Fun } c [] \sqsubseteq \sigma x)$ "

<proof>

lemma const_subterms_subst_cases:

assumes " $\text{Fun } c [] \sqsubseteq_{\text{set}} M \cdot_{\text{set}} \sigma$ "

shows "Fun c [] $\sqsubseteq_{set} M \vee (\exists x \in fv_{set} M. x \in subst_domain \sigma \wedge Fun\ c\ [] \sqsubseteq \sigma\ x)$ "
 <proof>

lemma const_subterms_subst_cases':
assumes "Fun c [] $\sqsubseteq_{set} M \cdot_{set} \sigma$ "
shows "Fun c [] $\sqsubseteq_{set} M \vee Fun\ c\ [] \sqsubseteq_{set} subst_range\ \sigma$ "
 <proof>

lemma fv_pairs_subst_fv_subset:
assumes "x $\in fv_{pairs} F$ "
shows "fv ($\vartheta\ x$) $\subseteq fv_{pairs} (F \cdot_{pairs} \vartheta)$ "
 <proof>

lemma fv_pairs_step_subst: "fv_{set} ($\delta \cdot fv_{pairs} F$) = fv_{pairs} (F $\cdot_{pairs} \delta$)"
 <proof>

lemma fv_pairs_subst_obtain_var:
fixes $\delta :: ('a, 'b) subst$
assumes "x $\in fv_{pairs} (F \cdot_{pairs} \delta)$ "
shows " $\exists y \in fv_{pairs} F. x \in fv (\delta\ y)$ "
 <proof>

lemma pair_subst_ident[*intro*]: "(fv t \cup fv t') $\cap subst_domain\ \vartheta = \{\}$ $\implies (t, t') \cdot_p \vartheta = (t, t')$ "
 <proof>

lemma pairs_substI[*intro*]:
assumes "subst_{domain} $\vartheta \cap (\bigcup (s, t) \in M. fv\ s \cup fv\ t) = \{\}$ "
shows "M $\cdot_{pset} \vartheta = M$ "
 <proof>

lemma fv_pairs_subst: "fv_{pairs} (F $\cdot_{pairs} \vartheta$) = fv_{set} ($\vartheta \cdot (fv_{pairs} F)$)"
 <proof>

lemma fv_pairs_subst_subset:
assumes "fv_{pairs} (F $\cdot_{pairs} \delta$) $\subseteq subst_domain\ \sigma$ "
shows "fv_{pairs} F $\subseteq subst_domain\ \sigma \cup subst_domain\ \delta$ "
 <proof>

lemma pairs_subst_comp: "F $\cdot_{pairs} \delta \circ_s \vartheta = ((F \cdot_{pairs} \delta) \cdot_{pairs} \vartheta)$ "
 <proof>

lemma pairs_substI'[*intro*]:
 "subst_{domain} $\vartheta \cap fv_{pairs} F = \{\}$ $\implies F \cdot_{pairs} \vartheta = F$ "
 <proof>

lemma subst_pair_compose[*simp*]: "d $\cdot_p (\delta \circ_s \mathcal{I}) = d \cdot_p \delta \cdot_p \mathcal{I}$ "
 <proof>

lemma subst_pairs_compose[*simp*]: "D $\cdot_{pset} (\delta \circ_s \mathcal{I}) = D \cdot_{pset} \delta \cdot_{pset} \mathcal{I}$ "
 <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*]: "[] $\cdot_{pairs} \delta = []$ "
 <proof>

lemma subst_apply_pairs_singleton[*simp*]: "[(t, s)] $\cdot_{pairs} \delta = [(t \cdot \delta, s \cdot \delta)]$ "
 <proof>

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

lemma subst_apply_pairs_pset_subst: "set (F $\cdot_{pairs} \vartheta$) = set F $\cdot_{pset} \vartheta$ "

<proof>

lemma subst_subterms:

" $t \sqsubseteq_{set} M \implies t \cdot \vartheta \sqsubseteq_{set} M \cdot_{set} \vartheta$ "

<proof>

lemma subst_subterms_fv:

" $x \in fv_{set} M \implies \vartheta x \in subterms_{set} M \cdot_{set} \vartheta$ "

<proof>

lemma subst_subterms_Var:

" $\text{Var } x \sqsubseteq_{set} M \implies \vartheta x \in subterms_{set} M \cdot_{set} \vartheta$ "

<proof>

lemma fv_subset_subterms_subset:

" $\delta \cdot fv_{set} M \subseteq subterms_{set} M \cdot_{set} \delta$ "

<proof>

lemma subst_const_swap_eq:

fixes $\vartheta \sigma :: ('a, 'b) \text{ subst}$

assumes $t: "t \cdot \vartheta = s \cdot \vartheta"$

and $\vartheta: "\forall x \in fv\ t \cup fv\ s. \exists k. \vartheta\ x = \text{Fun } k\ []"$

" $\forall x \in fv\ t. \neg(\vartheta\ x \sqsubseteq s)$ "

" $\forall x \in fv\ s. \neg(\vartheta\ x \sqsubseteq t)$ "

and $\sigma_def: "\sigma \equiv \lambda x. p\ (\vartheta\ x)"$

shows " $t \cdot \sigma = s \cdot \sigma$ "

<proof>

lemma term_subst_set_eq:

assumes " $\bigwedge x. x \in fv_{set} M \implies \delta\ x = \sigma\ x$ "

shows " $M \cdot_{set} \delta = M \cdot_{set} \sigma$ "

<proof>

lemma subst_const_swap_eq':

assumes " $t \cdot \vartheta = s \cdot \vartheta$ "

and " $\forall x \in fv\ t \cup fv\ s. \vartheta\ x = \sigma\ x \vee \neg(\vartheta\ x \sqsubseteq t) \wedge \neg(\vartheta\ x \sqsubseteq s)$ " (is "?A t s")

and " $\forall x \in fv\ t \cup fv\ s. \exists c. \vartheta\ x = \text{Fun } c\ []$ " (is "?B t s")

and " $\forall x \in fv\ t \cup fv\ s. \exists c. \sigma\ x = \text{Fun } c\ []$ " (is "?C t s")

and " $\forall x \in fv\ t \cup fv\ s. \forall y \in fv\ t \cup fv\ s. \vartheta\ x = \vartheta\ y \longleftrightarrow \sigma\ x = \sigma\ y$ " (is "?D t s")

shows " $t \cdot \sigma = s \cdot \sigma$ "

<proof>

lemma subst_const_swap_eq_mem:

assumes " $t \cdot \vartheta \in M \cdot_{set} \vartheta$ "

and " $\forall x \in fv_{set} M \cup fv\ t. \vartheta\ x = \sigma\ x \vee \neg(\vartheta\ x \sqsubseteq_{set} \text{insert } t\ M)$ "

and " $\forall x \in fv_{set} M \cup fv\ t. \exists c. \vartheta\ x = \text{Fun } c\ []$ " (is "?B (fv_{set} M \cup fv t)")

and " $\forall x \in fv_{set} M \cup fv\ t. \exists c. \sigma\ x = \text{Fun } c\ []$ " (is "?C (fv_{set} M \cup fv t)")

and " $\forall x \in fv_{set} M \cup fv\ t. \forall y \in fv_{set} M \cup fv\ t. \vartheta\ x = \vartheta\ y \longleftrightarrow \sigma\ x = \sigma\ y$ " (is "?D (fv_{set} M \cup fv t)")

shows " $t \cdot \sigma \in M \cdot_{set} \sigma$ "

<proof>

2.3.4 Finite Substitutions

inductive_set $fsubst :: ('a, 'b) \text{ subst set}$ where

fvar: " $\text{Var} \in fsubst$ "

| FUpdate: " $[\vartheta \in fsubst; v \notin \text{subst_domain } \vartheta; t \neq \text{Var } v] \implies \vartheta(v := t) \in fsubst$ "

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 " $\bigwedge \vartheta v t. \llbracket \text{finite (subst_domain } \vartheta); v \notin \text{subst_domain } \vartheta; t \neq \text{Var } v; P \vartheta \rrbracket \implies P (\vartheta(v := t))$ "
 shows "P δ "

<proof>

lemma fun_upd_fsubst: " $s(v := t) \in \text{fsubst} \iff s \in \text{fsubst}$ "

<proof>

lemma finite_img_if_fsubst: " $s \in \text{fsubst} \implies \text{finite (subst_range } s)$ "

<proof>

2.3.5 Unifiers and Most General Unifiers (MGUs)

abbreviation Unifier::" $('f, 'v) \text{subst} \Rightarrow ('f, 'v) \text{term} \Rightarrow ('f, 'v) \text{term} \Rightarrow \text{bool}$ " where
 "Unifier $\sigma t u \equiv (t \cdot \sigma = u \cdot \sigma)$ "

abbreviation MGU::" $('f, 'v) \text{subst} \Rightarrow ('f, 'v) \text{term} \Rightarrow ('f, 'v) \text{term} \Rightarrow \text{bool}$ " where

"MGU $\sigma t u \equiv \text{Unifier } \sigma t u \wedge (\forall \vartheta. \text{Unifier } \vartheta t u \longrightarrow \sigma \preceq_o \vartheta)$ "

lemma MGUI[*intro*]:

shows " $\llbracket t \cdot \sigma = u \cdot \sigma; \bigwedge \vartheta::('f, 'v) \text{subst}. t \cdot \vartheta = u \cdot \vartheta \implies \sigma \preceq_o \vartheta \rrbracket \implies \text{MGU } \sigma t u$ "

<proof>

lemma UnifierD[*dest*]:

fixes $\sigma::('f, 'v) \text{subst}$ and $f g::'f$ and $X Y::('f, 'v) \text{term list}$

assumes "Unifier $\sigma (\text{Fun } f X) (\text{Fun } g Y)$ "

shows " $f = g$ " "length $X = \text{length } Y$ "

<proof>

lemma MGUD[*dest*]:

fixes $\sigma::('f, 'v) \text{subst}$ and $f g::'f$ and $X Y::('f, 'v) \text{term list}$

assumes "MGU $\sigma (\text{Fun } f X) (\text{Fun } g Y)$ "

shows " $f = g$ " "length $X = \text{length } Y$ "

<proof>

lemma MGU_sym[*sym*]: "MGU $\sigma s t \implies \text{MGU } \sigma t s$ " *<proof>*

lemma Unifier_sym[*sym*]: "Unifier $\sigma s t \implies \text{Unifier } \sigma t s$ " *<proof>*

lemma MGU_nil: "MGU Var $s t \iff s = t$ " *<proof>*

lemma Unifier_comp: "Unifier $(\vartheta \circ_s \delta) t u \implies \text{Unifier } \delta (t \cdot \vartheta) (u \cdot \vartheta)$ "

<proof>

lemma Unifier_comp': "Unifier $\delta (t \cdot \vartheta) (u \cdot \vartheta) \implies \text{Unifier } (\vartheta \circ_s \delta) t u$ "

<proof>

lemma Unifier_excludes_subterm:

assumes $\vartheta: \text{Unifier } \vartheta t u$

shows " $\neg t \sqsubset u$ "

<proof>

lemma MGU_is_Unifier: "MGU $\sigma t u \implies \text{Unifier } \sigma t u$ " *<proof>*

lemma MGU_Var1:

assumes " $\neg \text{Var } v \sqsubset t$ "

shows "MGU $(\text{Var}(v := t)) (\text{Var } v) t$ "

<proof>

lemma MGU_Var2: " $v \notin \text{fv } t \implies \text{MGU } (\text{Var}(v := t)) (\text{Var } v) t$ "

<proof>

lemma MGU_Var3: "MGU Var $(\text{Var } v) (\text{Var } w) \iff v = w$ " *<proof>*

lemma *MGU_Const1*: "MGU Var (Fun c []) (Fun d []) \longleftrightarrow c = d" *<proof>*

lemma *MGU_Const2*: "MGU ϑ (Fun c []) (Fun d []) \implies 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 \implies Unifier s (t · r) (u · r) \implies Unifier (r \circ_s s) (t · r) (u · r)"
<proof>

lemma *subst_idem_comp*:
 "subst_idem r \implies Unifier s (t · r) (u · r) \implies
 (\wedge q. Unifier q (t · r) (u · r) \implies s \circ_s q = q) \implies
 subst_idem (r \circ_s s)"
<proof>

lemma *Unifier_mgt*: "[Unifier δ t u; $\delta \preceq_o \vartheta$] \implies Unifier ϑ t u" *<proof>*

lemma *Unifier_support*: "[Unifier δ t u; δ supports ϑ] \implies Unifier ϑ t u"
<proof>

lemma *MGU_mgt*: "[MGU σ t u; MGU δ t u] \implies $\sigma \preceq_o \delta$ " *<proof>*

lemma *Unifier_trm_fv_bound*:
 "[Unifier s t u; v \in fv t] \implies 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] \implies 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 \cup fv t' \subseteq 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 \in wf_subst_set"
 | Insert[simp]:
 "[$\vartheta \in$ wf_subst_set; v \notin subst_domain ϑ ;
 v \notin range_vars ϑ ; fv t \cap (insert v (subst_domain ϑ)) = {}]
 $\implies \vartheta(v := t) \in$ wf_subst_set"

definition *wf_subst*:"('a,'b) subst \implies bool" where
 "wf_subst $\vartheta \equiv$ subst_domain $\vartheta \cap$ range_vars $\vartheta = \{\}$ \wedge finite (subst_domain ϑ)"

definition *wf_MGU*:"('a,'b) subst \implies ('a,'b) term \implies ('a,'b) term \implies bool" where
 "wf_MGU ϑ s t \equiv wf_subst $\vartheta \wedge$ MGU ϑ s t \wedge subst_domain $\vartheta \cup$ range_vars $\vartheta \subseteq$ fv s \cup fv t"

```

lemma wf_subst_subst_idem: "wf_subst  $\vartheta \implies$  subst_idem  $\vartheta$ " <proof>

lemma wf_subst_properties: " $\vartheta \in$  wf_subst_set = wf_subst  $\vartheta$ "
<proof>

lemma wf_subst_induct[consumes 1, case_names Empty Insert]:
  assumes "wf_subst  $\delta$ " "P Var"
  and " $\wedge \vartheta v t. \llbracket$  wf_subst  $\vartheta$ ; P  $\vartheta$ ;  $v \notin$  subst_domain  $\vartheta$ ;  $v \notin$  range_vars  $\vartheta$ ;
    fv t  $\cap$  insert v (subst_domain  $\vartheta$ ) =  $\{\}$   $\rrbracket$ 
     $\implies$  P ( $\vartheta(v := t)$ )"
  shows "P  $\delta$ "
<proof>

lemma wf_subst_fsubst: "wf_subst  $\delta \implies \delta \in$  fsubst"
<proof>

lemma wf_subst_nil: "wf_subst Var" <proof>

lemma wf_MGU_nil: "MGU Var s t  $\implies$  wf_MGU Var s t"
<proof>

lemma wf_MGU_dom_bound: "wf_MGU  $\vartheta$  s t  $\implies$  subst_domain  $\vartheta \subseteq$  fv s  $\cup$  fv t" <proof>

lemma wf_subst_single:
  assumes " $v \notin$  fv t" " $\sigma v = t$ " " $\wedge w. v \neq w \implies \sigma w =$  Var  $w$ "
  shows "wf_subst  $\sigma$ "
<proof>

lemma wf_subst_reduction:
  "wf_subst s  $\implies$  wf_subst (rm_var v s)"
<proof>

lemma wf_subst_compose:
  assumes "wf_subst  $\vartheta1$ " "wf_subst  $\vartheta2$ "
  and "subst_domain  $\vartheta1 \cap$  subst_domain  $\vartheta2 = \{\}$ "
  and "subst_domain  $\vartheta1 \cap$  range_vars  $\vartheta2 = \{\}$ "
  shows "wf_subst ( $\vartheta1 \circ_s \vartheta2$ )"
<proof>

lemma wf_subst_append:
  fixes  $\vartheta1 \vartheta2::('f, 'v)$  subst"
  assumes "wf_subst  $\vartheta1$ " "wf_subst  $\vartheta2$ "
  and "subst_domain  $\vartheta1 \cap$  subst_domain  $\vartheta2 = \{\}$ "
  and "subst_domain  $\vartheta1 \cap$  range_vars  $\vartheta2 = \{\}$ "
  and "range_vars  $\vartheta1 \cap$  subst_domain  $\vartheta2 = \{\}$ "
  shows "wf_subst ( $\lambda v. \text{if } \vartheta1 v = \text{Var } v \text{ then } \vartheta2 v \text{ else } \vartheta1 v$ )"
<proof>

lemma wf_subst_elim_append:
  assumes "wf_subst  $\vartheta$ " "subst_elim  $\vartheta v$ " " $v \notin$  fv t"
  shows "subst_elim ( $\vartheta(w := t)$ ) v"
<proof>

lemma wf_subst_elim_dom:
  assumes "wf_subst  $\vartheta$ "
  shows " $\forall v \in$  subst_domain  $\vartheta. \text{subst\_elim } \vartheta v$ "
<proof>

lemma wf_subst_support_iff_mgt: "wf_subst  $\vartheta \implies \vartheta$  supports  $\delta \longleftrightarrow \vartheta \preceq_o \delta$ "
<proof>

```

2.3.7 Interpretations

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

lemma $\text{interpretation_substI}$:
 $\text{"}(\bigwedge v. \text{fv } (\vartheta v) = \{\}) \Longrightarrow \text{interpretation}_{subst} \vartheta\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_grounds[simp]}$:
 $\text{"interpretation}_{subst} \vartheta \Longrightarrow \text{fv } (t \cdot \vartheta) = \{\}\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_grounds_all}$:
 $\text{"interpretation}_{subst} \vartheta \Longrightarrow (\bigwedge v. \text{fv } (\vartheta v) = \{\})\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_grounds_all}'$:
 $\text{"interpretation}_{subst} \vartheta \Longrightarrow \text{ground } (M \cdot_{\text{set}} \vartheta)\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_comp}$:
 assumes $\text{"interpretation}_{subst} \vartheta\text{"}$
 shows $\text{"interpretation}_{subst} (\sigma \circ_s \vartheta)\text{"}$ $\text{"interpretation}_{subst} (\vartheta \circ_s \sigma)\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_subst_exists}$:
 $\text{"}\exists \mathcal{I}::('f,'v) \text{subst. interpretation}_{subst} \mathcal{I}\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_subst_exists}'$:
 $\text{"}\exists \vartheta::('f,'v) \text{subst. subst_domain } \vartheta = X \wedge \text{ground } (\text{subst_range } \vartheta)\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_subst_idem}$:
 $\text{"interpretation}_{subst} \vartheta \Longrightarrow \text{subst_idem } \vartheta\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{subst_idem_comp_upd_eq}$:
 assumes $\text{"}v \notin \text{subst_domain } \mathcal{I}\text{"}$ $\text{"subst_idem } \vartheta\text{"}$
 shows $\text{"}\mathcal{I} \circ_s \vartheta = \mathcal{I}(v := \vartheta v) \circ_s \vartheta\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{interpretation_dom_img_disjoint}$:
 $\text{"interpretation}_{subst} \mathcal{I} \Longrightarrow \text{subst_domain } \mathcal{I} \cap \text{range_vars } \mathcal{I} = \{\}\text{"}$
 $\langle \text{proof} \rangle$

2.3.8 Basic Properties of MGUs

lemma $\text{MGU_is_mgu_singleton}$: $\text{"MGU } \vartheta t u = \text{is_mgu } \vartheta \{(t,u)\}\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{Unifier_in_unifiers_singleton}$: $\text{"Unifier } \vartheta s t \longleftrightarrow \vartheta \in \text{unifiers } \{(s,t)\}\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{subst_list_singleton_fv_subset}$:
 $\text{"}(\bigcup x \in \text{set } (\text{subst_list } (\text{subst } v t) E). \text{fv } (\text{fst } x) \cup \text{fv } (\text{snd } x))$
 $\subseteq \text{fv } t \cup (\bigcup x \in \text{set } E. \text{fv } (\text{fst } x) \cup \text{fv } (\text{snd } x))\text{"}$
 $\langle \text{proof} \rangle$

lemma $\text{subst_of_dom_subset}$: $\text{"subst_domain } (\text{subst_of } L) \subseteq \text{set } (\text{map } \text{fst } L)\text{"}$
 $\langle \text{proof} \rangle$

2 Preliminaries and Intruder Model

lemma `wf_MGU_is_imgu_singleton`: " $wf_{MGU} \vartheta s t \implies is_imgu \vartheta \{(s,t)\}$ "
`<proof>`

lemmas `mgu_subst_range_vars = mgu_range_vars`

lemmas `mgu_same_empty = mgu_same`

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

lemma `mgu_gives_wellformed_subst`:
assumes " $mgu\ s\ t = Some\ \vartheta$ " **shows** " $wf_{subst}\ \vartheta$ "
`<proof>`

lemma `mgu_gives_wellformed_MGU`:
assumes " $mgu\ s\ t = Some\ \vartheta$ " **shows** " $wf_{MGU}\ \vartheta\ s\ t$ "
`<proof>`

lemma `mgu_gives_subst_idem`: " $mgu\ s\ t = Some\ \vartheta \implies subst_idem\ \vartheta$ "
`<proof>`

lemma `mgu_always_unifies`: " $Unifier\ \vartheta\ M\ N \implies \exists \delta. mgu\ M\ N = Some\ \delta$ "
`<proof>`

lemma `mgu_gives_MGU`: " $mgu\ s\ t = Some\ \vartheta \implies MGU\ \vartheta\ s\ t$ "
`<proof>`

lemma `mgu_vars_bounded[dest?]`:
 $mgu\ M\ N = Some\ \sigma \implies subst_domain\ \sigma \cup range_vars\ \sigma \subseteq fv\ M \cup fv\ N$
`<proof>`

lemma `mgu_vars_bounded'`:
assumes σ : " $mgu\ M\ N = Some\ \sigma$ "
and MN : " $fv\ M = \{\} \vee fv\ N = \{\}$ "
shows " $subst_domain\ \sigma = fv\ M \cup fv\ N$ " (is ?A)
and " $range_vars\ \sigma = \{\}$ " (is ?B)
`<proof>`

lemma `mgu_eliminate[dest?]`:
assumes " $mgu\ M\ N = Some\ \sigma$ "
shows " $(\exists v \in fv\ M \cup fv\ N. subst_elim\ \sigma\ v) \vee \sigma = Var$ "
(is "?P M N σ ")
`<proof>`

lemma `mgu_eliminate_dom`:
assumes " $mgu\ x\ y = Some\ \vartheta$ " " $v \in subst_domain\ \vartheta$ "
shows " $subst_elim\ \vartheta\ v$ "
`<proof>`

lemma `unify_list_distinct`:
assumes " $Unification.unify\ E\ B = Some\ U$ " " $distinct\ (map\ fst\ B)$ "
and " $(\bigcup x \in set\ E. fv\ (fst\ x) \cup fv\ (snd\ x)) \cap set\ (map\ fst\ B) = \{\}$ "
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 " $s \cdot \delta \neq t \cdot \delta$ "
`<proof>`

lemma `mgu_None_if_neq_ground`:
assumes " $t \neq t'$ " " $fv\ t = \{\}$ " " $fv\ t' = \{\}$ "

```

shows "mgu t t' = None"
⟨proof⟩

lemma mgu_None_commutes:
  "mgu s t = None  $\implies$  mgu t s = None"
  thm mgu_complete[of s t] Unifier_in_unifiers_singleton[of ]
⟨proof⟩

lemma mgu_img_subterm_subst:
  fixes  $\delta::('f, 'v) \text{subst}$  and s t u::('f, 'v) term"
  assumes "mgu s t = Some  $\delta$ " "u  $\in$  subtermsset (subst_range  $\delta$ ) - range Var"
  shows "u  $\in$  ((subterms s  $\cup$  subterms t) - range Var) .set  $\delta$ "
⟨proof⟩

lemma mgu_img_consts:
  fixes  $\delta::('f, 'v) \text{subst}$  and s t::('f, 'v) term" and c::'f and z::'v
  assumes "mgu s t = Some  $\delta$ " "Fun c []  $\in$  subtermsset (subst_range  $\delta$ )"
  shows "Fun c []  $\in$  subterms s  $\cup$  subterms t"
⟨proof⟩

lemma mgu_img_consts':
  fixes  $\delta::('f, 'v) \text{subst}$  and s t::('f, 'v) term" and c::'f and z::'v
  assumes "mgu s t = Some  $\delta$ " " $\delta$  z = Fun c []"
  shows "Fun c []  $\sqsubseteq$  s  $\vee$  Fun c []  $\sqsubseteq$  t"
⟨proof⟩

lemma mgu_img_composed_var_term:
  fixes  $\delta::('f, 'v) \text{subst}$  and s t::('f, 'v) term" and f::'f and Z::"v list"
  assumes "mgu s t = Some  $\delta$ " "Fun f (map Var Z)  $\in$  subtermsset (subst_range  $\delta$ )"
  shows " $\exists Z'. \text{map } \delta Z' = \text{map Var Z} \wedge \text{Fun f (map Var Z')} \in \text{subterms s} \cup \text{subterms t}$ "
⟨proof⟩

lemma mgu_ground_instance_case:
  assumes t: "fv (t  $\cdot$   $\delta$ ) = {}"
  shows "mgu t (t  $\cdot$   $\delta$ ) = Some (rm_vars (UNIV - fv t)  $\delta$ )" (is ?A)
  and mgu_ground_commutes: "mgu t (t  $\cdot$   $\delta$ ) = mgu (t  $\cdot$   $\delta$ ) t" (is ?B)
⟨proof⟩

```

2.3.9 Lemmata: The "Inequality Lemmata"

Subterm injectivity (a stronger injectivity property)

definition subterm_inj_on where

"subterm_inj_on f A $\equiv \forall x \in A. \forall y \in A. (\exists v. v \sqsubseteq f x \wedge v \sqsubseteq f y) \longrightarrow x = y$ "

lemma subterm_inj_on_imp_inj_on: "subterm_inj_on f A \implies 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 \longleftrightarrow
 (inj_on f A $\wedge (\forall s \in f \backslash A. \forall u \in f \backslash A. (\exists v. v \sqsubseteq s \wedge v \sqsubseteq u) \longrightarrow s = u)$)"
 (is "?A \longleftrightarrow ?B")

⟨proof⟩

lemma subterm_inj_on_alt_def':

"subterm_inj_on ϑ (subst_domain ϑ) \longleftrightarrow
 (inj_on ϑ (subst_domain ϑ) \wedge
 ($\forall s \in \text{subst_range } \vartheta. \forall u \in \text{subst_range } \vartheta. (\exists v. v \sqsubseteq s \wedge v \sqsubseteq u) \longrightarrow s = u)$)"
 (is "?A \longleftrightarrow ?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  $\mathcal{I}$   $\vartheta$   $\sigma$  :: "('f, 'v) subst" and  $s$   $t$  :: "('f, 'v) term"
  assumes  $\vartheta$ : "subterm_inj_on  $\vartheta$  (subst_domain  $\vartheta$ )" " $\forall x \in (fv\ s \cup fv\ t) - X. \exists c. \vartheta\ x = Fun\ c\ []$ "
    "subtermsset (subst_range  $\vartheta$ )  $\cap$  (subterms  $s \cup$  subterms  $t$ ) = {}" "ground (subst_range  $\vartheta$ )"
    "subst_domain  $\vartheta \cap X = \{\}$ "
  and  $\mathcal{I}$ : "ground (subst_range  $\mathcal{I}$ )" "subst_domain  $\mathcal{I} =$  subst_domain  $\vartheta$ "
  and unif: "Unifier  $\sigma$  ( $s \cdot \vartheta$ ) ( $t \cdot \vartheta$ )"
  shows " $\exists \delta$ . Unifier  $\delta$  ( $s \cdot \mathcal{I}$ ) ( $t \cdot \mathcal{I}$ )"
<proof>

```

```

lemma inj_subst_unif_comp_terms:
  fixes  $\mathcal{I}$   $\vartheta$   $\sigma$  :: "('f, 'v) subst" and  $s$   $t$  :: "('f, 'v) term"
  assumes  $\vartheta$ : "subterm_inj_on  $\vartheta$  (subst_domain  $\vartheta$ )" "ground (subst_range  $\vartheta$ )"
    "subtermsset (subst_range  $\vartheta$ )  $\cap$  (subterms  $s \cup$  subterms  $t$ ) = {}"
    "(fv  $s \cup$  fv  $t$ ) - subst_domain  $\vartheta \subseteq X$ "
  and tfr: " $\forall f\ U. Fun\ f\ U \in$  subterms  $s \cup$  subterms  $t \longrightarrow U = [] \vee (\exists u \in set\ U. u \notin Var\ ` X)"$ "
  and  $\mathcal{I}$ : "ground (subst_range  $\mathcal{I}$ )" "subst_domain  $\mathcal{I} =$  subst_domain  $\vartheta$ "
  and unif: "Unifier  $\sigma$  ( $s \cdot \vartheta$ ) ( $t \cdot \vartheta$ )"
  shows " $\exists \delta$ . Unifier  $\delta$  ( $s \cdot \mathcal{I}$ ) ( $t \cdot \mathcal{I}$ )"
<proof>

```

context

begin

private lemma sat_ineq_subterm_inj_subst_aux:

```

  fixes  $\mathcal{I}$  :: "('f, 'v) subst"
  assumes "Unifier  $\sigma$  ( $s \cdot \mathcal{I}$ ) ( $t \cdot \mathcal{I}$ )" "ground (subst_range  $\mathcal{I}$ )"
    "(fv  $s \cup$  fv  $t$ ) -  $X \subseteq$  subst_domain  $\mathcal{I}$ " "subst_domain  $\mathcal{I} \cap X = \{\}$ "
  shows " $\exists \delta$  :: ('f, 'v) subst. subst_domain  $\delta = X \wedge$  ground (subst_range  $\delta$ )  $\wedge$   $s \cdot \delta \cdot \mathcal{I} = t \cdot \delta \cdot \mathcal{I}$ "
<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  $\vartheta$   $\mathcal{I}$   $\delta$  :: "('f, 'v) subst"
  assumes  $\vartheta$ : "subterm_inj_on  $\vartheta$  (subst_domain  $\vartheta$ )"
    "ground (subst_range  $\vartheta$ )"
    "subst_domain  $\vartheta \cap X = \{\}$ "
    "subtermsset (subst_range  $\vartheta$ )  $\cap$  (subterms  $s \cup$  subterms  $t$ ) = {}"
    "(fv  $s \cup$  fv  $t$ ) - subst_domain  $\vartheta \subseteq X$ "
  and tfr: " $(\forall x \in (fv\ s \cup fv\ t) - X. \exists c. \vartheta\ x = Fun\ c\ []) \vee$ 
    ( $\forall f\ U. Fun\ f\ U \in$  subterms  $s \cup$  subterms  $t \longrightarrow U = [] \vee (\exists u \in set\ U. u \notin Var\ ` X))"$ "
  and  $\mathcal{I}$ : " $\forall \delta$  :: ('f, 'v) subst. subst_domain  $\delta = X \wedge$  ground (subst_range  $\delta$ )  $\longrightarrow s \cdot \delta \cdot \mathcal{I} \neq t \cdot \delta \cdot \mathcal{I}$ "
    "(fv  $s \cup$  fv  $t$ ) -  $X \subseteq$  subst_domain  $\mathcal{I}$ " "subst_domain  $\mathcal{I} \cap X = \{\}$ " "ground (subst_range  $\mathcal{I}$ )"
    "subst_domain  $\mathcal{I} =$  subst_domain  $\vartheta$ "
  and  $\delta$ : "subst_domain  $\delta = X$ " "ground (subst_range  $\delta$ )"
  shows " $s \cdot \delta \cdot \vartheta \neq t \cdot \delta \cdot \vartheta$ "
<proof>
end

```

lemma ineq_subterm_inj_cond_subst:

```

  assumes " $X \cap$  range_vars  $\vartheta = \{\}$ "
  and " $\forall f\ T. Fun\ f\ T \in$  subtermsset  $S \longrightarrow T = [] \vee (\exists u \in set\ T. u \notin Var\ ` X)"$ "

```

```
shows "∀ f T. Fun f T ∈ subtermsset (S ·set ∅) → T = [] ∨ (∃ u ∈ set T. u ∉ Var ` X)"
⟨proof⟩
```

2.3.10 Lemmata: Sufficient Conditions for Term Matching

```
definition subst_var_inv:: "('a, 'b) subst ⇒ 'b set ⇒ ('a, 'b) subst" where
  "subst_var_inv δ X ≡ (λx. if Var x ∈ δ ` X then Var ((inv_into X δ) (Var x)) else Var x)"
```

```
lemma subst_var_inv_subst_domain:
  assumes "x ∈ subst_domain (subst_var_inv δ X)"
  shows "Var x ∈ δ ` X"
⟨proof⟩
```

```
lemma subst_var_inv_subst_domain':
  assumes "X ⊆ subst_domain δ"
  shows "x ∈ subst_domain (subst_var_inv δ X) ↔ Var x ∈ δ ` X"
⟨proof⟩
```

```
lemma subst_var_inv_Var_range:
  "subst_range (subst_var_inv δ X) ⊆ range Var"
⟨proof⟩
```

Injective substitutions from variables to variables are invertible

```
lemma inj_var_ran_subst_is_invertible:
  assumes δ_inj_on_X: "inj_on δ X"
  and δ_var_on_X: "δ ` X ⊆ range Var"
  and fv_t: "fv t ⊆ X"
  shows "t = t · δ ∘s subst_var_inv δ X"
⟨proof⟩
```

```
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

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

2.4.1 Syntax for the Intruder Deduction Relations

```
consts INTRUDER_SYNT:: "('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  $\Rightarrow$  nat"
  and public :: "'fun  $\Rightarrow$  bool"
  and Ana :: "('fun,'var) term  $\Rightarrow$  (('fun,'var) term list  $\times$  ('fun,'var) term list)"
  assumes Ana_keys_fv: " $\bigwedge t K R. \text{Ana } t = (K,R) \implies \text{fv}_{\text{set}} (\text{set } K) \subseteq \text{fv } t$ "
  and Ana_keys_wf: " $\bigwedge t k K R f T. \text{Ana } t = (K,R) \implies (\bigwedge g S. \text{Fun } g S \sqsubseteq t \implies \text{length } S = \text{arity } g) \implies k \in \text{set } K \implies \text{Fun } f T \sqsubseteq k \implies \text{length } T = \text{arity } f$ "
  and Ana_var[simp]: " $\bigwedge x. \text{Ana } (\text{Var } x) = ([], [])$ "
  and Ana_fun_subterm: " $\bigwedge f T K R. \text{Ana } (\text{Fun } f T) = (K,R) \implies \text{set } R \subseteq \text{set } T$ "
  and Ana_subst: " $\bigwedge t \delta K R. [\text{Ana } t = (K,R); K \neq [] \vee R \neq []] \implies \text{Ana } (t \cdot \delta) = (K \cdot_{\text{list}} \delta, R \cdot_{\text{list}} \delta)$ "
begin

lemma Ana_subterm: assumes "Ana t = (K,T)" shows "set T  $\subset$  subterms t"
  <proof>

lemma Ana_subterm': "s  $\in$  set (snd (Ana t))  $\implies$  s  $\sqsubseteq$  t"
  <proof>

lemma Ana_vars: assumes "Ana t = (K,M)" shows "fvset (set K)  $\subseteq$  fv t" "fvset (set M)  $\subseteq$  fv t"
  <proof>

abbreviation  $\mathcal{V}$  where " $\mathcal{V} \equiv \text{UNIV}::\text{'var set}$ "
abbreviation  $\Sigma_n$  ( $\langle \Sigma_{\text{pub}} \rightarrow \rangle$ ) where " $\Sigma^n \equiv \{f::\text{'fun}. \text{arity } f = n\}$ "
abbreviation  $\Sigma_{\text{pub}}$  ( $\langle \Sigma_{\text{pub}} \rightarrow \rangle$ ) where " $\Sigma_{\text{pub}}^n \equiv \{f. \text{public } f\} \cap \Sigma^n$ "
abbreviation  $\Sigma_{\text{priv}}$  ( $\langle \Sigma_{\text{priv}} \rightarrow \rangle$ ) where " $\Sigma_{\text{priv}}^n \equiv \{f. \neg \text{public } f\} \cap \Sigma^n$ "
abbreviation  $\Sigma_{\text{pub}}$  where " $\Sigma_{\text{pub}} \equiv (\bigcup n. \Sigma_{\text{pub}}^n)$ "
abbreviation  $\Sigma_{\text{priv}}$  where " $\Sigma_{\text{priv}} \equiv (\bigcup n. \Sigma_{\text{priv}}^n)$ "
abbreviation  $\Sigma$  where " $\Sigma \equiv (\bigcup n. \Sigma^n)$ "
abbreviation  $\mathcal{C}$  where " $\mathcal{C} \equiv \Sigma^0$ "
abbreviation  $\mathcal{C}_{\text{pub}}$  where " $\mathcal{C}_{\text{pub}} \equiv \{f. \text{public } f\} \cap \mathcal{C}$ "
abbreviation  $\mathcal{C}_{\text{priv}}$  where " $\mathcal{C}_{\text{priv}} \equiv \{f. \neg \text{public } f\} \cap \mathcal{C}$ "
abbreviation  $\Sigma_f$  where " $\Sigma_f \equiv \Sigma - \mathcal{C}$ "
abbreviation  $\Sigma_{f\text{pub}}$  where " $\Sigma_{f\text{pub}} \equiv \Sigma_f \cap \Sigma_{\text{pub}}$ "
abbreviation  $\Sigma_{f\text{priv}}$  where " $\Sigma_{f\text{priv}} \equiv \Sigma_f \cap \Sigma_{\text{priv}}$ "

lemma disjoint_fun_syms: " $\Sigma_f \cap \mathcal{C} = \{\}$ " <proof>
lemma id_union_univ: " $\Sigma_f \cup \mathcal{C} = \text{UNIV}$ " " $\Sigma = \text{UNIV}$ " <proof>
lemma const_arity_eq_zero[dest]: "c  $\in$   $\mathcal{C} \implies \text{arity } c = 0$ " <proof>
lemma const_pub_arity_eq_zero[dest]: "c  $\in$   $\mathcal{C}_{\text{pub}} \implies \text{arity } c = 0 \wedge \text{public } c$ " <proof>
lemma const_priv_arity_eq_zero[dest]: "c  $\in$   $\mathcal{C}_{\text{priv}} \implies \text{arity } c = 0 \wedge \neg \text{public } c$ " <proof>
lemma fun_arity_gt_zero[dest]: "f  $\in$   $\Sigma_f \implies \text{arity } f > 0$ " <proof>
lemma pub_fun_public[dest]: "f  $\in$   $\Sigma_{f\text{pub}} \implies \text{public } f$ " <proof>
lemma pub_fun_arity_gt_zero[dest]: "f  $\in$   $\Sigma_{f\text{pub}} \implies \text{arity } f > 0$ " <proof>

lemma  $\Sigma_f$ _unfold: " $\Sigma_f = \{f::\text{'fun}. \text{arity } f > 0\}$ " <proof>
lemma  $\mathcal{C}$ _unfold: " $\mathcal{C} = \{f::\text{'fun}. \text{arity } f = 0\}$ " <proof>
lemma  $\mathcal{C}_{\text{pub}}$ _unfold: " $\mathcal{C}_{\text{pub}} = \{f::\text{'fun}. \text{arity } f = 0 \wedge \text{public } f\}$ " <proof>
lemma  $\mathcal{C}_{\text{priv}}$ _unfold: " $\mathcal{C}_{\text{priv}} = \{f::\text{'fun}. \text{arity } f = 0 \wedge \neg \text{public } f\}$ " <proof>
lemma  $\Sigma_{\text{pub}}$ _unfold: " $(\Sigma_{\text{pub}}^n) = \{f::\text{'fun}. \text{arity } f = n \wedge \text{public } f\}$ " <proof>
lemma  $\Sigma_{\text{priv}}$ _unfold: " $(\Sigma_{\text{priv}}^n) = \{f::\text{'fun}. \text{arity } f = n \wedge \neg \text{public } f\}$ " <proof>
lemma  $\Sigma_{f\text{pub}}$ _unfold: " $\Sigma_{f\text{pub}} = \{f::\text{'fun}. \text{arity } f > 0 \wedge \text{public } f\}$ " <proof>
lemma  $\Sigma_{f\text{priv}}$ _unfold: " $\Sigma_{f\text{priv}} = \{f::\text{'fun}. \text{arity } f > 0 \wedge \neg \text{public } f\}$ " <proof>
lemma  $\Sigma_n$ _m_eq: " $[(\Sigma^n) \neq \{\}; (\Sigma^n) = (\Sigma^m)] \implies n = m$ " <proof>

```

2.4.3 Term Well-formedness

definition " $\text{wf}_{\text{trm}} t \equiv \forall f T. \text{Fun } f T \sqsubseteq t \longrightarrow \text{length } T = \text{arity } f$ "

abbreviation " $\text{wf}_{\text{trms}} T \equiv \forall t \in T. \text{wf}_{\text{trm}} t$ "

lemma Ana_keys_wf': "Ana t = (K,T) \implies $\text{wf}_{\text{trm}} t \implies k \in \text{set } K \implies \text{wf}_{\text{trm}} k$ "
 <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: "( $\forall x. wf_{trm} (\vartheta x)$ )  $\longleftrightarrow wf_{trms} (subst\_range \vartheta)$ "
<proof>

lemma wf_trm_subst_ranged: "wf_trms (subst_range  $\vartheta$ )  $\implies wf_{trm} (\vartheta x)$ "
<proof>

lemma wf_trm_subst_rangeI[intro]:
  "( $\bigwedge x. wf_{trm} (\delta x)$ )  $\implies wf_{trms} (subst\_range \delta)$ "
<proof>

lemma wf_trmI[intro]:
  assumes " $\bigwedge t. t \in set\ T \implies wf_{trm}\ t$ " "length T = arity f"
  shows "wf_trm (Fun f T)"
<proof>

lemma wf_trm_subterm: "[wf_trm t; s  $\sqsubseteq$  t]  $\implies wf_{trm}\ s$ "
<proof>

lemma wf_trm_subtermeq:
  assumes "wf_trm t" "s  $\sqsubseteq$  t"
  shows "wf_trm s"
<proof>

lemma wf_trm_param:
  assumes "wf_trm (Fun f T)" "t  $\in set\ T$ "
  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  $\delta$ )"
  shows "wf_trm t = wf_trm (t  $\cdot$   $\delta$ )"
<proof>

lemma wf_trm_subst_singleton:
  assumes "wf_trm t" "wf_trm t'" shows "wf_trm (t  $\cdot$  Var(v := t'))"
<proof>

lemma wf_trm_subst_rm_vars:
  assumes "wf_trm (t  $\cdot$   $\delta$ )"
  shows "wf_trm (t  $\cdot$  rm_vars X  $\delta$ )"
<proof>

lemma wf_trm_subst_rm_vars': "wf_trm ( $\delta\ v$ )  $\implies wf_{trm} (rm\_vars\ X\ \delta\ v)$ "
<proof>

lemma wf_trms_subst:
  assumes "wf_trms (subst_range  $\delta$ )" "wf_trms M"
  shows "wf_trms (M  $\cdot_{set}\ \delta$ )"
<proof>

lemma wf_trms_subst_rm_vars:
  assumes "wf_trms (M  $\cdot_{set}\ \delta$ )"

```

shows "wf_{trms} (M ·_{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  $\Rightarrow$  ('fun,'var) term  $\Rightarrow$  bool"
where
  Axiom[simp]: "t  $\in$  M  $\Longrightarrow$  intruder_deduct M t"
  | Compose[simp]: "[[length T = arity f; public f;  $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  intruder_deduct M t]]
 $\Longrightarrow$  intruder_deduct M (Fun f T)"
  | Decompose: "[[intruder_deduct M t; Ana t = (K, T);  $\bigwedge$ k. k  $\in$  set K  $\Longrightarrow$  intruder_deduct M k;
    ti  $\in$  set T]]
 $\Longrightarrow$  intruder_deduct M ti"

```

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

```

inductive intruder_synth::("('fun,'var) terms  $\Rightarrow$  ('fun,'var) term  $\Rightarrow$  bool"
where
  AxiomC[simp]: "t  $\in$  M  $\Longrightarrow$  intruder_synth M t"
  | ComposeC[simp]: "[[length T = arity f; public f;  $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  intruder_synth M t]]
 $\Longrightarrow$  intruder_synth M (Fun f T)"

```

```

adhoc_overloading INTRUDER_DEDUCT  $\equiv$  intruder_deduct
adhoc_overloading INTRUDER_SYNTH  $\equiv$  intruder_synth

```

lemma intruder_deduct_induct[consumes 1, case_names Axiom Compose Decompose]:

```

assumes "M  $\vdash$  t" " $\bigwedge$ t. t  $\in$  M  $\Longrightarrow$  P M t"
  " $\bigwedge$ T f. [[length T = arity f; public f;
 $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  M  $\vdash$  t;
 $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  P M t]]  $\Longrightarrow$  P M (Fun f T)"
  " $\bigwedge$ t K T ti. [M  $\vdash$  t; P M t; Ana t = (K, T);  $\bigwedge$ k. k  $\in$  set K  $\Longrightarrow$  M  $\vdash$  k;
 $\bigwedge$ k. k  $\in$  set K  $\Longrightarrow$  P M k; ti  $\in$  set T]]  $\Longrightarrow$  P M ti"

```

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" " $\bigwedge$ t. t  $\in$  M  $\Longrightarrow$  P M t"
  " $\bigwedge$ T f. [[length T = arity f; public f;
 $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  M  $\vdash_c$  t;
 $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  P M t]]  $\Longrightarrow$  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 \Rightarrow bool" **where**

"analyzed M \equiv \forall t. M \vdash t \longleftrightarrow M \vdash_c t"

definition analyzed_in **where**

"analyzed_in t M \equiv \forall K R. (Ana t = (K,R) \wedge (\forall k \in set K. M \vdash_c k)) \longrightarrow (\forall r \in set R. M \vdash_c r)"

definition decomp_closure::("('fun,'var) terms \Rightarrow ('fun,'var) terms \Rightarrow bool" **where**

"decomp_closure M M' \equiv \forall t. M \vdash t \wedge (\exists t' \in M. t \sqsubseteq t') \longleftrightarrow t \in M'"

inductive public_ground_wf_term::("('fun,'var) term \Rightarrow bool" **where**

```

PGWT[simp]: "[[public f; arity f = length T;
 $\bigwedge$ t. t  $\in$  set T  $\Longrightarrow$  public_ground_wf_term t]]
 $\Longrightarrow$  public_ground_wf_term (Fun f T)"

```

abbreviation "public_ground_wf_terms \equiv {t. public_ground_wf_term t}"

lemma public_const_deduct:

```

  assumes "c ∈ Cpub"
  shows "M ⊢ Fun c []" "M ⊢c Fun c []"
⟨proof⟩

lemma public_const_deduct'[simp]:
  assumes "arity c = 0" "public c"
  shows "M ⊢ Fun c []" "M ⊢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 ∈ subtermsset M"
⟨proof⟩

lemma private_fun_deduct_in_ik':
  assumes t: "M ⊢ Fun f T"
  and f: "¬public f"
  shows "Fun f T ∈ subtermsset 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 ⇒ wftrm 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 v"
  shows "M ⊢c t · v"
⟨proof⟩

```

2.4.6 Lemmata: Monotonicity, Deduction of 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⟩

context
begin

```

— Used by *inductive_set*

private lemma *ideduct_mono_set* [*mono_set*]:

" $M \subseteq N \implies M \vdash t \longrightarrow N \vdash t$ "

" $M \subseteq N \implies M \vdash_c t \longrightarrow N \vdash_c t$ "

<proof>

end

lemma *ideduct_reduce*:

" $\llbracket M \cup M' \vdash t; \bigwedge t'. t' \in M' \implies M \vdash t' \rrbracket \implies M \vdash t$ "

<proof>

lemma *ideduct_synth_reduce*:

fixes *M*: "('fun, 'var) terms" **and** *t*: "('fun, 'var) term"

shows " $\llbracket M \cup M' \vdash_c t; \bigwedge t'. t' \in M' \implies M \vdash_c t' \rrbracket \implies M \vdash_c t$ "

<proof>

lemma *ideduct_mono_eq*:

assumes " $\forall t. M \vdash t \longleftrightarrow M' \vdash t$ " **shows** " $M \cup N \vdash t \longleftrightarrow M' \cup N \vdash t$ "

<proof>

lemma *deduct_synth_subterm*:

fixes *M*: "('fun, 'var) terms" **and** *t*: "('fun, 'var) term"

assumes " $M \vdash_c t$ " "*s* \in subterms *t*" " $\forall m \in M. \forall s \in$ subterms *m. M* \vdash_c *s*"

shows " $M \vdash_c s$ "

<proof>

lemma *deduct_if_synth* [*intro, dest*]: " $M \vdash_c t \implies M \vdash t$ "

<proof> **lemma** *ideduct_ik_eq*: **assumes** " $\forall t \in M. M' \vdash t$ " **shows** " $M' \vdash t \longleftrightarrow M' \cup M \vdash t$ "

<proof> **lemma** *synth_if_deduct_empty*: " $\{\} \vdash t \implies \{\} \vdash_c t$ "

<proof> **lemma** *ideduct_deduct_synth_mono_eq*:

assumes " $\forall t. M \vdash t \longleftrightarrow M' \vdash_c t$ " " $M \subseteq M'$ "

and " $\forall t. M' \cup N \vdash t \longleftrightarrow M' \cup N \cup D \vdash_c t$ "

shows " $M \cup N \vdash t \longleftrightarrow M' \cup N \cup D \vdash_c t$ "

<proof>

lemma *ideduct_subst*: " $M \vdash t \implies M \cdot_{set} \delta \vdash t \cdot \delta$ "

<proof>

lemma *ideduct_synth_subst*:

fixes *M*: "('fun, 'var) terms" **and** *t*: "('fun, 'var) term" **and** δ : "('fun, 'var) subst"

shows " $M \vdash_c t \implies M \cdot_{set} \delta \vdash_c t \cdot \delta$ "

<proof>

lemma *ideduct_vars*:

assumes " $M \vdash t$ "

shows " $fv\ t \subseteq fv_{set}\ M$ "

<proof>

lemma *ideduct_synth_vars*:

fixes *M*: "('fun, 'var) terms" **and** *t*: "('fun, 'var) term"

assumes " $M \vdash_c t$ "

shows " $fv\ t \subseteq fv_{set}\ M$ "

<proof>

lemma *ideduct_synth_priv_fun_in_ik*:

fixes *M*: "('fun, 'var) terms" **and** *t*: "('fun, 'var) term"

assumes " $M \vdash_c t$ " "*f* \in funs_term *t*" " \neg public *f*"

shows " $f \in \bigcup (funs_term \ ` M)$ "

<proof>

lemma *ideduct_synth_priv_const_in_ik*:

```

fixes M:: "('fun, 'var) terms" and t:: "('fun, 'var) term"
assumes "M ⊢c Fun c []" "¬public c"
shows "Fun c [] ∈ M"
⟨proof⟩

```

```

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

```

2.4.7 Lemmata: Analyzed Intruder Knowledge Closure

```

lemma deducts_eq_if_analyzed: "analyzed M ⇒ M ⊢ t ⇔ M ⊢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' ⊢c t] ⇒ M ⊢ t"
⟨proof⟩

```

```

lemma decomp_closure_subterms_composable:
  assumes "decomp_closure M M'"
  and "M' ⊢c t'" "M' ⊢ t" "t ⊆ t'"
  shows "M' ⊢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' ⊢c t) ∧ decomp_closure M M' ∧ (finite M → finite M')"
⟨proof⟩

```

```

lemma deducts_eq_if_empty_ik:
  "{} ⊢ t ⇔ {} ⊢c t"
⟨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"
(<<_>_> 50)
```

where

```
AxiomR[simp]: "t ∈ M ⇒ ⟨M; Q⟩ ⊢r t"
| ComposeR[simp]: "⟦length T = arity f; public f; ∧t. t ∈ set T ⇒ ⟨M; Q⟩ ⊢r t; Q (Fun f T)⟧
⇒ ⟨M; Q⟩ ⊢r Fun f T"
| DecomposeR: "⟦⟨M; Q⟩ ⊢r t; Ana t = (K, T); ∧k. k ∈ set K ⇒ ⟨M; Q⟩ ⊢r k; ti ∈ set T⟧
⇒ ⟨M; Q⟩ ⊢r ti"
```

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

inductive intruder_deduct_num:

```
"('fun,'var) terms ⇒ nat ⇒ ('fun,'var) term ⇒ bool"
(<<_>_> 50)
```

where

```
AxiomN[simp]: "t ∈ M ⇒ ⟨M; 0⟩ ⊢n t"
| ComposeN[simp]: "⟦length T = arity f; public f; ∧t. t ∈ set T ⇒ ⟨M; steps t⟩ ⊢n t⟧
⇒ ⟨M; Suc (Max (insert 0 (steps ` set T)))⟩ ⊢n Fun f T"
| DecomposeN: "⟦⟨M; n⟩ ⊢n t; Ana t = (K, T); ∧k. k ∈ set K ⇒ ⟨M; steps k⟩ ⊢n k; ti ∈ set T⟧
⇒ ⟨M; Suc (Max (insert n (steps ` set K)))⟩ ⊢n ti"
```

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

```
assumes "⟨M; Q⟩ ⊢r t" "∧t. t ∈ M ⇒ P M Q t"
"∧T f. ⟦length T = arity f; public f;
∧t. t ∈ set T ⇒ ⟨M; Q⟩ ⊢r t;
∧t. t ∈ set T ⇒ P M Q t; Q (Fun f T)
⟧ ⇒ P M Q (Fun f T)"
"∧t K T ti. ⟦⟨M; Q⟩ ⊢r t; P M Q t; Ana t = (K, T); ∧k. k ∈ set K ⇒ ⟨M; Q⟩ ⊢r k;
∧k. k ∈ set K ⇒ P M Q k; ti ∈ set T⟧ ⇒ P M Q ti"
```

shows "P M Q t"

<proof>

lemma intruder_deduct_num_induct[consumes 1, case_names AxiomN ComposeN DecomposeN]:

```
assumes "⟨M; n⟩ ⊢n t" "∧t. t ∈ M ⇒ P M 0 t"
"∧T f steps.
⟦length T = arity f; public f;
∧t. t ∈ set T ⇒ ⟨M; steps t⟩ ⊢n t;
∧t. t ∈ set T ⇒ P M (steps t) t⟧
⇒ P M (Suc (Max (insert 0 (steps ` set T)))) (Fun f T)"
"∧t K T ti steps n.
⟦⟨M; n⟩ ⊢n t; P M n t; Ana t = (K, T);
∧k. k ∈ set K ⇒ ⟨M; steps k⟩ ⊢n k;
ti ∈ set T; ∧k. k ∈ set K ⇒ P M (steps k) k⟧
⇒ P M (Suc (Max (insert n (steps ` set K)))) ti"
```

shows "P M n t"

<proof>

lemma ideduct_restricted_mono:

```
"⟦⟨M; P⟩ ⊢r t; M ⊆ M'⟧ ⇒ ⟨M'; P⟩ ⊢r t"
```

<proof>

2.4.9 Lemmata: Intruder Deduction Equivalences

lemma deduct_if_restricted_deduct: " $\langle M; P \rangle \vdash_r m \Rightarrow M \vdash m$ "

<proof>

lemma restricted_deduct_if_restricted_ik:

```
assumes "⟨M; P⟩ ⊢r m" "∀m ∈ M. P m"
and P: "∀t t'. P t → t' ⊆ t → P t'"
shows "P m"
```

<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 $M: "\forall m \in M. \forall f\ T. \text{Fun}\ f\ T \sqsubseteq m \longrightarrow P\ f\ T"$
 and $t: "\langle M; k \rangle \vdash_n\ t"$ $"\text{Fun}\ f\ T \sqsubseteq t"$ $"\neg P\ f\ T"$
 shows $"\exists l \leq k. (\langle M; l \rangle \vdash_n\ \text{Fun}\ f\ T) \wedge (\forall t \in \text{set}\ T. \exists j < l. \langle M; j \rangle \vdash_n\ t)"$

<proof>

lemma *deduct_inv*:

assumes $"\langle M; n \rangle \vdash_n\ t"$
 shows $"t \in M \vee$
 $(\exists f\ T. t = \text{Fun}\ f\ T \wedge \text{public}\ f \wedge \text{length}\ T = \text{arity}\ f \wedge (\forall t \in \text{set}\ T. \exists l < n. \langle M; l \rangle \vdash_n\ t)) \vee$
 $(\exists m \in \text{subterms}_{\text{set}}\ M.$
 $(\exists l < n. \langle M; l \rangle \vdash_n\ m) \wedge (\forall k \in \text{set}\ (\text{fst}\ (\text{Ana}\ m)). \exists l < n. \langle M; l \rangle \vdash_n\ k) \wedge$
 $t \in \text{set}\ (\text{snd}\ (\text{Ana}\ m)))"$
 (is $"?P\ t\ n \vee ?Q\ t\ n \vee ?R\ t\ n"$)

<proof>

lemma *deduct_inv'*:

assumes $"M \vdash \text{Fun}\ f\ ts"$
 shows $"\text{Fun}\ f\ ts \sqsubseteq_{\text{set}}\ M \vee (\forall t \in \text{set}\ ts. M \vdash t)"$

<proof>

lemma *restricted_deduct_if_deduct*:

assumes $M: "\forall m \in M. \forall f\ T. \text{Fun}\ f\ T \sqsubseteq m \longrightarrow P\ (\text{Fun}\ f\ T)"$
 and $P_{\text{subterm}}: "\forall f\ T\ t. M \vdash \text{Fun}\ f\ T \longrightarrow P\ (\text{Fun}\ f\ T) \longrightarrow t \in \text{set}\ T \longrightarrow P\ t"$
 and $P_{\text{Ana_key}}: "\forall t\ K\ T\ k. M \vdash t \longrightarrow P\ t \longrightarrow \text{Ana}\ t = (K, T) \longrightarrow M \vdash k \longrightarrow k \in \text{set}\ K \longrightarrow P\ k"$
 and $m: "M \vdash m"$ $"P\ m"$
 shows $"\langle M; P \rangle \vdash_r\ m"$

<proof>

lemma *restricted_deduct_if_deduct'*:

assumes $"\forall m \in M. P\ m"$
 and $"\forall t\ t'. P\ t \longrightarrow t' \sqsubseteq t \longrightarrow P\ t'"$
 and $"\forall t\ K\ T\ k. P\ t \longrightarrow \text{Ana}\ t = (K, T) \longrightarrow k \in \text{set}\ K \longrightarrow P\ k"$
 and $"M \vdash m"$ $"P\ m"$
 shows $"\langle M; P \rangle \vdash_r\ m"$

<proof>

lemma *private_const_deduct*:

assumes $c: "\neg \text{public}\ c"$ $"M \vdash (\text{Fun}\ c\ []::('fun, 'var)\ \text{term})"$
 shows $"\text{Fun}\ c\ [] \in M \vee$
 $(\exists m \in \text{subterms}_{\text{set}}\ M. M \vdash m \wedge (\forall k \in \text{set}\ (\text{fst}\ (\text{Ana}\ m)). M \vdash m) \wedge$
 $\text{Fun}\ c\ [] \in \text{set}\ (\text{snd}\ (\text{Ana}\ m)))"$

<proof>

lemma *private_fun_deduct_in_ik''*:

assumes $t: "M \vdash \text{Fun}\ f\ T"$ $"\text{Fun}\ c\ [] \in \text{set}\ T"$ $"\forall m \in \text{subterms}_{\text{set}}\ M. \text{Fun}\ f\ T \notin \text{set}\ (\text{snd}\ (\text{Ana}\ m))"$
 and $c: "\neg \text{public}\ c"$ $"\text{Fun}\ c\ [] \notin M"$ $"\forall m \in \text{subterms}_{\text{set}}\ M. \text{Fun}\ c\ [] \notin \text{set}\ (\text{snd}\ (\text{Ana}\ m))"$

```

  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 "wftrm' ar t ≡ (∀ s ∈ subterms t. is_Fun s → ar (the_Fun s) = length (args s))"

```

```

definition "wftrms' ar M ≡ (∀ t ∈ M. wftrm' 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) wftrm_code[code_unfold]:
  "wftrm t = wftrm' arity t"
  ⟨proof⟩

```

```

lemma (in intruder_model) wftrms_code[code_unfold]:
  "wftrms M = wftrms' arity M"
  ⟨proof⟩

```

```

lemma (in intruder_model) intruder_synth_code[code_unfold]:
  "intruder_synth M t = intruder_synth' public arity M t"
  (is "?A ↔ ?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

```
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 (funsstp: 'a, varsstp: 'b) strand_step =
  Send      "('a,'b) term list" (<send⟨_⟩st> 80)
| Receive   "('a,'b) term list" (<receive⟨_⟩st> 80)
| Equality  poscheckvariant "('a,'b) term" "('a,'b) term" (<⟨_ : _ ≐ _⟩st> [80,80])
| Inequality (bvarsstp: "'b list") ("('a,'b) term × ('a,'b) term) list" (<∀_⟨≠: _⟩st> [80,80])
where
  "bvarsstp (Send _) = []"
| "bvarsstp (Receive _) = []"
| "bvarsstp (Equality _ _ _) = []"
```

```
abbreviation "Send1 t ≡ Send [t]"
abbreviation "Receive1 t ≡ Receive [t]"
```

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 "trmspairs F ≡ ⋃ (t,t') ∈ set F. {t,t'}"
```

```
fun trmsstp :: "('a,'b) strand_step ⇒ ('a,'b) terms" where
  "trmsstp (Send ts) = set ts"
| "trmsstp (Receive ts) = set ts"
| "trmsstp (Equality _ t t') = {t,t'}"
| "trmsstp (Inequality _ F) = trmspairs F"
```

```
lemma varsstp_unfold[simp]: "varsstp x = fvset (trmsstp x) ∪ set (bvarsstp x)"
<proof>
```

The set of terms occurring in a strand

```
definition trmsst where "trmsst S ≡ ⋃ (trmsstp ` set S)"
```

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

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

3 The Typing Result for Non-Stateful Protocols

definition $trms_list_{st}$ **where** $"trms_list_{st} S \equiv remdups (concat (map trms_list_{stp} S))"$

The set of variables occurring in a sent message

definition $fv_{snd}::('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{snd} x \equiv case x of Send t \Rightarrow fv_{set} (set t) \mid _ \Rightarrow \{\}"$

The set of variables occurring in a received message

definition $fv_{rcv}::('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{rcv} x \equiv case x of Receive t \Rightarrow fv_{set} (set t) \mid _ \Rightarrow \{\}"$

The set of variables occurring in an equality constraint

definition $fv_{eq}::"poscheckvariant \Rightarrow ('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{eq} ac x \equiv case x of Equality ac' s t \Rightarrow if ac = ac' then fv s \cup fv t else \{\} \mid _ \Rightarrow \{\}"$

The set of variables occurring at the left-hand side of an equality constraint

definition $fv_{leq}::"poscheckvariant \Rightarrow ('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{leq} ac x \equiv case x of Equality ac' s t \Rightarrow if ac = ac' then fv s else \{\} \mid _ \Rightarrow \{\}"$

The set of variables occurring at the right-hand side of an equality constraint

definition $fv_{req}::"poscheckvariant \Rightarrow ('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{req} ac x \equiv case x of Equality ac' s t \Rightarrow if ac = ac' then fv t else \{\} \mid _ \Rightarrow \{\}"$

The free variables of inequality constraints

definition $fv_{ineq}::('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{ineq} x \equiv case x of Inequality X F \Rightarrow fv_{pairs} F - set X \mid _ \Rightarrow \{\}"$

fun $fv_{stp}::('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"fv_{stp} (Send t) = fv_{set} (set t)"$
 $\mid "fv_{stp} (Receive t) = fv_{set} (set t)"$
 $\mid "fv_{stp} (Equality _ t t') = fv t \cup fv t'"$
 $\mid "fv_{stp} (Inequality X F) = (\bigcup (t,t') \in set F. fv t \cup fv t') - set X"$

The set of free variables of a strand

definition $fv_{st}::('a,'b) strand \Rightarrow 'b set$ **where**
 $"fv_{st} S \equiv \bigcup (set (map fv_{stp} S))"$

The set of bound variables of a strand

definition $bvars_{st}::('a,'b) strand \Rightarrow 'b set$ **where**
 $"bvars_{st} S \equiv \bigcup (set (map (set \circ bvars_{stp}) S))"$

The set of all variables occurring in a strand

definition $vars_{st}::('a,'b) strand \Rightarrow 'b set$ **where**
 $"vars_{st} S \equiv \bigcup (set (map vars_{stp} S))"$

abbreviation $wfrestrictedvars_{stp}::('a,'b) strand_step \Rightarrow 'b set$ **where**
 $"wfrestrictedvars_{stp} x \equiv$
 $case x of Inequality _ _ \Rightarrow \{\} \mid Equality Check _ _ \Rightarrow \{\} \mid _ \Rightarrow vars_{stp} x"$

The variables of a strand whose occurrences might be restricted by well-formedness constraints

definition $wfrestrictedvars_{st}::('a,'b) strand \Rightarrow 'b set$ **where**
 $"wfrestrictedvars_{st} S \equiv \bigcup (set (map wfrestrictedvars_{stp} S))"$

abbreviation $wfvarsoccs_{stp}$ **where**
 $"wfvarsoccs_{stp} x \equiv case x of Send t \Rightarrow fv_{set} (set t) \mid Equality Assign s t \Rightarrow fv s \mid _ \Rightarrow \{\}"$

The variables of a strand that occur in sent messages or in assignments

definition $wfvarsoccs_{st}$ **where**
 $"wfvarsoccs_{st} S \equiv \bigcup (set (map wfvarsoccs_{stp} S))"$

The variables occurring at the right-hand side of assignment steps

fun $assignment_rhs_{st}$ **where**

```

"assignment_rhsst [] = {}"
| "assignment_rhsst (Equality Assign t t'#S) = insert t' (assignment_rhsst S)"
| "assignment_rhsst (x#S) = assignment_rhsst S"

```

The set of function symbols occurring in a strand

```

definition funsst::('a,'b) strand ⇒ 'a set" where
  "funsst S ≡ ⋃ (set (map funsstp S))"

```

```

fun subst_apply_strand_step::('a,'b) strand_step ⇒ ('a,'b) subst ⇒ ('a,'b) strand_step"
  (infix <·stp> 51) where
  "Send t ·stp ∅ = Send (t ·list ∅)"
| "Receive t ·stp ∅ = Receive (t ·list ∅)"
| "Equality a t t' ·stp ∅ = Equality a (t · ∅) (t' · ∅)"
| "Inequality X F ·stp ∅ = Inequality X (F ·pairs rm_vars (set X) ∅)"

```

Substitution application for strands

```

definition subst_apply_strand::('a,'b) strand ⇒ ('a,'b) subst ⇒ ('a,'b) strand"
  (infix <·st> 51) where
  "S ·st ∅ ≡ map (λx. x ·stp ∅) 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) →
      (∃(t,t') ∈ set F. t · δ ◦s I ≠ t' · δ ◦s I))"

```

```

fun simplestp where
  "simplestp (Receive t) = True"
| "simplestp (Send [Var v]) = True"
| "simplestp (Inequality X F) = (∃I. ineq_model I X F)"
| "simplestp _ = False"

```

Simple constraints

```

definition simple where "simple S ≡ list_all simplestp S"

```

The intruder knowledge of a constraint

```

fun ikst::('a,'b) strand ⇒ ('a,'b) terms" where
  "ikst [] = {}"
| "ikst (Receive t#S) = set t ∪ (ikst S)"
| "ikst (_#S) = ikst S"

```

Strand well-formedness

```

fun wfst::'b set ⇒ ('a,'b) strand ⇒ bool" where
  "wfst V [] = True"
| "wfst V (Receive ts#S) = (fvset (set ts) ⊆ V ∧ wfst V S)"
| "wfst V (Send ts#S) = wfst (V ∪ fvset (set ts)) S"
| "wfst V (Equality Assign s t#S) = (fv t ⊆ V ∧ wfst (V ∪ fv s) S)"
| "wfst V (Equality Check s t#S) = wfst V S"
| "wfst V (Inequality _ _#S) = wfst V S"

```

Well-formedness of constraint states

```

definition wfconstr::('a,'b) strand ⇒ ('a,'b) subst ⇒ bool" where
  "wfconstr S ∅ ≡ (wfsubst ∅ ∧ wfst {} S ∧ subst_domain ∅ ∩ varsst S = {} ∧
    range_vars ∅ ∩ bvarsst S = {} ∧ fvst S ∩ bvarsst S = {})"

```

```

declare trmsst_def[simp]
declare fvsnd_def[simp]
declare fvrcv_def[simp]
declare fveq_def[simp]
declare fvleq_def[simp]
declare fvreq_def[simp]
declare fvineq_def[simp]

```

```

declare fvst_def[simp]
declare varsst_def[simp]
declare bvarsst_def[simp]
declare wfrestrictedvarsst_def[simp]
declare wfvarsoccsst_def[simp]

lemmas wfst_induct = wfst.induct[case_names Nil ConsRcv ConsSnd ConsEq ConsEq2 ConsIneq]
lemmas ikst_induct = ikst.induct[case_names Nil ConsRcv ConsSnd ConsEq ConsIneq]
lemmas assignment_rhsst_induct = assignment_rhsst.induct[case_names Nil ConsEq2 ConsSnd ConsRcv ConsEq ConsIneq]

```

Lexicographical measure on strands

```

definition sizest:: "('a, 'b) strand ⇒ nat" where
  "sizest S ≡ size_list (λx. Max (insert 0 (size ` trmsstp x))) S"

definition measurest:: "((('a, 'b) strand × ('a, 'b) subst) × ('a, 'b) strand × ('a, 'b) subst) set"
where
  "measurest ≡ measures [λ(S, ϑ). card (fvst S), λ(S, ϑ). sizest S]"

```

```

lemma measurest_alt_def:
  "((s, x), (t, y)) ∈ measurest =
    (card (fvst s) < card (fvst t) ∨ (card (fvst s) = card (fvst t) ∧ sizest s < sizest t))"
⟨proof⟩

```

```

lemma measurest_trans: "trans measurest"
⟨proof⟩

```

Some lemmata

```

lemma trms_listst_is_trmsst: "trmsst S = set (trms_listst S)"
⟨proof⟩

```

```

lemma subst_apply_strand_step_def:
  "sstp ϑ = (case s of
    Send t ⇒ Send (tlist ϑ)
  | Receive t ⇒ Receive (tlist ϑ)
  | Equality a t t' ⇒ Equality a (t · ϑ) (t' · ϑ)
  | Inequality X F ⇒ Inequality X (Fpairs rm_vars (set X) ϑ))"
⟨proof⟩

```

```

lemma subst_apply_strand_nil[simp]: "[ ]st δ = [ ]"
⟨proof⟩

```

```

lemma finite_funsstp[simp]: "finite (funsstp x)" ⟨proof⟩
lemma finite_funsst[simp]: "finite (funsst S)" ⟨proof⟩
lemma finite_trmspairs[simp]: "finite (trmspairs x)" ⟨proof⟩
lemma finite_trmsstp[simp]: "finite (trmsstp x)" ⟨proof⟩
lemma finite_varsstp[simp]: "finite (varsstp x)" ⟨proof⟩
lemma finite_bvarsstp[simp]: "finite (set (bvarsstp x))" ⟨proof⟩
lemma finite_fvsnd[simp]: "finite (fvsnd x)" ⟨proof⟩
lemma finite_fvrcv[simp]: "finite (fvrcv x)" ⟨proof⟩
lemma finite_fvstp[simp]: "finite (fvstp x)" ⟨proof⟩
lemma finite_varsst[simp]: "finite (varsst S)" ⟨proof⟩
lemma finite_bvarsst[simp]: "finite (bvarsst S)" ⟨proof⟩
lemma finite_fvst[simp]: "finite (fvst S)" ⟨proof⟩

```

```

lemma finite_wfrestrictedvarsstp[simp]: "finite (wfrestrictedvarsstp x)"
⟨proof⟩

```

```

lemma finite_wfrestrictedvarsst[simp]: "finite (wfrestrictedvarsst S)"
⟨proof⟩

```

```

lemma finite_wfvarsoccsstp[simp]: "finite (wfvarsoccsstp x)"
⟨proof⟩

lemma finite_wfvarsoccsst[simp]: "finite (wfvarsoccsst S)"
⟨proof⟩

lemma finite_ikst[simp]: "finite (ikst S)"
⟨proof⟩

lemma finite_assignment_rhsst[simp]: "finite (assignment_rhsst S)"
⟨proof⟩

lemma ikst_is_rcv_set: "ikst A = {t | ts t. Receive ts ∈ set A ∧ t ∈ set ts}"
⟨proof⟩

lemma ikst_snoc_no_receive_eq:
  assumes "¬∃ts. a = receive(ts)st"
  shows "ikst (A@[a]) ·set I = ikst A ·set I"
⟨proof⟩

lemma ikstD[dest]: "t ∈ ikst S ⇒ ∃ts. t ∈ set ts ∧ Receive ts ∈ set S"
⟨proof⟩

lemma ikstD'[dest]: "t ∈ ikst S ⇒ t ∈ trmsst S"
⟨proof⟩

lemma ikstD''[dest]: "t ∈ subtermsset (ikst S) ⇒ t ∈ subtermsset (trmsst S)"
⟨proof⟩

lemma ikst_subterm_exD:
  assumes "t ∈ ikst S"
  shows "∃x ∈ set S. t ∈ subtermsset (trmsstp x)"
⟨proof⟩

lemma assignment_rhsstD[dest]: "t ∈ assignment_rhsst S ⇒ ∃t'. Equality Assign t' t ∈ set S"
⟨proof⟩

lemma assignment_rhsstD'[dest]: "t ∈ subtermsset (assignment_rhsst S) ⇒ t ∈ subtermsset (trmsst S)"
⟨proof⟩

lemma bvarsst_split: "bvarsst (S@S') = bvarsst S ∪ bvarsst S'"
⟨proof⟩

lemma bvarsst_singleton: "bvarsst [x] = set (bvarsstp x)"
⟨proof⟩

lemma strand_fv_bvars_disjointD:
  assumes "fvst S ∩ bvarsst S = {}" "Inequality X F ∈ set S"
  shows "set X ⊆ bvarsst S" "fvpairs F - set X ⊆ fvst S"
⟨proof⟩

lemma strand_fv_bvars_disjoint_unfold:
  assumes "fvst S ∩ bvarsst S = {}" "Inequality X F ∈ set S" "Inequality Y G ∈ set S"
  shows "set Y ∩ (fvpairs F - set X) = {}"
⟨proof⟩

lemma strand_subst_hom[iff]:
  "(S@S') ·st ϑ = (S ·st ϑ)@(S' ·st ϑ)" "(x#S) ·st ϑ = (x ·stp ϑ)#(S ·st ϑ)"
⟨proof⟩

lemma strand_subst_comp: "range_vars δ ∩ bvarsst S = {} ⇒ S ·st δ ∘S ϑ = ((S ·st δ) ·st ϑ)"
⟨proof⟩

```

3 The Typing Result for Non-Stateful Protocols

lemma strand_substI[intro]:

"subst_domain $\vartheta \cap \text{fv}_{st} S = \{\}$ $\implies S \cdot_{st} \vartheta = S$ "
 "subst_domain $\vartheta \cap \text{vars}_{st} S = \{\}$ $\implies S \cdot_{st} \vartheta = S$ "

<proof>

lemma strand_substI':

" $\text{fv}_{st} S = \{\}$ $\implies S \cdot_{st} \vartheta = S$ "
 " $\text{vars}_{st} S = \{\}$ $\implies S \cdot_{st} \vartheta = S$ "

<proof>

lemma strand_subst_set: "(set (S \cdot_{st} ϑ)) = (($\lambda x. x \cdot_{stp} \vartheta$) ` (set S))"

<proof>

lemma strand_map_inv_set_snd_rcv_subst:

assumes "finite (M::('a,'b) terms)"
 shows "set ((map Send1 (inv set M)) \cdot_{st} ϑ) = Send1 ` (M \cdot_{set} ϑ)" (is ?A)
 "set ((map Receive1 (inv set M)) \cdot_{st} ϑ) = Receive1 ` (M \cdot_{set} ϑ)" (is ?B)

<proof>

lemma strand_ground_subst_vars_subset:

assumes "ground (subst_range ϑ)" shows " $\text{vars}_{st} (S \cdot_{st} \vartheta) \subseteq \text{vars}_{st} S$ "

<proof>

lemma ik_union_subset: " $\bigcup (P \setminus \text{ik}_{st} S) \subseteq (\bigcup x \in (\text{set } S). \bigcup (P \setminus \text{trms}_{stp} x))$ "

<proof>

lemma ik_snd_empty[simp]: " $\text{ik}_{st} (\text{map Send } X) = \{\}$ "

<proof>

lemma ik_snd_empty'[simp]: " $\text{ik}_{st} [\text{Send } t] = \{\}$ " *<proof>*

lemma ik_append[iff]: " $\text{ik}_{st} (S@S') = \text{ik}_{st} S \cup \text{ik}_{st} S'$ " *<proof>*

lemma ik_cons: " $\text{ik}_{st} (x\#S) = \text{ik}_{st} [x] \cup \text{ik}_{st} S$ " *<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: " $\text{assignment_rhs}_{st} (\text{map Receive } M) = \{\}$ "

<proof>

lemma ik_rcv_map: assumes " $ts \in \text{set } L$ " shows " $\text{set } ts \subseteq \text{ik}_{st} (\text{map Receive } L)$ "

<proof>

lemma ik_subst: " $\text{ik}_{st} (S \cdot_{st} \delta) = \text{ik}_{st} S \cdot_{set} \delta$ "

<proof>

lemma ik_rcv_map': assumes " $t \in \text{ik}_{st} (\text{map Receive } L)$ " shows " $\exists ts \in \text{set } L. t \in \text{set } ts$ "

<proof>

lemma ik_append_subset[simp]: " $\text{ik}_{st} S \subseteq \text{ik}_{st} (S@S')$ " " $\text{ik}_{st} S' \subseteq \text{ik}_{st} (S@S')$ "

<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: " $\text{trms}_{st} (x\#S) = \text{trms}_{stp} x \cup \text{trms}_{st} S$ " *<proof>*

lemma trm_strand_subst_cong:

" $t \in \text{trms}_{st} S \implies t \cdot \delta \in \text{trms}_{st} (S \cdot_{st} \delta)$ "
 $\vee (\exists X F. \text{Inequality } X F \in \text{set } S \wedge t \cdot \text{rm_vars } (\text{set } X) \delta \in \text{trms}_{st} (S \cdot_{st} \delta))$ "

```

(is "t ∈ trmsst S ⇒ ?P t δ S")
"t ∈ trmsst (S ·st δ) ⇒ (∃ t'. t = t' · δ ∧ t' ∈ trmsst S)
∨ (∃ X F. Inequality X F ∈ set S ∧ (∃ t' ∈ trmspairs F. t = t' · rm_vars (set X) δ))"
(is "t ∈ trmsst (S ·st δ) ⇒ ?Q t δ S")
⟨proof⟩

```

3.1.2 Lemmata: Free Variables of Strands

```

lemma fv_trm_snd_rcv[simp]:
  "fvset (trmsstp (Send ts)) = fvset (set ts)" "fvset (trmsstp (Receive ts)) = fvset (set ts)"
⟨proof⟩

lemma in_strand_fv_subset: "x ∈ set S ⇒ varsstp x ⊆ varsst S"
⟨proof⟩

lemma in_strand_fv_subset_snd: "Send ts ∈ set S ⇒ fvset (set ts) ⊆ ⋃ (set (map fvsnd S))"
⟨proof⟩

lemma in_strand_fv_subset_rcv: "Receive ts ∈ set S ⇒ fvset (set ts) ⊆ ⋃ (set (map fvrcv S))"
⟨proof⟩

lemma fvsndE:
  assumes "v ∈ ⋃ (set (map fvsnd S))"
  obtains ts where "send⟨ts⟩st ∈ set S" "v ∈ fvset (set ts)"
⟨proof⟩

lemma fvrcvE:
  assumes "v ∈ ⋃ (set (map fvrcv S))"
  obtains ts where "receive⟨ts⟩st ∈ set S" "v ∈ fvset (set ts)"
⟨proof⟩

lemma varsstpI[intro]: "x ∈ fvstp s ⇒ x ∈ varsstp s"
⟨proof⟩

lemma varsstI[intro]: "x ∈ fvst S ⇒ x ∈ varsst S" ⟨proof⟩

lemma fvst_subset_varsst[simp]: "fvst S ⊆ varsst S" ⟨proof⟩

lemma varsst_is_fvst_bvarsst: "varsst S = fvst S ∪ bvarsst S"
⟨proof⟩

lemma fvstp_is_subterm_trmsstp: "x ∈ fvstp a ⇒ Var x ∈ subtermsset (trmsstp a)"
⟨proof⟩

lemma fvst_is_subterm_trmsst: "x ∈ fvst A ⇒ Var x ∈ subtermsset (trmsst A)"
⟨proof⟩

lemma varsst_snd_map: "varsst (map Send tss) = fvset (Fun f ` set tss)" ⟨proof⟩

lemma varsst_rcv_map: "varsst (map Receive tss) = fvset (Fun f ` set tss)" ⟨proof⟩

lemma vars_snd_rcv_union:
  "varsstp x = fvsnd x ∪ fvrcv x ∪ fveq assign x ∪ fveq check x ∪ fvineq x ∪ set (bvarsstp x)"
⟨proof⟩

lemma fv_snd_rcv_union:
  "fvstp x = fvsnd x ∪ fvrcv x ∪ fveq assign x ∪ fveq check x ∪ fvineq x"
⟨proof⟩

lemma fv_snd_rcv_empty[simp]: "fvsnd x = {} ∨ fvrcv x = {}" ⟨proof⟩

lemma vars_snd_rcv_strand[iff]:
  "varsst (S::('a, 'b) strand) =

```

3 The Typing Result for Non-Stateful Protocols

```

  (⋃(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"
  "wfvvarsoccs_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" "wfvvarsoccs_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 ·_stp δ) ⊆ fv_stp x ∪ range_vars δ"
⟨proof⟩

```

```

lemma subst_sends_strand_fv_to_img: "fv_st (S ·_st δ) ⊆ fv_st S ∪ range_vars δ"
⟨proof⟩

```

```

lemma ineq_apply_subst:
  assumes "subst_domain δ ∩ set X = {}"
  shows "(Inequality X F) ·_stp δ = Inequality X (F ·_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 ·_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 ·_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 ·_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 ·_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 ·_{pairs} ∅) = trms_{pairs} a ·_{set} ∅"
 <proof>

lemma trms_{pairs}_fv_subst_subset:
 "t ∈ trms_{pairs} F ⇒ fv (t · ∅) ⊆ fv_{pairs} (F ·_{pairs} ∅)"
 <proof>

lemma trms_{pairs}_fv_subst_subset':
 fixes t::('a,'b) term and ∅::('a,'b) subst"
 assumes "t ∈ subterms_{set} (trms_{pairs} F)"
 shows "fv (t · ∅) ⊆ fv_{pairs} (F ·_{pairs} ∅)"
 <proof>

lemma trms_{pairs}_funs_term_cases:
 assumes "t ∈ trms_{pairs} (F ·_{pairs} ∅)" "f ∈ funs_term t"
 shows "(∃u ∈ trms_{pairs} F. f ∈ funs_term u) ∨ (∃x ∈ fv_{pairs} F. f ∈ funs_term (∅ x))"
 <proof>

lemma trm_{stp}_subst:
 assumes "subst_domain ∅ ∩ set (bvars_{stp} a) = {}"
 shows "trms_{stp} (a ·_{stp} ∅) = trms_{stp} a ·_{set} ∅"
 <proof>

lemma trms_{st}_subst:
 assumes "subst_domain ∅ ∩ bvars_{st} A = {}"
 shows "trms_{st} (A ·_{st} ∅) = trms_{st} A ·_{set} ∅"
 <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>

3 The Typing Result for Non-Stateful Protocols

lemma *subst_apply_fv_subset_strand2*:

assumes $P: "P = fv_{stp} \vee P = fv_{rcv} \vee P = fv_{snd} \vee P = fv_{eq} \text{ ac} \vee P = fv_{ineq} \vee P = fv_{req} \text{ ac}"$
and $P_subset: "P \subseteq wfrestrictedvars_{st} S \cup V"$
and $\delta: "bvars_{st} S \cap (subst_domain \delta \cup range_vars \delta) = \{\}"$
 $"set (bvars_{stp} x) \cap (subst_domain \delta \cup range_vars \delta) = \{\}"$
shows $"P (x \cdot_{stp} \delta) \subseteq wfrestrictedvars_{st} (S \cdot_{st} \delta) \cup fv_{set} (\delta \setminus V)"$

<proof>

lemma *strand_subst_fv_bounded_if_img_bounded*:

assumes $"range_vars \delta \subseteq fv_{st} S"$
shows $"fv_{st} (S \cdot_{st} \delta) \subseteq fv_{st} S"$

<proof>

lemma *strand_fv_subst_subset_if_subst_elim*:

assumes $"subst_elim \delta v"$ and $"v \in fv_{st} S \vee bvars_{st} S \cap (subst_domain \delta \cup range_vars \delta) = \{\}"$
shows $"v \notin fv_{st} (S \cdot_{st} \delta)"$

<proof>

lemma *strand_fv_subst_subset_if_subst_elim'*:

assumes $"subst_elim \delta v"$ $"v \in fv_{st} S"$ $"range_vars \delta \subseteq fv_{st} S"$
shows $"fv_{st} (S \cdot_{st} \delta) \subseteq fv_{st} S"$

<proof>

lemma *fv_ik_is_fv_rcv*: $"fv_{set} (ik_{st} S) = \bigcup (set (map fv_{rcv} S))"$

<proof>

lemma *fv_ik_subset_fv_st[simp]*: $"fv_{set} (ik_{st} S) \subseteq wfrestrictedvars_{st} S"$

<proof>

lemma *fv_assignment_rhs_subset_fv_st[simp]*: $"fv_{set} (assignment_rhs_{st} S) \subseteq wfrestrictedvars_{st} S"$

<proof>

lemma *fv_ik_subset_fv_st'[simp]*: $"fv_{set} (ik_{st} S) \subseteq fv_{st} S"$

<proof>

lemma *ik_st_var_is_fv*: $"Var x \in subterms_{set} (ik_{st} A) \implies x \in fv_{st} A"$

<proof>

lemma *fv_assignment_rhs_subset_fv_st'[simp]*: $"fv_{set} (assignment_rhs_{st} S) \subseteq fv_{st} S"$

<proof>

lemma *ik_st_assignment_rhs_st_wfrestrictedvars_subset*:

$"fv_{set} (ik_{st} A \cup assignment_rhs_{st} A) \subseteq wfrestrictedvars_{st} A"$

<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]*: $"vars_{st} (S \cdot_{st} \delta) \subseteq vars_{st} S \cup range_vars \delta"$

<proof>

lemma *strand_vars_split*:

$"vars_{st} (S@S') = vars_{st} S \cup vars_{st} S'"$
 $"wfrestrictedvars_{st} (S@S') = wfrestrictedvars_{st} S \cup wfrestrictedvars_{st} S'"$
 $"fv_{st} (S@S') = fv_{st} S \cup fv_{st} S'"$

<proof>

lemma *bvars_subst_ident*: $"bvars_{st} S = bvars_{st} (S \cdot_{st} \delta)"$

<proof>

lemma *strand_subst_subst_idem*:

assumes $"subst_idem \delta"$ $"subst_domain \delta \cup range_vars \delta \subseteq fv_{st} S"$ $"subst_domain \vartheta \cap fv_{st} S = \{\}"$

```

      "range_vars  $\delta \cap \text{bvars}_{st} S = \{\}$ " "range_vars  $\vartheta \cap \text{bvars}_{st} S = \{\}$ "
shows "(S  $\cdot_{st} \delta$ )  $\cdot_{st} \vartheta = (S \cdot_{st} \delta)$ "
and "(S  $\cdot_{st} \delta$ )  $\cdot_{st} (\vartheta \circ_s \delta) = (S \cdot_{st} \delta)$ "
⟨proof⟩

lemma strand_subst_img_bound:
  assumes "subst_domain  $\delta \cup \text{range\_vars } \delta \subseteq \text{fv}_{st} S$ "
  and "(subst_domain  $\delta \cup \text{range\_vars } \delta) \cap \text{bvars}_{st} S = \{\}$ "
  shows "range_vars  $\delta \subseteq \text{fv}_{st} (S \cdot_{st} \delta)$ "
⟨proof⟩

lemma strand_subst_img_bound':
  assumes "subst_domain  $\delta \cup \text{range\_vars } \delta \subseteq \text{vars}_{st} S$ "
  and "(subst_domain  $\delta \cup \text{range\_vars } \delta) \cap \text{bvars}_{st} S = \{\}$ "
  shows "range_vars  $\delta \subseteq \text{vars}_{st} (S \cdot_{st} \delta)$ "
⟨proof⟩

lemma strand_subst_all_fv_subset:
  assumes "fv t  $\subseteq \text{fv}_{st} S$ " "(subst_domain  $\delta \cup \text{range\_vars } \delta) \cap \text{bvars}_{st} S = \{\}$ "
  shows "fv (t  $\cdot \delta$ )  $\subseteq \text{fv}_{st} (S \cdot_{st} \delta)$ "
⟨proof⟩

lemma strand_subst_not_dom_fixed:
  assumes "v  $\in \text{fv}_{st} S$ " and "v  $\notin \text{subst\_domain } \delta$ "
  shows "v  $\in \text{fv}_{st} (S \cdot_{st} \delta)$ "
⟨proof⟩

lemma strand_vars_unfold: "v  $\in \text{vars}_{st} S \implies \exists S' x S''. S = S'@x\#S'' \wedge v \in \text{vars}_{stp} x$ "
⟨proof⟩

lemma strand_fv_unfold: "v  $\in \text{fv}_{st} S \implies \exists S' x S''. S = S'@x\#S'' \wedge v \in \text{fv}_{stp} x$ "
⟨proof⟩

lemma subterm_if_in_strand_ik:
  "t  $\in \text{ik}_{st} S \implies \exists ts. \text{Receive } ts \in \text{set } S \wedge t \sqsubseteq_{\text{set}} \text{set } ts$ "
⟨proof⟩

lemma fv_subset_if_in_strand_ik:
  "t  $\in \text{ik}_{st} S \implies \text{fv } t \subseteq \bigcup (\text{set } (\text{map } \text{fv}_{rcv} S))$ "
⟨proof⟩

lemma fv_subset_if_in_strand_ik':
  "t  $\in \text{ik}_{st} S \implies \text{fv } t \subseteq \text{fv}_{st} S$ "
⟨proof⟩

lemma vars_subset_if_in_strand_ik2:
  "t  $\in \text{ik}_{st} S \implies \text{fv } t \subseteq \text{wfrestrictedvars}_{st} S$ "
⟨proof⟩

```

3.1.3 Lemmata: Simple Strands

```

lemma simple_Cons[dest]: "simple (s\#S)  $\implies$  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']]  $\implies$  simple (S@S'"
⟨proof⟩

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

```

lemma `not_simple_if_snd_fun`: " $\text{Fun } f \ T \in \text{set } ts \implies S = S'@Send\ ts\#S'' \implies \neg\text{simple } S$ "
 ⟨*proof*⟩

lemma `not_list_all_elim`: " $\neg\text{list_all } P \ A \implies \exists B \ x \ C. \ A = B@x\#C \wedge \neg P \ x \wedge \text{list_all } P \ B$ "
 ⟨*proof*⟩

lemma `not_simple_stp_elim`:
assumes " $\neg\text{simple}_{stp} \ x$ "
shows " $(\exists ts. \ x = Send\ ts \wedge (\nexists y. \ ts = [Var\ y])) \vee$
 $(\exists a \ t \ t'. \ x = Equality\ a \ t \ t') \vee$
 $(\exists X \ F. \ x = Inequality\ X \ F \wedge (\nexists \mathcal{I}. \ \text{ineq_model } \mathcal{I} \ X \ F))$ "
 ⟨*proof*⟩

lemma `not_simple_elim`:
assumes " $\neg\text{simple } S$ "
shows " $(\exists A \ B \ ts. \ S = A@Send\ ts\#B \wedge (\nexists x. \ ts = [Var\ x]) \wedge \text{simple } A) \vee$
 $(\exists A \ B \ a \ t \ t'. \ S = A@Equality\ a \ t \ t'\#B \wedge \text{simple } A) \vee$
 $(\exists A \ B \ X \ F. \ S = A@Inequality\ X \ F\#B \wedge (\nexists \mathcal{I}. \ \text{ineq_model } \mathcal{I} \ X \ F) \wedge \text{simple } A)$ "
 ⟨*proof*⟩

lemma `simple_snd_is_var`: " $\llbracket Send\ ts \in \text{set } S; \text{simple } S \rrbracket \implies \exists v. \ ts = [Var\ v]$ "
 ⟨*proof*⟩

3.1.4 Lemmata: Strand Measure

lemma `measure_st_wellfounded`: " $wf\ \text{measure}_{st}$ " ⟨*proof*⟩

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

lemma `strand_size_map_fun_lt[simp]`:
 $\text{size}_{st} (\text{map } Send1 \ X) < \text{size} (\text{Fun } f \ X)$
 $\text{size}_{st} (\text{map } Send1 \ X) < \text{size}_{st} [Send\ [Fun\ f \ X]]$
 $\text{size}_{st} (\text{map } Receive1 \ X) < \text{size}_{st} [Receive\ [Fun\ f \ X]]$
 $\text{size}_{st} [Send\ X] < \text{size}_{st} [Send\ [Fun\ f \ X]]$
 $\text{size}_{st} [Receive\ X] < \text{size}_{st} [Receive\ [Fun\ f \ X]]$
 ⟨*proof*⟩

lemma `strand_size_rm_fun_lt[simp]`:
 $\text{size}_{st} (S@S') < \text{size}_{st} (S@Send\ ts\#S')$
 $\text{size}_{st} (S@S') < \text{size}_{st} (S@Receive\ ts\#S')$
 ⟨*proof*⟩

lemma `strand_fv_card_map_fun_eq`:
 $\text{card} (fv_{st} (S@Send\ [Fun\ f \ X]\#S')) = \text{card} (fv_{st} (S@(\text{map } Send1 \ X)\#S'))$
 ⟨*proof*⟩

lemma `strand_fv_card_rm_fun_le[simp]`: " $\text{card} (fv_{st} (S@S')) \leq \text{card} (fv_{st} (S@Send\ [Fun\ f \ X]\#S'))$ "
 ⟨*proof*⟩

lemma `strand_fv_card_rm_eq_le[simp]`: " $\text{card} (fv_{st} (S@S')) \leq \text{card} (fv_{st} (S@Equality\ a \ t \ t'\#S'))$ "
 ⟨*proof*⟩

3.1.5 Lemmata: Well-formed Strands

lemma `wf_prefix[dest]`: " $wf_{st} \ V \ (S@S') \implies wf_{st} \ V \ S$ "
 ⟨*proof*⟩

lemma `wf_vars_mono[simp]`: " $wf_{st} \ V \ S \implies wf_{st} \ (V \cup W) \ S$ "
 ⟨*proof*⟩

lemma `wf_stI[intro]`: " $wf_{restrictedvars_{st}} \ S \subseteq V \implies wf_{st} \ V \ S$ "

<proof>

lemma *wf_{st}I'*[intro]: " $\bigcup (fv_{rcv} \setminus \text{set } S) \cup \bigcup (fv_{req} \text{ assign} \setminus \text{set } S) \subseteq V \implies wf_{st} V S$ "

<proof>

lemma *wf_append_exec*: " $wf_{st} V (S@S') \implies wf_{st} (V \cup wfvarsoccs_{st} S) S'$ "

<proof>

lemma *wf_append_suffix*:

" $wf_{st} V S \implies wf_{restrictedvars_{st}} S' \subseteq wf_{restrictedvars_{st}} S \cup V \implies wf_{st} V (S@S')$ "

<proof>

lemma *wf_append_suffix'*:

assumes " $wf_{st} V S$ "

and " $\bigcup (fv_{rcv} \setminus \text{set } S') \cup \bigcup (fv_{req} \text{ assign} \setminus \text{set } S') \subseteq wfvarsoccs_{st} S \cup V$ "

shows " $wf_{st} V (S@S')$ "

<proof>

lemma *wf_send_compose*: " $wf_{st} V (S@(\text{map Send1 } X)@S') = wf_{st} V (S@Send1 (\text{Fun } f X)\#S')$ "

<proof>

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

<proof>

lemma *wf_snd_append'*: " $wf_{st} V S \implies wf_{st} V (\text{Send } t\#S)$ "

<proof>

lemma *wf_rcv_append[dest]*: " $wf_{st} V (S@Receive t\#S') \implies wf_{st} V (S@S')$ "

<proof>

lemma *wf_rcv_append'[intro]*:

" $\llbracket wf_{st} V (S@S'); fv_{set} (\text{set } ts) \subseteq wf_{restrictedvars_{st}} S \cup V \rrbracket \implies wf_{st} V (S@Receive ts\#S')$ "

<proof>

lemma *wf_rcv_append''[intro]*:

" $\llbracket wf_{st} V S; fv_{set} (\text{set } ts) \subseteq \bigcup (\text{set } (\text{map } fv_{snd} S)) \rrbracket \implies wf_{st} V (S@[Receive ts])$ "

<proof>

lemma *wf_rcv_append'''[intro]*:

" $\llbracket wf_{st} V S; fv_{set} (\text{set } ts) \subseteq wf_{restrictedvars_{st}} S \cup V \rrbracket \implies wf_{st} V (S@[Receive ts])$ "

<proof>

lemma *wf_eq_append[dest]*:

" $wf_{st} V (S@Equality a t t'\#S') \implies fv t \subseteq wf_{restrictedvars_{st}} S \cup V \implies wf_{st} V (S@S')$ "

<proof>

lemma *wf_eq_append'[intro]*:

" $\llbracket wf_{st} V (S@S'); fv t' \subseteq wf_{restrictedvars_{st}} S \cup V \rrbracket \implies wf_{st} V (S@Equality a t t'\#S')$ "

<proof>

lemma *wf_eq_append''[intro]*:

" $\llbracket wf_{st} V (S@S'); fv t' \subseteq wfvarsoccs_{st} S \cup V \rrbracket \implies wf_{st} V (S@[Equality a t t']@S')$ "

<proof>

lemma *wf_eq_append'''[intro]*:

" $\llbracket wf_{st} V S; fv t' \subseteq wf_{restrictedvars_{st}} S \cup V \rrbracket \implies wf_{st} V (S@[Equality a t t'])$ "

<proof>

lemma *wf_eq_check_append[dest]*: " $wf_{st} V (S@Equality Check t t'\#S') \implies wf_{st} V (S@S')$ "

<proof>

lemma *wf_eq_check_append'[intro]*: " $wf_{st} V (S@S') \implies wf_{st} V (S@Equality Check t t'\#S')$ "

<proof>

lemma `wf_eq_check_append'` [intro]: " $wf_{st} V S \implies wf_{st} V (S@[Equality\ Check\ t\ t'])$ "
 <proof>

lemma `wf_ineq_append[dest]`: " $wf_{st} V (S@[Inequality\ X\ F\#S']) \implies wf_{st} V (S@S')$ "
 <proof>

lemma `wf_ineq_append'` [intro]: " $wf_{st} V (S@S') \implies wf_{st} V (S@[Inequality\ X\ F\#S'])$ "
 <proof>

lemma `wf_ineq_append''` [intro]: " $wf_{st} V S \implies wf_{st} V (S@[Inequality\ X\ F])$ "
 <proof>

lemma `wf_Receive1_prefix`:
 assumes " $wf_{st} X S$ "
 and " $fv_{set} (set\ ts) \subseteq X$ "
 shows " $wf_{st} X (map\ Receive1\ ts@S)$ "
 <proof>

lemma `wf_Send1_prefix`:
 assumes " $wf_{st} (X \cup fv_{set} (set\ ts)) S$ "
 shows " $wf_{st} X (map\ Send1\ ts@S)$ "
 <proof>

lemma `wf_rcv_fv_single[elim]`: " $wf_{st} V (Receive\ ts\#S') \implies fv_{set} (set\ ts) \subseteq V$ "
 <proof>

lemma `wf_rcv_fv`: " $wf_{st} V (S@Receive\ ts\#S') \implies fv_{set} (set\ ts) \subseteq wfvarsoccs_{st} S \cup V$ "
 <proof>

lemma `wf_eq_fv`: " $wf_{st} V (S@Equality\ Assign\ t\ t'\#S') \implies fv\ t' \subseteq wfvarsoccs_{st} S \cup V$ "
 <proof>

lemma `wf_simple_fv_occurrence`:
 assumes " $wf_{st} \{ \} S$ " "simple S " " $v \in wfrestrictedvars_{st} S$ "
 shows " $\exists S_{pre} S_{suf}. S = S_{pre}@Send\ [Var\ v]\#S_{suf} \wedge v \notin wfrestrictedvars_{st} S_{pre}$ "
 <proof>

lemma `Unifier_strand_fv_subset`:
 assumes `g_in_ik`: " $t \in ik_{st} S$ "
 and `delta`: "`Unifier` δ (Fun $f\ X$) t "
 and `disj`: " $bvars_{st} S \cap (subst_domain\ \delta \cup range_vars\ \delta) = \{ \}$ "
 shows " $fv\ (Fun\ f\ X \cdot \delta) \subseteq \bigcup (set\ (map\ fv_{rcv}\ (S \cdot_{st}\ \delta)))$ "
 <proof>

lemma `wf_st_induct'` [consumes 1, case_names Nil ConsSnd ConsRcv ConsEq ConsEq2 ConsIneq]:
 fixes `S::`"('a,'b) strand"
 assumes " $wf_{st} V S$ "
 " $P\ []$ "
 " $\bigwedge ts\ S. \llbracket wf_{st} V S; P\ S \rrbracket \implies P (S@[Send\ ts])$ "
 " $\bigwedge ts\ S. \llbracket wf_{st} V S; P\ S; fv_{set} (set\ ts) \subseteq V \cup wfvarsoccs_{st} S \rrbracket \implies P (S@[Receive\ ts])$ "
 " $\bigwedge t\ t'\ S. \llbracket wf_{st} V S; P\ S; fv\ t' \subseteq V \cup wfvarsoccs_{st} S \rrbracket \implies P (S@[Equality\ Assign\ t\ t'])$ "
 " $\bigwedge t\ t'\ S. \llbracket wf_{st} V S; P\ S \rrbracket \implies P (S@[Equality\ Check\ t\ t'])$ "
 " $\bigwedge X\ F\ S. \llbracket wf_{st} V S; P\ S \rrbracket \implies P (S@[Inequality\ X\ F])$ "
 shows " $P\ S$ "
 <proof>

lemma `wf_subst_apply`:
 " $wf_{st} V S \implies wf_{st} (fv_{set} (\delta \setminus V)) (S \cdot_{st} \delta)$ "
 <proof>

lemma `wf_unify`:
 assumes `wf`: " $wf_{st} V (S@Send\ [Fun\ f\ X]\#S')$ "

```

and g_in_ik: "t ∈ ikst S"
and δ: "Unifier δ (Fun f X) t"
and disj: "bvarsst (S@Send [Fun f X]#S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
shows "wfst (fvset (δ ` V)) ((S@S') .st δ)"
⟨proof⟩

lemma wf_equality:
  assumes wf: "wfst V (S@Equality ac t t'#S)"
  and δ: "mgu t t' = Some δ"
  and disj: "bvarsst (S@Equality ac t t'#S') ∩ (subst_domain δ ∪ range_vars δ) = {}"
  shows "wfst (fvset (δ ` V)) ((S@S') .st δ)"
⟨proof⟩

lemma wf_rcv_prefix_ground:
  "wfst {} ((map Receive M)@S) ⇒ varsst (map Receive M) = {}"
⟨proof⟩

lemma simple_wfvarsoccsst_is_fvsnd:
  assumes "simple S"
  shows "wfvarsoccsst S = ∪ (set (map fvsnd S))"
⟨proof⟩

lemma wfst_simple_induct[consumes 2, case_names Nil ConsSnd ConsRcv ConsIneq]:
  fixes S: "('a, 'b) strand"
  assumes "wfst V S" "simple S"
  "P []"
  "∧V S. [[wfst V S; simple S; P S]] ⇒ P (S@[Send [Var v]])"
  "∧ts S. [[wfst V S; simple S; P S; fvset (set ts) ⊆ V ∪ ∪ (set (map fvsnd S))]] ⇒ P
(S@[Receive ts])"
  "∧X F S. [[wfst V S; simple S; P S]] ⇒ P (S@[Inequality X F])"
  shows "P S"
⟨proof⟩

lemma wf_trm_stp_dom_fv_disjoint:
  "[[wfconstr S ∅; t ∈ trmsst S]] ⇒ subst_domain ∅ ∩ fv t = {}"
⟨proof⟩

lemma wf_constr_bvars_disj: "wfconstr S ∅ ⇒ (subst_domain ∅ ∪ range_vars ∅) ∩ bvarsst S = {}"
⟨proof⟩

lemma wf_constr_bvars_disj':
  assumes "wfconstr S ∅" "subst_domain δ ∪ range_vars δ ⊆ fvst S"
  shows "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}" (is ?A)
  and "(subst_domain ∅ ∪ range_vars ∅) ∩ bvarsst (S .st δ) = {}" (is ?B)
⟨proof⟩

lemma (in intruder_model) wf_simple_strand_first_Send_var_split:
  assumes "wfst {} S" "simple S" "∃ v ∈ wfrestrictedvarsst S. t · I = I v"
  shows "∃ v Spre Ssuf. S = Spre@Send [Var v]#Ssuf ∧ t · I = I v
  ∧ ¬(∃ w ∈ wfrestrictedvarsst Spre. t · I = I w)"
  (is "?P S")
⟨proof⟩

lemma (in intruder_model) wf_strand_first_Send_var_split:
  assumes "wfst {} S" "∃ v ∈ wfrestrictedvarsst S. t · I ⊆ I v"
  shows "∃ Spre Ssuf. ¬(∃ w ∈ wfrestrictedvarsst Spre. t · I ⊆ I w)
  ∧ ((∃ t'. S = Spre@Send t'#Ssuf ∧ t · I ⊆set set t' .set I)
  ∨ (∃ t' t''. S = Spre@Equality Assign t' t''#Ssuf ∧ t · I ⊆ t' · I))"
  (is "∃ Spre Ssuf. ?P Spre ∧ ?Q S Spre Ssuf")
⟨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 ts#S]_c = (λI. (∀t ∈ set ts. M ⊢_c t · I) ∧ [M; S]_c I)"
  | "[M; Receive ts#S]_c = (λI. [(set ts ·_set 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 " $\mathcal{I} \models_c \langle S, \vartheta \rangle \equiv (\vartheta \text{ supports } \mathcal{I} \wedge [\{ \}; S]_c \mathcal{I})$ "
abbreviation constr_sem_c' (<_ ⊢_c <_> 90) where " $\mathcal{I} \models_c \langle S \rangle \equiv \mathcal{I} \models_c \langle S, \text{Var} \rangle$ "

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 ts#S]_d = (λI. (∀t ∈ set ts. M ⊢ t · I) ∧ [M; S]_d I)"
  | "[M; Receive ts#S]_d = (λI. [(set ts ·_set 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 " $\mathcal{I} \models \langle S, \vartheta \rangle \equiv (\vartheta \text{ supports } \mathcal{I} \wedge [\{ \}; S]_d \mathcal{I})$ "
abbreviation constr_sem_d' (<_ ⊢ <_> 90) where " $\mathcal{I} \models \langle S \rangle \equiv \mathcal{I} \models \langle S, \text{Var} \rangle$ "

lemmas strand_sem_induct = strand_sem_c.induct[case_names Nil ConsSnd ConsRcv ConsEq ConsIneq]

Lemmata

lemma strand_sem_d_if_c: " $\mathcal{I} \models_c \langle S, \vartheta \rangle \implies \mathcal{I} \models \langle S, \vartheta \rangle$ "
<proof>

lemma strand_sem_mono_ik:
" $[M \subseteq M'; [M; S]_c \vartheta] \implies [M'; S]_c \vartheta$ " (is "[?A'; ?A'] ⇒ ?A")
" $[M \subseteq M'; [M; S]_d \vartheta] \implies [M'; S]_d \vartheta$ " (is "[?B'; ?B'] ⇒ ?B")
<proof>

context
begin

private lemma strand_sem_split_left:

" $[M; S@S']_c \vartheta \implies [M; S]_c \vartheta$ "
" $[M; S@S']_d \vartheta \implies [M; S]_d \vartheta$ "

<proof> lemma strand_sem_split_right:

" $[M; S@S']_c \vartheta \implies [M \cup (\text{ik}_{st} S \cdot_{set} \vartheta); S']_c \vartheta$ "
" $[M; S@S']_d \vartheta \implies [M \cup (\text{ik}_{st} S \cdot_{set} \vartheta); S']_d \vartheta$ "

<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]:

" $[[M; \text{map Send } T]_c \vartheta; ts \in \text{set } T] \implies [M; [\text{Send } ts]]_c \vartheta$ " (is "[?A'; ?A'] ⇒ ?A")
" $[[M; \text{map Send } T]_d \vartheta; ts \in \text{set } T] \implies [M; [\text{Send } ts]]_d \vartheta$ " (is "[?B'; ?B'] ⇒ ?B")
" $[[M; \text{map Send } T@S]_c \vartheta; ts \in \text{set } T] \implies [M; \text{Send } ts\#S]_c \vartheta$ " (is "[?C'; ?C'] ⇒ ?C")
" $[[M; \text{map Send } T@S]_d \vartheta; ts \in \text{set } T] \implies [M; \text{Send } ts\#S]_d \vartheta$ " (is "[?D'; ?D'] ⇒ ?D")

```

"[[M; map Send1 T']c ∅; t ∈ set T'] ⇒ [[M; [Send1 t]]c ∅" (is "[?E'; ?E'] ⇒ ?E")
"[[M; map Send1 T']d ∅; t ∈ set T'] ⇒ [[M; [Send1 t]]d ∅" (is "[?F'; ?F'] ⇒ ?F")
"[[M; map Send1 T'@S]c ∅; t ∈ set T'] ⇒ [[M; Send1 t#S]c ∅" (is "[?G'; ?G'] ⇒ ?G")
"[[M; map Send1 T'@S]d ∅; t ∈ set T'] ⇒ [[M; Send1 t#S]d ∅" (is "[?H'; ?H'] ⇒ ?H")
⟨proof⟩

```

lemma strand_sem_Send_map:

```

"⟨∧ts. ts ∈ set T ⇒ [[M; [Send ts]]c I⟩ ⇒ [[M; map Send T]c I"
"⟨∧ts. ts ∈ set T ⇒ [[M; [Send ts]]d I⟩ ⇒ [[M; map Send T]d I"
"⟨∧t. t ∈ set T' ⇒ [[M; [Send1 t]]c I⟩ ⇒ [[M; map Send1 T']c I"
"⟨∧t. t ∈ set T' ⇒ [[M; [Send1 t]]d I⟩ ⇒ [[M; map Send1 T']d I"
"[[M; map Send1 T']c I ↔ [[M; [Send T']]c I"
"[[M; map Send1 T']d I ↔ [[M; [Send T']]d I"
⟨proof⟩

```

lemma strand_sem_Receive_map:

```

"[[M; map Receive T]c I" " [[M; map Receive T]d I"
"[[M; map Receive1 T']c I" " [[M; map Receive1 T']d I"
"[[M; [Receive T']]c I" " [[M; [Receive T']]d I"
⟨proof⟩

```

lemma strand_sem_append[intro]:

```

"[[M; S]c ∅; [M ∪ (ikst S ·set ∅); S']c ∅] ⇒ [[M; S@S']c ∅"
"[[M; S]d ∅; [M ∪ (ikst S ·set ∅); S']d ∅] ⇒ [[M; S@S']d ∅"
⟨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 ∅) X F"
shows "ineq_model ∅ 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 ∅ X (F ·pairs δ)"
shows "ineq_model (δ ∘s ∅) X F"
⟨proof⟩

```

lemma ineq_model_ground_subst:

```

fixes F::"((('a,'b) term × ('a,'b) term) list)"
assumes "fvpairs F - set X ⊆ subst_domain δ"
and "ground (subst_range δ)"
and "ineq_model δ X F"
shows "ineq_model (δ ∘s ∅) X F"
⟨proof⟩

```

context

begin

private lemma strand_sem_subst_c:

```

assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
shows "[M; S]c (δ ∘s ∅) ⇒ [M; S ·st δ]c ∅"
⟨proof⟩ lemma strand_sem_subst_c':

```

```

assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
shows "[M; S ·st δ]c ∅ ⇒ [M; S]c (δ ∘s ∅)"

```

lemma strand_sem_subst_d:

```

assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
shows "[M; S]d (δ ∘s ∅) ⇒ [M; S ·st δ]d ∅"

```

lemma strand_sem_subst_d':

```

assumes "(subst_domain δ ∪ range_vars δ) ∩ bvarsst S = {}"
shows "[M; S ·st δ]d ∅ ⇒ [M; S]d (δ ∘s ∅)"
⟨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  $\delta$ : "(subst_domain  $\delta \cup \text{range\_vars } \delta) \cap \text{bvars}_{st} S = \{\}$ "
  shows " $\llbracket M; S \cdot_{st} \delta \rrbracket_c (\delta \circ_s \vartheta); \text{subst\_idem } \delta \rrbracket \implies \llbracket M; S \rrbracket_c (\delta \circ_s \vartheta)$ "
  <proof>
```

```
lemma strand_sem_subst_comp:
  assumes "(subst_domain  $\vartheta \cup \text{range\_vars } \vartheta) \cap \text{bvars}_{st} S = \{\}$ "
  and " $\llbracket M; S \rrbracket_c \delta$ " "subst_domain  $\vartheta \cap (\text{vars}_{st} S \cup \text{fv}_{set} M) = \{\}$ "
  shows " $\llbracket M; S \rrbracket_c (\vartheta \circ_s \delta)$ "
  <proof>
```

```
lemma strand_sem_c_imp_ineqs_neq:
  assumes " $\llbracket M; S \rrbracket_c \mathcal{I}$ " "Inequality  $X [(t, t')]$   $\in$  set  $S$ "
  shows " $t \neq t' \wedge (\forall \delta. \text{subst\_domain } \delta = \text{set } X \wedge \text{ground } (\text{subst\_range } \delta) \longrightarrow t \cdot \delta \neq t' \cdot \delta \wedge t \cdot \delta \cdot \mathcal{I} \neq t' \cdot \delta \cdot \mathcal{I})$ "
  <proof>
```

```
lemma strand_sem_c_imp_ineq_model:
  assumes " $\llbracket M; S \rrbracket_c \mathcal{I}$ " "Inequality  $X F \in$  set  $S$ "
  shows "ineq_model  $\mathcal{I} X F$ "
  <proof>
```

```
lemma strand_sem_wf_simple_fv_sat:
  assumes "wfst  $\{\}$   $S$ " "simple  $S$ " " $\llbracket \{\}; S \rrbracket_c \mathcal{I}$ "
  shows " $\bigwedge v. v \in \text{wfrestrictedvars}_{st} S \implies \text{ik}_{st} S \cdot_{set} \mathcal{I} \vdash_c \mathcal{I} v$ "
  <proof>
```

```
lemma strand_sem_wf_ik_or_assignment_rhs_fun_subterm:
  assumes "wfst  $\{\}$   $A$ " " $\llbracket \{\}; A \rrbracket_c \mathcal{I}$ " "Var  $x \in \text{ik}_{st} A$ " " $\mathcal{I} x = \text{Fun } f T$ "
  "ti  $\in$  set  $T$ " " $\neg \text{ik}_{st} A \cdot_{set} \mathcal{I} \vdash_c t_i$ " "interpretationsubst  $\mathcal{I}$ "
  obtains  $S$  where
  "Fun  $f S \in \text{subterms}_{set} (\text{ik}_{st} A) \vee \text{Fun } f S \in \text{subterms}_{set} (\text{assignment\_rhs}_{st} A)$ "
  "Fun  $f T = \text{Fun } f S \cdot \mathcal{I}$ "
  <proof>
```

```
lemma ineq_model_not_unif_is_sat_ineq:
  assumes " $\nexists \vartheta. \text{Unifier } \vartheta t t'$ "
  shows "ineq_model  $\mathcal{I} X [(t, t')]$ "
  <proof>
```

```
lemma strand_sem_not_unif_is_sat_ineq:
  assumes " $\nexists \vartheta. \text{Unifier } \vartheta t t'$ "
  shows " $\llbracket M; [\text{Inequality } X [(t, t')]] \rrbracket_c \mathcal{I}$ " " $\llbracket M; [\text{Inequality } X [(t, t')]] \rrbracket_d \mathcal{I}$ "
  <proof>
```

```
lemma ineq_model_singleI[intro]:
  assumes " $\forall \delta. \text{subst\_domain } \delta = \text{set } X \wedge \text{ground } (\text{subst\_range } \delta) \longrightarrow t \cdot \delta \cdot \mathcal{I} \neq t' \cdot \delta \cdot \mathcal{I}$ "
  shows "ineq_model  $\mathcal{I} X [(t, t')]$ "
  <proof>
```

```
lemma ineq_model_singleE:
  assumes "ineq_model  $\mathcal{I} X [(t, t')]$ "
  shows " $\forall \delta. \text{subst\_domain } \delta = \text{set } X \wedge \text{ground } (\text{subst\_range } \delta) \longrightarrow t \cdot \delta \cdot \mathcal{I} \neq t' \cdot \delta \cdot \mathcal{I}$ "
  <proof>
```

```
lemma ineq_model_single_iff:
  fixes  $F::((\text{'a}, \text{'b}) \text{ term} \times (\text{'a}, \text{'b}) \text{ term}) \text{ list}$ "
  shows "ineq_model  $\mathcal{I} X F \longleftrightarrow$ 
```

```

      ineq_model  $\mathcal{I}$  X [(Fun f (Fun c []#map fst F),Fun f (Fun c []#map snd F))]
    (is "?A  $\longleftrightarrow$  ?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  $\Rightarrow$  ('fun, 'var) strand  $\Rightarrow$  ('fun, 'var) subst  $\Rightarrow$  bool" (<[_;
_]_c'>)
  where
    "[M; []]_c' = ( $\lambda \mathcal{I}$ . True)"
  | "[M; Send ts#S]_c' = ( $\lambda \mathcal{I}$ . ( $\forall t \in \text{set } ts. M \cdot_{\text{set}} \mathcal{I} \vdash_c t \cdot \mathcal{I}$ )  $\wedge$  [M; S]_c'  $\mathcal{I}$ )"
  | "[M; Receive ts#S]_c' = [set ts  $\cup$  M; S]_c'"
  | "[M; Equality _ t t'#S]_c' = ( $\lambda \mathcal{I}$ .  $t \cdot \mathcal{I} = t' \cdot \mathcal{I} \wedge$  [M; S]_c'  $\mathcal{I}$ )"
  | "[M; Inequality X F#S]_c' = ( $\lambda \mathcal{I}$ . ineq_model  $\mathcal{I}$  X F  $\wedge$  [M; S]_c'  $\mathcal{I}$ )"

fun strand_sem_d:: "('fun, 'var) terms  $\Rightarrow$  ('fun, 'var) strand  $\Rightarrow$  ('fun, 'var) subst  $\Rightarrow$  bool" (<[_;
_]_d'>)
  where
    "[M; []]_d' = ( $\lambda \mathcal{I}$ . True)"
  | "[M; Send ts#S]_d' = ( $\lambda \mathcal{I}$ . ( $\forall t \in \text{set } ts. M \cdot_{\text{set}} \mathcal{I} \vdash t \cdot \mathcal{I}$ )  $\wedge$  [M; S]_d'  $\mathcal{I}$ )"
  | "[M; Receive ts#S]_d' = [set ts  $\cup$  M; S]_d'"
  | "[M; Equality _ t t'#S]_d' = ( $\lambda \mathcal{I}$ .  $t \cdot \mathcal{I} = t' \cdot \mathcal{I} \wedge$  [M; S]_d'  $\mathcal{I}$ )"
  | "[M; Inequality X F#S]_d' = ( $\lambda \mathcal{I}$ . ineq_model  $\mathcal{I}$  X F  $\wedge$  [M; S]_d'  $\mathcal{I}$ )"

lemma strand_sem_eq_defs:
  "[M; A]_c'  $\mathcal{I} =$  [M  $\cdot_{\text{set}} \mathcal{I}$ ; A]_c'  $\mathcal{I}$ "
  "[M; A]_d'  $\mathcal{I} =$  [M  $\cdot_{\text{set}} \mathcal{I}$ ; A]_d'  $\mathcal{I}$ "
  <proof>

lemma strand_sem_split'[dest]:
  "[M; S@S']_c'  $\vartheta \Longrightarrow$  [M; S]_c'  $\vartheta$ "
  "[M; S@S']_c'  $\vartheta \Longrightarrow$  [M  $\cup$  ikst S; S']_c'  $\vartheta$ "
  "[M; S@S']_d'  $\vartheta \Longrightarrow$  [M; S]_d'  $\vartheta$ "
  "[M; S@S']_d'  $\vartheta \Longrightarrow$  [M  $\cup$  ikst S; S']_d'  $\vartheta$ "
  <proof>

lemma strand_sem_append'[intro]:
  "[M; S]_c'  $\vartheta \Longrightarrow$  [M  $\cup$  ikst S; S']_c'  $\vartheta \Longrightarrow$  [M; S@S']_c'  $\vartheta$ "
  "[M; S]_d'  $\vartheta \Longrightarrow$  [M  $\cup$  ikst S; S']_d'  $\vartheta \Longrightarrow$  [M; S@S']_d'  $\vartheta$ "
  <proof>

end

```

3.1.8 Dual Strands

```

fun dual_st:: "('a, 'b) strand  $\Rightarrow$  ('a, 'b) strand" where
  "dual_st [] = []"
  | "dual_st (Receive t#S) = Send t#(dual_st S)"
  | "dual_st (Send t#S) = Receive t#(dual_st S)"
  | "dual_st (x#S) = x#(dual_st S)"

lemma dual_st_append: "dual_st (A@B) = (dual_st A)@(dual_st B)"
  <proof>

lemma dual_st_self_inverse: "dual_st (dual_st S) = S"
  <proof>

lemma dual_st_trms_eq: "trms_st (dual_st S) = trms_st S"
  <proof>

lemma dual_st_fv: "fv_st (dual_st A) = fv_st A"
  <proof>

```

```
lemma dual_st_bvars: "bvars_st (dual_st A) = bvars_st A"
<proof>
```

```
end
```

3.2 The 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 Send1 T)@S',ϑ)"
  | Unify: "[simple S; Fun f T' ∈ ik_st S; Some δ = mgu (Fun f T) (Fun f T')]"
    ⇒ (S@Send [Fun f T]#S',ϑ) ~ ((S@S') .st δ,ϑ o_s δ)"
  | Equality: "[simple S; Some δ = mgu t t']"
    ⇒ (S@Equality _ t t'#S',ϑ) ~ ((S@S') .st δ,ϑ o_s δ)"
```

A "pre-processing step" to be applied before constraint reduction. It transforms constraints such that exactly one message is transmitted in each message transmission step. It is sound and complete and preserves the various well-formedness properties required by the lazy intruder.

```
fun LI_preproc where
  "LI_preproc [] = []"
  | "LI_preproc (Send ts#S) = map Send1 ts@LI_preproc S"
  | "LI_preproc (Receive ts#S) = map Receive1 ts@LI_preproc S"
  | "LI_preproc (x#S) = x#LI_preproc S"
```

```
definition LI_preproc_prop where
  "LI_preproc_prop S ≡ ∀ ts. Send ts ∈ set S ∨ Receive ts ∈ set S ⇒ (∃ t. ts = [t])"
```

3.2.2 Lemmata: Preprocessing

```
lemma LI_preproc_preproc_prop:
  "LI_preproc_prop (LI_preproc S)"
<proof>
```

```
lemma LI_preproc_sem_eq:
  "[M; S]_c I ↔ [M; LI_preproc S]_c I" (is "?A ↔ ?B")
<proof>
```

```
lemma LI_preproc_sem_eq':
  "(I ⊢_c ⟨S, ϑ⟩) ↔ (I ⊢_c ⟨LI_preproc S, ϑ⟩)"
```

<proof>

lemma *LI_preproc_vars_eq:*

"fv_{st} (LI_preproc S) = fv_{st} S"

"bvars_{st} (LI_preproc S) = bvars_{st} S"

"vars_{st} (LI_preproc S) = vars_{st} S"

<proof>

lemma *LI_preproc_trms_eq:*

"trms_{st} (LI_preproc S) = trms_{st} S"

<proof>

lemma *LI_preproc_wf_{st}:*

assumes "wf_{st} X S"

shows "wf_{st} X (LI_preproc S)"

<proof>

lemma *LI_preproc_preserves_wellformedness:*

assumes "wf_{constr} S ϑ "

shows "wf_{constr} (LI_preproc S) ϑ "

<proof>

lemma *LI_preproc_prop_SendE:*

assumes "LI_preproc_prop S"

and "Send ts \in set S"

shows "($\exists x. ts = [Var x]$) \vee ($\exists f T. ts = [Fun f T]$)"

<proof>

lemma *LI_preproc_prop_split:*

"LI_preproc_prop (S@S') \longleftrightarrow LI_preproc_prop S \wedge LI_preproc_prop S'" (is "?A \longleftrightarrow ?B")

<proof>

3.2.3 Lemma: The Lazy Intruder is Well-founded

context

begin

private lemma *LI_compose_measure_lt:*

"((S@(map Send1 T)@S', ϑ_1), (S@Send [Fun f T]#S', ϑ_2)) \in measure_{st}"

<proof> **lemma** *LI_unify_measure_lt:*

assumes "Some $\delta = \text{mgu} (\text{Fun } f \text{ T}) t$ " "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 $\delta = \text{mgu } t \text{ t}'$ "

shows "((S@S') \cdot_{st} δ, ϑ_1), (S@Equality a t t'#S', ϑ_2)) \in measure_{st}"

<proof> **lemma** *LI_in_measure:* "(S₁, ϑ_1) \rightsquigarrow (S₂, ϑ_2) \implies ((S₂, ϑ_2), (S₁, ϑ_1)) \in measure_{st}"

<proof> **lemma** *LI_in_measure_trans:* "(S₁, ϑ_1) \rightsquigarrow^+ (S₂, ϑ_2) \implies ((S₂, ϑ_2), (S₁, 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:* " $\neg(\exists f. \forall i. f \text{ i } \rightsquigarrow^+ f (\text{Suc } i))$ "

<proof> **lemma** *LI_unify_finite:*

assumes "finite M"

shows "finite {(S@Send [Fun f T]#S', ϑ), ((S@S') \cdot_{st} $\delta, \vartheta \circ_s \delta$)} | δT ."

simple S \wedge Fun f T' \in M \wedge Some $\delta = \text{mgu} (\text{Fun } f \text{ T}) (\text{Fun } f \text{ T}')$ "

<proof>

end

3.2.4 Lemma: The Lazy Intruder Preserves Well-formedness

context

begin

```

private lemma LI_preserves_subst_wf_single:
  assumes "(S1, ϑ1) ↪ (S2, ϑ2)" "fvst S1 ∩ bvarsst S1 = {}" "wfsubst ϑ1"
  and "subst_domain ϑ1 ∩ varsst S1 = {}" "range_vars ϑ1 ∩ bvarsst S1 = {}"
  shows "fvst S2 ∩ bvarsst S2 = {}" "wfsubst ϑ2"
  and "subst_domain ϑ2 ∩ varsst S2 = {}" "range_vars ϑ2 ∩ bvarsst S2 = {}"
⟨proof⟩ lemma LI_preserves_subst_wf:
  assumes "(S1, ϑ1) ↪* (S2, ϑ2)" "fvst S1 ∩ bvarsst S1 = {}" "wfsubst ϑ1"
  and "subst_domain ϑ1 ∩ varsst S1 = {}" "range_vars ϑ1 ∩ bvarsst S1 = {}"
  shows "fvst S2 ∩ bvarsst S2 = {}" "wfsubst ϑ2"
  and "subst_domain ϑ2 ∩ varsst S2 = {}" "range_vars ϑ2 ∩ bvarsst S2 = {}"
⟨proof⟩

lemma LI_preserves_wellformedness:
  assumes "(S1, ϑ1) ↪* (S2, ϑ2)" "wfconstr S1 ϑ1"
  shows "wfconstr S2 ϑ2"
⟨proof⟩

lemma LI_preserves_trm_wf:
  assumes "(S, ϑ) ↪* (S', ϑ')" "wftrms (trmsst S)"
  shows "wftrms (trmsst S')"
⟨proof⟩

lemma LI_preproc_prop_subst:
  "LI_preproc_prop S ⟷ LI_preproc_prop (S ·st δ)"
⟨proof⟩

lemma LI_preserves_LI_preproc_prop:
  assumes "(S1, ϑ1) ↪* (S2, ϑ2)" "LI_preproc_prop S1"
  shows "LI_preproc_prop S2"
⟨proof⟩

end

```

3.2.5 Theorem: Soundness of the Lazy Intruder

```

context
begin
private lemma LI_soundness_single:
  assumes "wfconstr S1 ϑ1" "(S1, ϑ1) ↪ (S2, ϑ2)" "I ⊨c ⟨S2, ϑ2⟩"
  shows "I ⊨c ⟨S1, ϑ1⟩"
⟨proof⟩

theorem LI_soundness:
  assumes "wfconstr S1 ϑ1" "(LI_preproc S1, ϑ1) ↪* (S2, ϑ2)" "I ⊨c ⟨S2, ϑ2⟩"
  shows "I ⊨c ⟨S1, ϑ1⟩"
⟨proof⟩
end

```

3.2.6 Theorem: Completeness of the Lazy Intruder

```

context
begin
private lemma LI_completeness_single:
  assumes "wfconstr S1 ϑ1" "I ⊨c ⟨S1, ϑ1⟩" "¬simple S1" "LI_preproc_prop S1"
  shows "∃ S2 ϑ2. (S1, ϑ1) ↪ (S2, ϑ2) ∧ (I ⊨c ⟨S2, ϑ2⟩)"
⟨proof⟩

theorem LI_completeness:
  assumes "wfconstr S1 ϑ1" "I ⊨c ⟨S1, ϑ1⟩"
  shows "∃ S2 ϑ2. (LI_preproc S1, ϑ1) ↪* (S2, ϑ2) ∧ simple S2 ∧ (I ⊨c ⟨S2, ϑ2⟩)"
⟨proof⟩
end

```

3.2.7 Corollary: Soundness and Completeness as a Single Theorem

```

corollary LI_soundness_and_completeness:
  assumes "wf_constr S1  $\vartheta_1$ "
  shows " $\mathcal{I} \models_c \langle S_1, \vartheta_1 \rangle \longleftrightarrow (\exists S_2 \vartheta_2. (LI\_preproc S_1, \vartheta_1) \rightsquigarrow^* (S_2, \vartheta_2) \wedge simple S_2 \wedge (\mathcal{I} \models_c \langle S_2, \vartheta_2 \rangle))$ "
<proof>

end

end

```

3.3 The 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  $\Rightarrow$  ('f, 'v) term_type" where
  "TAtom a  $\equiv$  Var a"

```

```

abbreviation (input) TComp::"'f, ('f, 'v) term_type list]  $\Rightarrow$  ('f, 'v) term_type" where
  "TComp f ts  $\equiv$  Fun f ts"

```

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  $\Rightarrow$  nat"
  and public::"'fun  $\Rightarrow$  bool"
  and Ana::"'(fun, 'var) term  $\Rightarrow$  (('fun, 'var) term list  $\times$  ('fun, 'var) term list)"
  +
  fixes  $\Gamma$ ::"'(fun, 'var) term  $\Rightarrow$  ('fun, 'atom::finite) term_type"
  assumes const_type: " $\bigwedge c. arity c = 0 \implies \exists a. \forall ts. \Gamma (Fun c ts) = TAtom a$ "
  and fun_type: " $\bigwedge f ts. arity f > 0 \implies \Gamma (Fun f ts) = TComp f (map \Gamma ts)$ "
  and  $\Gamma\_wf$ : " $\bigwedge x f ts. TComp f ts \sqsubseteq \Gamma (Var x) \implies arity f > 0$ "
  " $\bigwedge x. wf_{trm} (\Gamma (Var x))$ "
begin

```

3.3.1 Definitions

The set of atomic types

```

abbreviation " $\mathfrak{A}_a \equiv UNIV::('atom set)$ "

```

Well-typed substitutions

```

definition wt_subst where
  "wt_subst  $\sigma \equiv (\forall v. \Gamma (Var v) = \Gamma (\sigma v))$ "

```

The set of sub-message patterns (SMP)

```

inductive_set SMP::"'(fun, 'var) terms  $\Rightarrow$  ('fun, 'var) terms" for M where
  MP[intro]: "t  $\in$  M  $\implies$  t  $\in$  SMP M"
| Subterm[intro]: " $\llbracket t \in SMP M; t' \sqsubseteq t \rrbracket \implies t' \in SMP M$ "
| Substitution[intro]: " $\llbracket t \in SMP M; wt\_subst \delta; wf_{trms} (subst\_range \delta) \rrbracket \implies (t \cdot \delta) \in SMP M$ "
| Ana[intro]: " $\llbracket t \in SMP M; Ana t = (K, T); k \in set K \rrbracket \implies k \in SMP M$ "

```

Type-flaw resistance for sets: Unifiable sub-message patterns must have the same type (unless they are variables)

```

definition tfr_set where
  "tfr_set M  $\equiv (\forall s \in SMP M - (Var \backslash \mathcal{V}). \forall t \in SMP M - (Var \backslash \mathcal{V}). (\exists \delta. Unifier \delta s t) \longrightarrow \Gamma s = \Gamma t)$ "

```

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))))"
  <proof>

```

```

lemma Γ_wf': "TComp f T ⊆ Γ t ⇒ arity f > 0"
  <proof>

```

```

lemma Γ_wf': "wftrm t ⇒ wftrm (Γ t)"
  <proof>

```

```

lemma fun_type_inv: assumes "Γ t = TComp f T" shows "arity f > 0"
  <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': "∀c ∈ C. ∃a ∈ Σa. ∀X. Γ (Fun c X) = TAtom a" <proof>

```

```

lemma fun_type': "∀f ∈ Σf. ∀X. Γ (Fun f X) = TComp f (map Γ X)" <proof>

```

```

lemma fun_type_id_eq: "Γ (Fun f X) = TComp g Y ⇒ f = g"
  <proof>

```

```

lemma fun_type_length_eq: "Γ (Fun f X) = TComp g Y ⇒ length X = length Y"
  <proof>

```

```

lemma pgwt_type_map:
  assumes "public_ground_wf_term t"
  shows "Γ t = TAtom a ⇒ ∃f. t = Fun f []" "Γ t = TComp g Y ⇒ ∃X. t = Fun g X ∧ map Γ X = Y"
  <proof>

```

```

lemma wt_subst_Var[simp]: "wtsubst Var" <proof>

```

```

lemma wt_subst_trm: "(∧v. v ∈ fv t ⇒ Γ (Var v) = Γ (∂ v)) ⇒ Γ t = Γ (t · ∂)"
  <proof>

```

lemma `wt_subst_trm'`: " $\llbracket wt_{subst} \sigma; \Gamma s = \Gamma t \rrbracket \implies \Gamma (s \cdot \sigma) = \Gamma (t \cdot \sigma)$ "
 <proof>

lemma `wt_subst_trm''`: " $wt_{subst} \sigma \implies \Gamma t = \Gamma (t \cdot \sigma)$ "
 <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 : " $wt_{subst} \vartheta$ " " $wf_{trms} (subst_range \vartheta)$ "
 and x : " $\Gamma (Var x) = TAtom a$ "
 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 " $\Gamma (Var x) = \Gamma t$ "
 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 : " $wt_{subst} \delta$ " " $wf_{trms} (subst_range \delta)$ "
 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 " $wf_{trm} t$ " " $\Gamma t = TComp f T$ "
 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_termtyp_subterm:

assumes "x ∈ fv (∅ y)"
 and "wt_{subst} ∅"
 and "wf_{trm} (∅ y)"
 shows "Γ (Var x) ⊆ Γ (Var y)"
 ⟨proof⟩

lemma wt_subst_fv_{set}_termtyp_subterm:

assumes "x ∈ fv_{set} (∅ ` Y)"
 and "wt_{subst} ∅"
 and "wf_{trms} (subst_range ∅)"
 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 ⇒ trmsstp x ⊆ SMP (trmsst S)" <proof>

lemma SMP_subset_I:
  assumes M: "∀ t ∈ M. ∃ s δ. s ∈ N ∧ wtsubst δ ∧ wftrms (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 (trmsst (S@S')) = SMP (trmsst S) ∪ SMP (trmsst 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 (trmsst (x#S)) = SMP (trmsst [x]) ∪ SMP (trmsst S)"
<proof>

lemma SMP_Nil[simp]: "SMP (trmsst []) = {}"
<proof>

lemma SMP_subset_union_eq: assumes "M ⊆ SMP N" shows "SMP N = SMP (M ∪ N)"
<proof>

lemma SMP_subterms_subset: "subtermsset M ⊆ SMP M"
<proof>

lemma SMP_SMP_subset: "N ⊆ SMP M ⇒ SMP N ⊆ SMP M"
<proof>

lemma wt_subst_rm_vars: "wtsubst δ ⇒ wtsubst (rm_vars X δ)"
<proof>

lemma wt_subst_SMP_subset:
  assumes "trmsst S ⊆ SMP S'" "wtsubst δ" "wftrms (subst_range δ)"
  shows "trmsst (S ·st δ) ⊆ SMP S'"
<proof>

lemma MP_subset_SMP: "⋃ (trmsstp ` set S) ⊆ SMP (trmsst S)" "trmsst S ⊆ SMP (trmsst S)" "M ⊆ SMP M"
<proof>

lemma SMP_fun_map_snd_subset: "SMP (trmsst (map Send1 X)) ⊆ SMP (trmsst [Send1 (Fun f X)])"
<proof>

lemma SMP_wt_subst_subset:
  assumes "t ∈ SMP (M ·set I)" "wtsubst I" "wftrms (subst_range I)"
  shows "t ∈ SMP M"
<proof>

lemma SMP_wt_instances_subset:
  assumes "∀ t ∈ M. ∃ s ∈ N. ∃ δ. t = s · δ ∧ wtsubst δ ∧ wftrms (subst_range δ)"
  and "t ∈ SMP M"
  shows "t ∈ SMP N"
<proof>

```

lemma *SMP_consts*:

assumes " $\forall t \in M. \exists c. t = \text{Fun } c \ []$ "
 and " $\forall t \in M. \text{Ana } t = ([], [])$ "
 shows " $\text{SMP } M = M$ "

<proof>

lemma *SMP_subterms_eq*:

" $\text{SMP } (\text{subterms}_{\text{set}} M) = \text{SMP } M$ "

<proof>

lemma *SMP_funs_term*:

assumes $t: "t \in \text{SMP } M"$ " $f \in \text{funs_term } t \vee (\exists x \in \text{fv } t. f \in \text{funs_term } (\Gamma (\text{Var } x)))$ "
 and $f: "arity\ f > 0"$
 and $M: "wf_{trms}\ M"$
 and $\text{Ana}_f: "\bigwedge s\ K\ T. \text{Ana } s = (K, T) \implies f \in \bigcup (\text{funs_term } \setminus \text{set } K) \implies f \in \text{funs_term } s"$
 shows " $f \in \bigcup (\text{funs_term } \setminus M) \vee (\exists x \in \text{fv}_{\text{set}} M. f \in \text{funs_term } (\Gamma (\text{Var } x)))$ "

<proof>

lemma *id_type_eq*:

assumes " $\Gamma (\text{Fun } f\ X) = \Gamma (\text{Fun } g\ Y)$ "
 shows " $f \in \mathcal{C} \implies g \in \mathcal{C}$ " " $f \in \Sigma_f \implies g \in \Sigma_f$ "

<proof>

lemma *fun_type_arg_cong*:

assumes " $f \in \Sigma_f$ " " $g \in \Sigma_f$ " " $\Gamma (\text{Fun } f\ (x\#X)) = \Gamma (\text{Fun } g\ (y\#Y))$ "
 shows " $\Gamma\ x = \Gamma\ y$ " " $\Gamma (\text{Fun } f\ X) = \Gamma (\text{Fun } g\ Y)$ "

<proof>

lemma *fun_type_arg_cong'*:

assumes " $f \in \Sigma_f$ " " $g \in \Sigma_f$ " " $\Gamma (\text{Fun } f\ (X\@x\#X')) = \Gamma (\text{Fun } g\ (Y\@y\#Y'))$ " " $\text{length } X = \text{length } Y$ "
 shows " $\Gamma\ x = \Gamma\ y$ "

<proof>

lemma *fun_type_param_idx*: " $\Gamma (\text{Fun } f\ T) = \text{Fun } g\ S \implies i < \text{length } T \implies \Gamma (T\ !\ i) = S\ !\ i$ "

<proof>

lemma *fun_type_param_ex*:

assumes " $\Gamma (\text{Fun } f\ T) = \text{Fun } g\ (\text{map } \Gamma\ S)$ " " $t \in \text{set } S$ "
 shows " $\exists s \in \text{set } T. \Gamma\ s = \Gamma\ t$ "

<proof>

lemma *tfr_stp_all_split*:

" $\text{list_all } \text{tfr}_{\text{stp}}\ (x\#S) \implies \text{list_all } \text{tfr}_{\text{stp}}\ [x]$ "
 " $\text{list_all } \text{tfr}_{\text{stp}}\ (x\#S) \implies \text{list_all } \text{tfr}_{\text{stp}}\ S$ "
 " $\text{list_all } \text{tfr}_{\text{stp}}\ (S\@S') \implies \text{list_all } \text{tfr}_{\text{stp}}\ S$ "
 " $\text{list_all } \text{tfr}_{\text{stp}}\ (S\@S') \implies \text{list_all } \text{tfr}_{\text{stp}}\ S'$ "
 " $\text{list_all } \text{tfr}_{\text{stp}}\ (S\@x\#S') \implies \text{list_all } \text{tfr}_{\text{stp}}\ (S\@S')$ "

<proof>

lemma *tfr_stp_all_append*:

assumes " $\text{list_all } \text{tfr}_{\text{stp}}\ S$ " " $\text{list_all } \text{tfr}_{\text{stp}}\ S'$ "
 shows " $\text{list_all } \text{tfr}_{\text{stp}}\ (S\@S')$ "

<proof>

lemma *tfr_stp_all_wt_subst_apply*:

assumes " $\text{list_all } \text{tfr}_{\text{stp}}\ S$ "
 and $\vartheta: "wt_{\text{subst}}\ \vartheta"$ " $wf_{trms}\ (\text{subst_range } \vartheta)$ "
 " $\text{bvars}_{\text{st}}\ S \cap \text{range_vars } \vartheta = \{\}$ "
 shows " $\text{list_all } \text{tfr}_{\text{stp}}\ (S \cdot_{\text{st}} \vartheta)$ "

<proof>

lemma *tfr_stp_all_same_type*:

```

"list_all tfr_stp (S@Equality a t t'#S')  $\implies$  Unifier  $\delta$  t t'  $\implies$   $\Gamma$  t =  $\Gamma$  t'"
<proof>

lemma tfr_subset:
  " $\bigwedge$ A B. tfr_set (A  $\cup$  B)  $\implies$  tfr_set A"
  " $\bigwedge$ A B. tfr_set B  $\implies$  A  $\subseteq$  B  $\implies$  tfr_set A"
  " $\bigwedge$ A B. tfr_set B  $\implies$  SMP A  $\subseteq$  SMP B  $\implies$  tfr_set A"
<proof>

lemma tfr_empty[simp]: "tfr_set {}"
<proof>

lemma tfr_consts_mono:
  assumes " $\forall$ t  $\in$  M.  $\exists$ c. t = Fun c []"
  and " $\forall$ t  $\in$  M. Ana t = ([], [])"
  and "tfr_set N"
  shows "tfr_set (N  $\cup$  M)"
<proof>

lemma dual_st_tfr_stp: "list_all tfr_stp S  $\implies$  list_all tfr_stp (dual_st S)"
<proof>

lemma subst_var_inv_wt:
  assumes "wt_subst  $\delta$ "
  shows "wt_subst (subst_var_inv  $\delta$  X)"
<proof>

lemma subst_var_inv_wf_trms:
  "wf_trms (subst_range (subst_var_inv  $\delta$  X))"
<proof>

lemma unify_list_wt_if_same_type:
  assumes "Unification.unify E B = Some U" " $\forall$ (s,t)  $\in$  set E. wf_trm s  $\wedge$  wf_trm t  $\wedge$   $\Gamma$  s =  $\Gamma$  t"
  and " $\forall$ (v,t)  $\in$  set B.  $\Gamma$  (Var v) =  $\Gamma$  t"
  shows " $\forall$ (v,t)  $\in$  set U.  $\Gamma$  (Var v) =  $\Gamma$  t"
<proof>

lemma mgu_wt_if_same_type:
  assumes "mgu s t = Some  $\sigma$ " "wf_trm s" "wf_trm t" " $\Gamma$  s =  $\Gamma$  t"
  shows "wt_subst  $\sigma$ "
<proof>

lemma wt_Unifier_if_Unifier:
  assumes s_t: "wf_trm s" "wf_trm t" " $\Gamma$  s =  $\Gamma$  t"
  and  $\delta$ : "Unifier  $\delta$  s t"
  shows " $\exists$  $\vartheta$ . Unifier  $\vartheta$  s t  $\wedge$  wt_subst  $\vartheta$   $\wedge$  wf_trms (subst_range  $\vartheta$ )"
<proof>

end

```

3.3.3 Automatically Proving Type-Flaw Resistance

Definitions: Variable Renaming

abbreviation "max_var t \equiv Max (insert 0 (snd ` fv t))"

abbreviation "max_var_set X \equiv Max (insert 0 (snd ` X))"

definition "var_rename n v \equiv Var (fst v, snd v + Suc n)"

definition "var_rename_inv n v \equiv 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  $\Gamma$  t s  $\equiv$  (
   $\Gamma$  t =  $\Gamma$  s  $\wedge$  (case mgu t s of
    None  $\Rightarrow$  False
  | Some  $\delta \Rightarrow$  inj_on  $\delta$  (fv t)  $\wedge$  ( $\forall x \in$  fv t. is_Var ( $\delta$  x))))"
```

definition *has_all_wt_instances_of* **where**

```
"has_all_wt_instances_of  $\Gamma$  N M  $\equiv$   $\forall t \in$  N.  $\exists s \in$  M. is_wt_instance_of_cond  $\Gamma$  t s"
```

This function computes a finite representation of the set of sub-message patterns

definition *SMP0* **where**

```
"SMP0 Ana  $\Gamma$  M  $\equiv$  let
  f =  $\lambda$ t. Fun (the_Fun ( $\Gamma$  t)) (map Var (zip (args ( $\Gamma$  t)) [0.. $\text{length}$  (args ( $\Gamma$  t))]));
  g =  $\lambda$ M'. map f (filter ( $\lambda$ t. is_Var t  $\wedge$  is_Fun ( $\Gamma$  t)) M')@
    concat (map (fst  $\circ$  Ana) M')@concat (map subterms_list M');
  h = remdups  $\circ$  g
  in while ( $\lambda$ A. set (h A)  $\neq$  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  $\Gamma$  p n (Fun f T) = (let  $\tau = \Gamma$  (Fun f T)
  in if p  $\tau$  then (Var ( $\tau$ , n), Suc n)
  else let (T',n') = foldr ( $\lambda$ t (S,m). let (t',m') = generalize_term  $\Gamma$  p m t in (t'#S,m'))
    T ([],n)
  in (Fun f T', n'))"
```

definition *generalize_terms* **where**

```
"generalize_terms  $\Gamma$  p  $\equiv$  map (fst  $\circ$  generalize_term  $\Gamma$  p 0)"
```

definition *remove_superfluous_terms* **where**

```
"remove_superfluous_terms  $\Gamma$  T  $\equiv$ 
  let
    f =  $\lambda$ S t R.  $\exists s \in$  set S - R. s  $\neq$  t  $\wedge$  is_wt_instance_of_cond  $\Gamma$  t s;
    g =  $\lambda$ S t (U,R). if f S t R then (U, insert t R) else (t#U, R);
    h =  $\lambda$ S. remdups (fst (foldr (g S) S ([],{})))
  in while ( $\lambda$ S. h S  $\neq$  S) h T"
```

Definitions: Checking Type-Flaw Resistance

definition *is_TComp_var_instance_closed* **where**

```
"is_TComp_var_instance_closed  $\Gamma$  M  $\equiv$   $\forall x \in$  fvset M. is_Fun ( $\Gamma$  (Var x))  $\longrightarrow$ 
  ( $\exists t \in$  M. is_Fun t  $\wedge$   $\Gamma$  t =  $\Gamma$  (Var x)  $\wedge$  list_all is_Var (args t)  $\wedge$  distinct (args t))"
```

definition *finite_SMP_representation* **where**

```
"finite_SMP_representation arity Ana  $\Gamma$  M  $\equiv$ 
  (M = {}  $\vee$  card M > 0)  $\wedge$ 
  wftrms' arity M  $\wedge$ 
  has_all_wt_instances_of  $\Gamma$  (subtermsset M) M  $\wedge$ 
  has_all_wt_instances_of  $\Gamma$  ( $\bigcup$  ((set  $\circ$  fst  $\circ$  Ana) ` M)) M  $\wedge$ 
  is_TComp_var_instance_closed  $\Gamma$  M"
```

definition *comp_tfr_{set}* **where**

```
"comp_tfrset arity Ana  $\Gamma$  M  $\equiv$ 
  finite_SMP_representation arity Ana  $\Gamma$  M  $\wedge$ 
  (let  $\delta =$  var_rename (max_var_set (fvset M))
  in  $\forall s \in$  M.  $\forall t \in$  M. is_Fun s  $\wedge$  is_Fun t  $\wedge$   $\Gamma$  s  $\neq$   $\Gamma$  t  $\longrightarrow$  mgu s (t  $\cdot$   $\delta$ ) = None)"
```

fun *comp_tfr_{stp}* **where**

```
"comp_tfrstp  $\Gamma$  ( $\langle \_ : t \doteq t' \rangle_{st}$ ) = (mgu t t'  $\neq$  None  $\longrightarrow$   $\Gamma$  t =  $\Gamma$  t')"
| "comp_tfrstp  $\Gamma$  ( $\forall X(\forall \neq : F)_{st}$ ) = (
  ( $\forall x \in$  fvpairs F - set X. is_Var ( $\Gamma$  (Var x)))  $\vee$ 
  ( $\forall u \in$  subtermsset (trmspairs F)."
```

```

    is_Fun u  $\longrightarrow$  (args u = []  $\vee$  ( $\exists s \in \text{set (args u)}. s \notin \text{Var} \setminus \text{set X}$ )))"
  | "comp_tfr_stp _ _ = True"

```

definition *comp_tfr_{st}* where

```

"comp_tfrst arity Ana  $\Gamma$  M S  $\equiv$ 
  list_all (comp_tfrstp  $\Gamma$ ) S  $\wedge$ 
  list_all (wftrm 'arity) (trms_listst S)  $\wedge$ 
  has_all_wt_instances_of  $\Gamma$  (trmsst S) M  $\wedge$ 
  comp_tfrset arity Ana  $\Gamma$  M"

```

Small Lemmata

lemma *max_var_set_mono*:

```

  assumes "finite N"
  and "M  $\subseteq$  N"
  shows "max_var_set M  $\leq$  max_var_set N"

```

<proof>

lemma *less_Suc_max_var_set*:

```

  assumes z: "z  $\in$  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  $\Gamma$  M"
  shows "wftrms M"
  and "has_all_wt_instances_of  $\Gamma$  (subtermsset M) M"
  and "has_all_wt_instances_of  $\Gamma$  ( $\bigcup$ ((set  $\circ$  fst  $\circ$  Ana)  $\setminus$  M)) M"
  and "is_TComp_var_instance_closed  $\Gamma$  M"
  and "finite M"

```

<proof>

lemma (in *typed_model*) *is_wt_instance_of_condD*:

```

  assumes t_instance_s: "is_wt_instance_of_cond  $\Gamma$  t s"
  obtains  $\delta$  where
    " $\Gamma$  t =  $\Gamma$  s" "mgu t s = Some  $\delta$ "
    "inj_on  $\delta$  (fv t)" " $\delta \setminus$  (fv t)  $\subseteq$  range Var"

```

<proof>

lemma (in *typed_model*) *is_wt_instance_of_condD'*:

```

  assumes t_wf_trm: "wftrm t"
  and s_wf_trm: "wftrm s"
  and t_instance_s: "is_wt_instance_of_cond  $\Gamma$  t s"
  shows " $\exists \delta. \text{wt}_{\text{subst}} \delta \wedge \text{wf}_{\text{trms}} (\text{subst\_range } \delta) \wedge t = s \cdot \delta$ "

```

<proof>

lemma (in *typed_model*) *is_wt_instance_of_condD''*:

```

  assumes s_wf_trm: "wftrm s"
  and t_instance_s: "is_wt_instance_of_cond  $\Gamma$  t s"
  and t_var: "t = Var x"
  shows " $\exists y. s = \text{Var } y \wedge \Gamma (\text{Var } y) = \Gamma (\text{Var } x)$ "

```

<proof>

lemma (in *typed_model*) *has_all_wt_instances_ofD*:

```

  assumes N_instance_M: "has_all_wt_instances_of  $\Gamma$  N M"
  and t_in_N: "t  $\in$  N"
  obtains s  $\delta$  where
    "s  $\in$  M" " $\Gamma$  t =  $\Gamma$  s" "mgu t s = Some  $\delta$ "
    "inj_on  $\delta$  (fv t)" " $\delta \setminus$  (fv t)  $\subseteq$  range Var"

```

<proof>

lemma (in *typed_model*) *has_all_wt_instances_ofD'*:

3 The Typing Result for Non-Stateful Protocols

```

assumes N_wf_trms: "wf_trms N"
  and M_wf_trms: "wf_trms M"
  and N_instance_M: "has_all_wt_instances_of  $\Gamma$  N M"
  and t_in_N: "t  $\in$  N"
shows " $\exists \delta. \text{wt}_{\text{subst}} \delta \wedge \text{wf}_{\text{trms}} (\text{subst\_range } \delta) \wedge t \in M \cdot_{\text{set}} \delta$ "
<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  $\Gamma$  N M"
    and t_in_N: "Var x  $\in$  N"
  shows " $\exists y. \text{Var } y \in M \wedge \Gamma (\text{Var } y) = \Gamma (\text{Var } x)$ "
<proof>

```

```

lemma (in typed_model) has_all_instances_of_if_subset:
  assumes "N  $\subseteq$  M"
  shows "has_all_wt_instances_of  $\Gamma$  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  $\Gamma$  N M"
    and t_in_N: "t  $\in$  N"
  shows "t  $\in$  SMP M"
<proof>

```

Lemma: Proving Type-Flaw Resistance

```

locale typed_model' = typed_model arity public Ana  $\Gamma$ 
  for arity::"fun  $\Rightarrow$  nat"
  and public::"fun  $\Rightarrow$  bool"
  and Ana::"('fun, (('fun, 'atom)::finite) term_type  $\times$  nat)) term
     $\Rightarrow$  (('fun, (('fun, 'atom) term_type  $\times$  nat)) term list
       $\times$  ('fun, (('fun, 'atom) term_type  $\times$  nat)) term list)"
  and  $\Gamma$ ::"('fun, (('fun, 'atom) term_type  $\times$  nat)) term  $\Rightarrow$  ('fun, 'atom) term_type"
+
  assumes  $\Gamma$ _Var_fst: " $\bigwedge \tau n m. \Gamma (\text{Var } (\tau, n)) = \Gamma (\text{Var } (\tau, m))$ "
  and Ana_const: " $\bigwedge c T. \text{arity } c = 0 \implies \text{Ana } (\text{Fun } c T) = ([], [])$ "
  and Ana_subst'_or_Ana_keys_subterm:
    " $(\forall f T \delta K R. \text{Ana } (\text{Fun } f T) = (K, R) \longrightarrow \text{Ana } (\text{Fun } f T \cdot \delta) = (K \cdot_{\text{list}} \delta, R \cdot_{\text{list}} \delta)) \vee$ 
     $(\forall t K R k. \text{Ana } t = (K, R) \longrightarrow k \in \text{set } K \longrightarrow k \sqsubseteq t)$ "

```

begin

```

lemma var_rename_inv_comp: "t  $\cdot$  (var_rename n  $\circ_s$  var_rename_inv n) = t"
<proof>

```

```

lemma var_rename_fv_disjoint:
  "fv s  $\cap$  fv (t  $\cdot$  var_rename (max_var s)) = {}"
<proof>

```

```

lemma var_rename_fv_set_disjoint:
  assumes "finite M" "s  $\in$  M"
  shows "fv s  $\cap$  fv (t  $\cdot$  var_rename (max_var_set (fv_set M))) = {}"
<proof>

```

```

lemma var_rename_fv_set_disjoint':
  assumes "finite M"
  shows "fv_set M  $\cap$  fv_set (N  $\cdot_{\text{set}}$  var_rename (max_var_set (fv_set M))) = {}"
<proof>

```

```

lemma var_rename_is_renaming[simp]:

```

```

"subst_range (var_rename n)  $\subseteq$  range Var"
"subst_range (var_rename_inv n)  $\subseteq$  range Var"
⟨proof⟩

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

lemma var_rename_wt':
  assumes "wt_subst  $\delta$ " "s = m  $\cdot$   $\delta$ "
  shows "wt_subst (var_rename_inv n  $\circ_s$   $\delta$ )" "s = m  $\cdot$  var_rename n  $\cdot$  var_rename_inv n  $\circ_s$   $\delta$ "
⟨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:
  "( $\forall f T. \text{Fun } f T \in M \longrightarrow P f T$ )  $\longleftrightarrow$  ( $\forall u \in M. \text{case } u \text{ of Fun } f T \Rightarrow P f T \mid \_ \Rightarrow \text{True}$ )"
  "( $\forall f T. \text{Fun } f T \in M \longrightarrow P f T$ )  $\longleftrightarrow$  ( $\forall u \in M. \text{is\_Fun } u \longrightarrow P (\text{the\_Fun } u) (\text{args } u)$ )"
⟨proof⟩

lemma is_TComp_var_instance_closedD:
  assumes x: " $\exists y \in \text{fv\_set } M. \Gamma (\text{Var } x) = \Gamma (\text{Var } y)$ " " $\Gamma (\text{Var } x) = \text{TComp } f T$ "
  and closed: "is_TComp_var_instance_closed  $\Gamma M$ "
  shows " $\exists g U. \text{Fun } g U \in M \wedge \Gamma (\text{Fun } g U) = \Gamma (\text{Var } x) \wedge (\forall u \in \text{set } U. \text{is\_Var } u) \wedge \text{distinct } U$ "
⟨proof⟩

lemma is_TComp_var_instance_closedD':
  assumes " $\exists y \in \text{fv\_set } M. \Gamma (\text{Var } x) = \Gamma (\text{Var } y)$ " " $\text{TComp } f T \sqsubseteq \Gamma (\text{Var } x)$ "
  and closed: "is_TComp_var_instance_closed  $\Gamma M$ "
  and wf: "wf_trms M"
  shows " $\exists g U. \text{Fun } g U \in M \wedge \Gamma (\text{Fun } g U) = \text{TComp } f T \wedge (\forall u \in \text{set } U. \text{is\_Var } u) \wedge \text{distinct } U$ "
⟨proof⟩

lemma TComp_var_instance_wt_subst_exists:
  assumes gT: " $\Gamma (\text{Fun } g T) = \text{TComp } g (\text{map } \Gamma U)$ " "wf_trm (Fun g T)"
  and U: " $\forall u \in \text{set } U. \exists y. u = \text{Var } y$ " "distinct U"
  shows " $\exists \vartheta. \text{wt\_subst } \vartheta \wedge \text{wf\_trms } (\text{subst\_range } \vartheta) \wedge \text{Fun } g T = \text{Fun } g U \cdot \vartheta$ "
⟨proof⟩

lemma TComp_var_instance_closed_has_Var:
  assumes closed: "is_TComp_var_instance_closed  $\Gamma M$ "
  and wf_M: "wf_trms M"
  and wf_ $\delta$ x: "wf_trm ( $\delta$  x)"
  and y_ex: " $\exists y \in \text{fv\_set } M. \Gamma (\text{Var } x) = \Gamma (\text{Var } y)$ "
  and t: "t  $\sqsubseteq$   $\delta$  x"
  and  $\delta$ _wt: "wt_subst  $\delta$ "
  shows " $\exists y \in \text{fv\_set } M. \Gamma (\text{Var } y) = \Gamma t$ "
⟨proof⟩

lemma TComp_var_instance_closed_has_Fun:
  assumes closed: "is_TComp_var_instance_closed  $\Gamma M$ "
  and wf_M: "wf_trms M"
  and wf_ $\delta$ x: "wf_trm ( $\delta$  x)"
  and y_ex: " $\exists y \in \text{fv\_set } M. \Gamma (\text{Var } x) = \Gamma (\text{Var } y)$ "
  and t: "t  $\sqsubseteq$   $\delta$  x"
  and  $\delta$ _wt: "wt_subst  $\delta$ "
  and t_ $\Gamma$ : " $\Gamma t = \text{TComp } g T$ "
  and t_fun: "is_Fun t"
  shows " $\exists m \in M. \exists \vartheta. \text{wt\_subst } \vartheta \wedge \text{wf\_trms } (\text{subst\_range } \vartheta) \wedge t = m \cdot \vartheta \wedge \text{is\_Fun } m$ "

```

<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} *M*) *M*"
and *M_wf*: "wf_{trms} *M*"
and *t*: "*t* \sqsubseteq_{set} *M*"
and *s*: "*s* \sqsubseteq *t* · δ "
and δ : "wt_{subst} δ " "wf_{trms} (subst_range δ)"
shows " $\exists m \in M. \exists \vartheta. wt_{subst} \vartheta \wedge wf_{trms} (subst_range \vartheta) \wedge s = m \cdot \vartheta$ "
<proof>

context

begin

private lemma *SMP_D_aux1*:

assumes "*t* \in *SMP M*"
and *closed*: "has_all_wt_instances_of Γ (subterms_{set} *M*) *M*"
" is_TComp_var_instance_closed Γ *M*"
and *wf_M*: "wf_{trms} *M*"
shows " $\forall x \in fv\ t. \exists y \in fv_{set}\ M. \Gamma (Var\ y) = \Gamma (Var\ x)$ "

<proof> **lemma** *SMP_D_aux2*:

fixes *t*: "('fun, ('fun, 'atom) term \times nat) term"
assumes *t_SMP*: "*t* \in *SMP M*"
and *t_Var*: " $\exists x. t = Var\ x$ "
and *M_SMP_repr*: "finite_SMP_representation arity Ana Γ *M*"
shows " $\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t = m \cdot \delta$ "

<proof> **lemma** *SMP_D_aux3*:

assumes *hyps*: "*t'* \sqsubseteq *t*" **and** *wf_t*: "wf_{trm} *t*" **and** *prems*: "is_Fun *t'*"
and *IH*:
" $(\exists f. t = Fun\ f\ []) \wedge (\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t = m \cdot \delta) \vee$
 $(\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t = m \cdot \delta \wedge is_Fun\ m)$ "
and *M_SMP_repr*: "finite_SMP_representation arity Ana Γ *M*"
shows " $(\exists f. t' = Fun\ f\ []) \wedge (\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t' = m \cdot \delta) \vee$
 $(\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t' = m \cdot \delta \wedge is_Fun\ m)$ "

<proof>

lemma *SMP_D*:

assumes "*t* \in *SMP M*" "is_Fun *t*"
and *M_SMP_repr*: "finite_SMP_representation arity Ana Γ *M*"
shows " $(\exists f. t = Fun\ f\ []) \wedge (\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t = m \cdot \delta) \vee$
 $(\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t = m \cdot \delta \wedge is_Fun\ m)$ "

<proof>

lemma *SMP_D'*:

fixes *M*
defines " $\delta \equiv var_rename (max_var_set (fv_{set}\ M))$ "
assumes *M_SMP_repr*: "finite_SMP_representation arity Ana Γ *M*"
and *s*: "*s* \in *SMP M*" "is_Fun *s*" " $\nexists f. s = Fun\ f\ []$ "
and *t*: "*t* \in *SMP M*" "is_Fun *t*" " $\nexists f. t = Fun\ f\ []$ "
obtains $\sigma\ s0\ \vartheta\ t0$
where "wt_{subst} σ " "wf_{trms} (subst_range σ)" "*s0* \in *M*" "is_Fun *s0*" "*s* = *s0* · σ " " $\Gamma\ s = \Gamma\ s0$ "
and "wt_{subst} ϑ " "wf_{trms} (subst_range ϑ)" "*t0* \in *M*" "is_Fun *t0*" "*t* = *t0* · δ · ϑ " " $\Gamma\ t = \Gamma\ t0$ "

<proof>

lemma *SMP_D''*:

fixes *t*: "('fun, ('fun, 'atom) term \times nat) term"
assumes *t_SMP*: "*t* \in *SMP M*"
and *M_SMP_repr*: "finite_SMP_representation arity Ana Γ *M*"
shows " $\exists m \in M. \exists \delta. wt_{subst} \delta \wedge wf_{trms} (subst_range \delta) \wedge t = m \cdot \delta$ "

<proof>

end

lemma *tfr_set_if_comp_tfr_set*:

```

  assumes "comp_tfr_set arity Ana  $\Gamma$  M"
  shows "tfr_set M"
<proof>

lemma tfr_set_if_comp_tfr_set':
  assumes "let N = SMP0 Ana  $\Gamma$  M in set M  $\subseteq$  set N  $\wedge$  comp_tfr_set arity Ana  $\Gamma$  (set N)"
  shows "tfr_set (set M)"
<proof>

lemma tfr_stp_is_comp_tfr_stp: "tfr_stp a = comp_tfr_stp  $\Gamma$  a"
<proof>

lemma tfr_st_if_comp_tfr_st:
  assumes "comp_tfr_st arity Ana  $\Gamma$  M S"
  shows "tfr_st S"
<proof>

lemma tfr_st_if_comp_tfr_st':
  assumes "comp_tfr_st arity Ana  $\Gamma$  (set (SMP0 Ana  $\Gamma$  (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  $\times$  nat) term set"
  assumes f_def: "f  $\equiv$   $\lambda$ M. {t  $\cdot$   $\delta$  | t  $\delta$ . t  $\in$  M  $\wedge$  wt_subst  $\delta$   $\wedge$  wf_trms (subst_range  $\delta$ )  $\wedge$  fv (t  $\cdot$   $\delta$ ) = {}}"
  and g_def: "g  $\equiv$   $\lambda$ M. {t  $\in$  M. fv t = {}}"
  shows "f (SMP M) = g (SMP M)"
<proof> lemma ground_SMP_disjointI_aux2:
  fixes M::('fun, ('fun, 'atom) term  $\times$  nat) term set"
  assumes f_def: "f  $\equiv$   $\lambda$ M. {t  $\cdot$   $\delta$  | t  $\delta$ . t  $\in$  M  $\wedge$  wt_subst  $\delta$   $\wedge$  wf_trms (subst_range  $\delta$ )  $\wedge$  fv (t  $\cdot$   $\delta$ ) = {}}"
  and M_SMP_repr: "finite_SMP_representation arity Ana  $\Gamma$  M"
  shows "f M = f (SMP M)"
<proof> lemma ground_SMP_disjointI_aux3:
  fixes A B C::('fun, ('fun, 'atom) term  $\times$  nat) term set"
  defines "P  $\equiv$   $\lambda$ t s.  $\exists$  $\delta$ . wt_subst  $\delta$   $\wedge$  wf_trms (subst_range  $\delta$ )  $\wedge$  Unifier  $\delta$  t s"
  assumes f_def: "f  $\equiv$   $\lambda$ M. {t  $\cdot$   $\delta$  | t  $\delta$ . t  $\in$  M  $\wedge$  wt_subst  $\delta$   $\wedge$  wf_trms (subst_range  $\delta$ )  $\wedge$  fv (t  $\cdot$   $\delta$ ) = {}}"
  and Q_def: "Q  $\equiv$   $\lambda$ t. intruder_synth' public arity {} t"
  and R_def: "R  $\equiv$   $\lambda$ t.  $\exists$ u  $\in$  C. is_wt_instance_of_cond  $\Gamma$  t u"
  and AB: "wf_trms A" "wf_trms B" "fv_set A  $\cap$  fv_set B = {}"
  and C: "wf_trms C"
  and ABC: " $\forall$ t  $\in$  A.  $\forall$ s  $\in$  B. P t s  $\longrightarrow$  Q t  $\vee$  R t"
  shows "f A  $\cap$  f B  $\subseteq$  f C  $\cup$  {m. {}  $\vdash_c$  m}"
<proof>

lemma ground_SMP_disjointI:
  fixes A B::('fun, ('fun, 'atom) term  $\times$  nat) term set" and C
  defines "f  $\equiv$   $\lambda$ M. {t  $\cdot$   $\delta$  | t  $\delta$ . t  $\in$  M  $\wedge$  wt_subst  $\delta$   $\wedge$  wf_trms (subst_range  $\delta$ )  $\wedge$  fv (t  $\cdot$   $\delta$ ) = {}}"
  and "g  $\equiv$   $\lambda$ M. {t  $\in$  M. fv t = {}}"
  and "Q  $\equiv$   $\lambda$ t. intruder_synth' public arity {} t"
  and "R  $\equiv$   $\lambda$ t.  $\exists$ u  $\in$  C. is_wt_instance_of_cond  $\Gamma$  t u"
  assumes AB_fv_disj: "fv_set A  $\cap$  fv_set B = {}"
  and A_SMP_repr: "finite_SMP_representation arity Ana  $\Gamma$  A"
  and B_SMP_repr: "finite_SMP_representation arity Ana  $\Gamma$  B"
  and C_wf: "wf_trms C"
  and ABC: " $\forall$ t  $\in$  A.  $\forall$ s  $\in$  B.  $\Gamma$  t =  $\Gamma$  s  $\wedge$  mgu t s  $\neq$  None  $\longrightarrow$  Q t  $\vee$  R t"
  shows "g (SMP A)  $\cap$  g (SMP B)  $\subseteq$  f C  $\cup$  {m. {}  $\vdash_c$  m}"

```

<proof>

end

end

end

3.4 The Typing Result

```
theory Typing_Result
imports Typed_Model
begin
```

3.4.1 Locale Setup

```
locale typing_result = typed_model arity public Ana  $\Gamma$ 
  for arity::"'fun  $\Rightarrow$  nat"
    and public::"'fun  $\Rightarrow$  bool"
    and Ana::"'(fun,'var) term  $\Rightarrow$  ((fun,'var) term list  $\times$  (fun,'var) term list)"
    and  $\Gamma$ ::"'(fun,'var) term  $\Rightarrow$  (fun,'atom::finite) term_type"
  +
  assumes infinite_typed_consts: " $\bigwedge a. \text{infinite } \{c. \Gamma (\text{Fun } c []) = \text{TAtom } a \wedge \text{public } c\}$ "
    and no_private_funs[simp]: " $\bigwedge f. \text{arity } f > 0 \implies \text{public } f$ "
begin
```

Minor Lemmata

```
lemma fun_type_inv': assumes " $\Gamma t = \text{TComp } f T$ " shows " $\text{arity } f > 0$ " "public  $f$ "
<proof>
```

```
lemma infinite_public_consts[simp]: " $\text{infinite } \{c. \text{public } c \wedge \text{arity } c = 0\}$ "
<proof>
```

```
lemma infinite_fun_syms[simp]:
  " $\text{infinite } \{c. \text{public } c \wedge \text{arity } c > 0\} \implies \text{infinite } \Sigma_f$ "
  " $\text{infinite } \mathcal{C}$ " " $\text{infinite } \mathcal{C}_{\text{pub}}$ " " $\text{infinite } (\text{UNIV}::'\text{fun set})$ "
<proof>
```

```
lemma id_univ_proper_subset[simp]: " $\Sigma_f \subset \text{UNIV}$ " " $(\exists f. \text{arity } f > 0) \implies \mathcal{C} \subset \text{UNIV}$ "
<proof>
```

```
lemma exists_fun_notin_funs_term: " $\exists f::'\text{fun}. f \notin \text{funs\_term } t$ "
<proof>
```

```
lemma exists_fun_notin_funs_terms:
  assumes " $\text{finite } M$ " shows " $\exists f::'\text{fun}. f \notin \bigcup (\text{funs\_term } ` M)$ "
<proof>
```

```
lemma exists_notin_funs_st: " $\exists f. f \notin \text{funs}_{\text{st}} (S::(\text{fun,'var}) \text{strand})$ "
<proof>
```

```
lemma infinite_typed_consts': " $\text{infinite } \{c. \Gamma (\text{Fun } c []) = \text{TAtom } a \wedge \text{public } c \wedge \text{arity } c = 0\}$ "
<proof>
```

```
lemma atypes_inhabited: " $\exists c. \Gamma (\text{Fun } c []) = \text{TAtom } a \wedge \text{wf}_{\text{trm}} (\text{Fun } c []) \wedge \text{public } c \wedge \text{arity } c = 0$ "
<proof>
```

```
lemma atype_ground_term_ex: " $\exists t. \text{fv } t = \{\} \wedge \Gamma t = \text{TAtom } a \wedge \text{wf}_{\text{trm}} t$ "
<proof>
```

```
lemma type_ground_inhabited: " $\exists t'. \text{fv } t' = \{\} \wedge \Gamma t = \Gamma t'$ "
```

<proof>

lemma type_wfttype_inhabited:
 assumes " $\bigwedge f T. \text{Fun } f T \sqsubseteq \tau \implies 0 < \text{arity } f$ " "wf_{trm} τ "
 shows " $\exists t. \Gamma t = \tau \wedge \text{wf}_{trm} t$ "
<proof>

lemma type_pgwt_inhabited: "wf_{trm} $t \implies \exists t'. \Gamma t = \Gamma t' \wedge \text{public_ground_wf_term } t'$ "
<proof>

end

3.4.2 The Typing Result for the Composition-Only Intruder

context typing_result
 begin

Well-typedness and Type-Flaw Resistance Preservation

context
 begin

private lemma LI_preserves_tfr_stp_all_single:
 assumes " $(S, \vartheta) \rightsquigarrow (S', \vartheta')$ " "wf_{constr} $S \vartheta$ " "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 " $(S, \vartheta) \rightsquigarrow (S', \vartheta')$ " "wf_{constr} $S \vartheta$ " "wt_{subst} ϑ "
 "tfr_{set} (trms_{st} S)" "wf_{trms} (trms_{st} S)" "list_all tfr_{stp} S "
 and "trms_{st} $S \subseteq \text{SMP } M$ "
 shows "trms_{st} $S' \subseteq \text{SMP } M$ "

<proof> lemma LI_preserves_tfr_single:
 assumes " $(S, \vartheta) \rightsquigarrow (S', \vartheta')$ " "wf_{constr} $S \vartheta$ " "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} $S')$ \wedge wf_{trms} (trms_{st} $S')$ "

<proof> lemma LI_preserves_welltypedness_single:
 assumes " $(S, \vartheta) \rightsquigarrow (S', \vartheta')$ " "wf_{constr} $S \vartheta$ " "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} $\vartheta' \wedge$ wf_{trms} (subst_range $\vartheta')$ "
<proof>

lemma LI_preserves_welltypedness:
 assumes " $(S, \vartheta) \rightsquigarrow^* (S', \vartheta')$ " "wf_{constr} $S \vartheta$ " "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 $\vartheta')$ " (is "?B ϑ' ")
<proof>

lemma LI_preserves_tfr:
 assumes " $(S, \vartheta) \rightsquigarrow^* (S', \vartheta')$ " "wf_{constr} $S \vartheta$ " "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>

lemma LI_preproc_preserves_tfr:
 assumes "tfr_{st} S "
 shows "tfr_{st} (LI_preproc S)"
<proof>
 end

Simple Constraints are Well-typed Satisfiable

Proving the existence of a well-typed interpretation

```
context
begin
```

```
lemma wt_interpretation_exists:
  obtains  $\mathcal{I}::('fun, 'var) \text{subst}$ 
  where "interpretationsubst  $\mathcal{I}$ " "wtsubst  $\mathcal{I}$ " "subst_range  $\mathcal{I} \subseteq \text{public\_ground\_wf\_terms}$ "
<proof>
```

```
lemma wt_grounding_subst_exists:
  " $\exists \vartheta. \text{wt}_{subst} \vartheta \wedge \text{wf}_{trms} (\text{subst\_range } \vartheta) \wedge \text{fv} (t \cdot \vartheta) = \{\}$ "
<proof> fun fresh_pgwt::"fun set  $\Rightarrow$  ('fun, 'atom) term_type  $\Rightarrow$  ('fun, 'var) term" where
  "fresh_pgwt S (TAtom a) =
    Fun (SOME c. c  $\notin$  S  $\wedge$   $\Gamma$  (Fun c []) = TAtom a  $\wedge$  public c) []"
| "fresh_pgwt S (TComp f T) = Fun f (map (fresh_pgwt S) T)"
```

```
private lemma fresh_pgwt_same_type:
  assumes "finite S" "wftrm t"
  shows " $\Gamma$  (fresh_pgwt S ( $\Gamma$  t)) =  $\Gamma$  t"
```

```
<proof> lemma fresh_pgwt_empty_synth:
```

```
  assumes "finite S" "wftrm t"
  shows " $\{\} \vdash_c \text{fresh\_pgwt } S (\Gamma t)$ "
```

```
<proof> lemma fresh_pgwt_has_fresh_const:
```

```
  assumes "finite S" "wftrm t"
  obtains c where "Fun c []  $\sqsubseteq$  fresh_pgwt S ( $\Gamma$  t)" "c  $\notin$  S"
```

```
<proof> lemma fresh_pgwt_subterm_fresh:
```

```
  assumes "finite S" "wftrm t" "wftrm s" "funs_term s  $\subseteq$  S"
  shows "s  $\notin$  subterms (fresh_pgwt S ( $\Gamma$  t))"
```

```
<proof> lemma wt_fresh_pgwt_term_exists:
```

```
  assumes "finite T" "wftrm s" "wftrms T"
  obtains t where " $\Gamma$  t =  $\Gamma$  s" " $\{\} \vdash_c t$ " " $\forall s \in T. \forall u \in \text{subterms } s. u \notin \text{subterms } t$ "
```

```
<proof>
```

```
lemma wt_bij_finite_subst_exists:
```

```
  assumes "finite (S::'var set)" "finite (T::('fun, 'var) terms)" "wftrms T"
  shows " $\exists \sigma::('fun, 'var) \text{subst}$ .
```

```
    subst_domain  $\sigma = S$ 
     $\wedge$  bij_betw  $\sigma$  (subst_domain  $\sigma$ ) (subst_range  $\sigma$ )
     $\wedge$  subtermsset (subst_range  $\sigma$ )  $\subseteq$  {t.  $\{\} \vdash_c t$ } - T
     $\wedge$  ( $\forall s \in \text{subst\_range } \sigma. \forall u \in \text{subst\_range } \sigma. (\exists v. v \sqsubseteq s \wedge v \sqsubseteq u) \longrightarrow s = u$ )
     $\wedge$  wtsubst  $\sigma$ 
     $\wedge$  wftrms (subst_range  $\sigma$ )"
```

```
<proof> lemma wt_bij_finite_tatom_subst_exists_single:
```

```
  assumes "finite (S::'var set)" "finite (T::('fun, 'var) terms)"
  and " $\wedge x. x \in S \implies \Gamma$  (Var x) = TAtom a"
```

```
  shows " $\exists \sigma::('fun, 'var) \text{subst}$ . subst_domain  $\sigma = S$ 
```

```
     $\wedge$  bij_betw  $\sigma$  (subst_domain  $\sigma$ ) (subst_range  $\sigma$ )
     $\wedge$  subst_range  $\sigma \subseteq ((\lambda c. \text{Fun } c []) \setminus \{c. \Gamma$  (Fun c []) = TAtom a  $\wedge$ 
      public c  $\wedge$  arity c = 0}) - T
     $\wedge$  wtsubst  $\sigma$ 
     $\wedge$  wftrms (subst_range  $\sigma$ )"
```

```
<proof>
```

```
lemma wt_bij_finite_tatom_subst_exists:
```

```
  assumes "finite (S::'var set)" "finite (T::('fun, 'var) terms)"
```

```
  and " $\wedge x. x \in S \implies \exists a. \Gamma$  (Var x) = TAtom a"
```

```
  shows " $\exists \sigma::('fun, 'var) \text{subst}$ . subst_domain  $\sigma = S$ 
```

```
     $\wedge$  bij_betw  $\sigma$  (subst_domain  $\sigma$ ) (subst_range  $\sigma$ )
     $\wedge$  subst_range  $\sigma \subseteq ((\lambda c. \text{Fun } c []) \setminus \mathcal{C}_{pub}) - T$ 
     $\wedge$  wtsubst  $\sigma$ 
     $\wedge$  wftrms (subst_range  $\sigma$ )"
```

<proof>

theorem *wt_sat_if_simple:*

```

assumes "simple S" "wf_constr S  $\vartheta$ " "wt_subst  $\vartheta$ " "wf_trms (subst_range  $\vartheta$ )" "wf_trms (trmsst S)"
and  $\mathcal{I}'$ : " $\forall X F$ . Inequality  $X F \in \text{set } S \longrightarrow \text{ineq\_model } \mathcal{I}' X F$ "
      "ground (subst_range  $\mathcal{I}'$ )"
      "subst_domain  $\mathcal{I}' = \{x \in \text{vars}_{st} S. \exists X F$ . Inequality  $X F \in \text{set } S \wedge x \in \text{fv}_{pairs} F - \text{set } X\}$ "
and tfr_stp_all: "list_all tfr_stp S"
shows " $\exists \mathcal{I}$ . interpretationsubst  $\mathcal{I} \wedge (\mathcal{I} \models_c \langle S, \vartheta \rangle) \wedge \text{wt}_{subst} \mathcal{I} \wedge \text{wf}_{trms} (\text{subst\_range } \mathcal{I})$ "

```

<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 "interpretationsubst  $\mathcal{I}$ "
and " $\mathcal{I} \models_c \langle S, \vartheta \rangle$ "
and "wf_constr S  $\vartheta$ "
and "wt_subst  $\vartheta$ "
and "tfrst S"
and "wf_trms (trmsst S)"
and "wf_trms (subst_range  $\vartheta$ )"
obtains  $\mathcal{I}_\tau$  where "interpretationsubst  $\mathcal{I}_\tau$ "
and " $\mathcal{I}_\tau \models_c \langle S, \vartheta \rangle$ "
and "wt_subst  $\mathcal{I}_\tau$ "
and "wf_trms (subst_range  $\mathcal{I}_\tau$ )"

```

<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 " $\neg (\exists \mathcal{I}_\tau$ . interpretationsubst  $\mathcal{I}_\tau \wedge (\mathcal{I}_\tau \models_c \langle S, \vartheta \rangle) \wedge \text{wt}_{subst} \mathcal{I}_\tau)$ "
and "wf_constr S  $\vartheta$ " "wt_subst  $\vartheta$ " "tfrst S"
and "wf_trms (trmsst S)" "wf_trms (subst_range  $\vartheta$ )"
shows " $\neg (\exists \mathcal{I}$ . interpretationsubst  $\mathcal{I} \wedge (\mathcal{I} \models_c \langle S, \vartheta \rangle))$ "

```

<proof>

end

3.4.3 Lifting the Composition-Only Typing Result to the Full Intruder Model

context *typing_result*

begin

Analysis Invariance

definition (in *typed_model*) *Ana_invar_subst* where

```

"Ana_invar_subst  $\mathcal{M} \equiv$ 
( $\forall f T K M \delta$ . Fun  $f T \in (\text{subterms}_{set} \mathcal{M}) \longrightarrow$ 
  Ana (Fun  $f T) = (K, M) \longrightarrow \text{Ana} (\text{Fun } f T \cdot \delta) = (K \cdot \text{list } \delta, M \cdot \text{list } \delta))$ "

```

lemma (in *typed_model*) *Ana_invar_subst_subset:*

```

assumes "Ana_invar_subst M" "N  $\subseteq$  M"
shows "Ana_invar_subst N"

```

<proof>

lemma (in *typed_model*) *Ana_invar_substD:*

```

assumes "Ana_invar_subst M"
and "Fun  $f T \in \text{subterms}_{set} \mathcal{M}$ " "Ana (Fun  $f T) = (K, M)$ "
shows "Ana (Fun  $f T \cdot \mathcal{I}) = (K \cdot \text{list } \mathcal{I}, M \cdot \text{list } \mathcal{I})$ "

```

<proof>

end

Preliminary Definitions

Strands extended with "decomposition steps"

```
datatype (funsestp: 'a, varsestp: 'b) extstrand_step =
  Step "'a,'b) strand_step"
| Decomp "'a,'b) term"
```

```
context typing_result
begin
```

```
context
```

```
begin
```

```
private fun trmsestp where
```

```
  "trmsestp (Step x) = trmsstp x"
| "trmsestp (Decomp t) = {t}"
```

```
private abbreviation trmsest where "trmsest S ≡ ⋃ (trmsestp ` 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, send⟨K⟩st, receive⟨T⟩st])"
```

```
private fun tost where
```

```
  "tost [] = []"
| "tost (Step x#S) = x#(tost S)"
| "tost (Decomp t#S) = (decomp t)@(tost S)"
```

```
private fun toest where
```

```
  "toest [] = []"
| "toest (x#S) = Step x#toest S"
```

```
private abbreviation "ikest A ≡ ikst (tost A)"
```

```
private abbreviation "wfest V A ≡ wfst V (tost A)"
```

```
private abbreviation "assignment_rhsest A ≡ assignment_rhsst (tost A)"
```

```
private abbreviation "varsest A ≡ varsst (tost A)"
```

```
private abbreviation "wfrestrictedvarsest A ≡ wfrestrictedvarsst (tost A)"
```

```
private abbreviation "bvarsest A ≡ bvarsst (tost A)"
```

```
private abbreviation "fvest A ≡ fvst (tost A)"
```

```
private abbreviation "funsest A ≡ funsst (tost A)"
```

```
private definition wfsts'::('fun,'var) strands ⇒ ('fun,'var) extstrand ⇒ bool where
```

```
  "wfsts' S A ≡ (∀S ∈ S. wfst (wfrestrictedvarsest A) (dualst S)) ∧
    (∀S ∈ S. ∀S' ∈ S. fvst S ∩ bvarsst S' = {}) ∧
    (∀S ∈ S. fvst S ∩ bvarsest A = {}) ∧
    (∀S ∈ S. fvst (tost A) ∩ bvarsst S = {})"
```

```
private definition wfsts::('fun,'var) strands ⇒ bool where
```

```
  "wfsts S ≡ (∀S ∈ S. wfst {} (dualst S)) ∧ (∀S ∈ S. ∀S' ∈ S. fvst S ∩ bvarsst 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 ∈ subtermsset (ikest A ∪ assignment_rhsest 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⟨ts⟩st)) ·estp ϑ = Step (send⟨ts ·list ϑ⟩st)"
  "(Step (receive⟨ts⟩st)) ·estp ϑ = Step (receive⟨ts ·list 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 varsestp_subst_apply_simps[simp]:
  "varsestp ((Step (send⟨ts⟩st)) ·estp ϑ) = fvset (set ts ·set ϑ)"
  "varsestp ((Step (receive⟨ts⟩st)) ·estp ϑ) = fvset (set ts ·set ϑ)"
  "varsestp ((Step (⟨a: t ≐ t'⟩st)) ·estp ϑ) = fv (t · ϑ) ∪ fv (t' · ϑ)"
  "varsestp ((Step (∀X(∀≠: F)st)) ·estp ϑ) = set X ∪ fvpairs (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 updatest:: "('fun, 'var) strands ⇒ ('fun, 'var) strand ⇒ ('fun, 'var) strands"
where
  "updatest S S ≡ (case S of Nil ⇒ S - {S} | Cons _ S' ⇒ insert S' (S - {S}))"

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

for M and N and I where
  Nil: "[] ∈ decompest M N I"
| Decompose: "[D ∈ decompest M N I; Fun f T ∈ subtermsset (M ∪ N);
  Ana (Fun f T) = (K, M); M ≠ [];
  (M ∪ ikest D) ·set I ⊢c Fun f T · I;
  ∧k. k ∈ set K ⇒ (M ∪ ikest D) ·set I ⊢c k · I]
  ⇒ D@[Decompose (Fun f T)] ∈ decompest M N I"

private fun decomprmest:: "('fun, 'var) extstrand ⇒ ('fun, 'var) extstrand" where
  "decomprmest [] = []"
| "decomprmest (Decompose t#S) = decomprmest S"
| "decomprmest (Step x#S) = Step x#(decomprmest S)"

private inductive semest_d:: "('fun, 'var) terms ⇒ ('fun, 'var) subst ⇒ ('fun, 'var) extstrand ⇒ bool"
where
  Nil[simp]: "semest_d M0 I []"
| Send: "semest_d M0 I S ⇒ ∀t ∈ set ts. (ikest S ∪ M0) ·set I ⊢ t · I
  ⇒ semest_d M0 I (S@[Step (send⟨ts⟩st)])"
| Receive: "semest_d M0 I S ⇒ semest_d M0 I (S@[Step (receive⟨t⟩st)])"
| Equality: "semest_d M0 I S ⇒ t · I = t' · I ⇒ semest_d M0 I (S@[Step (⟨a: t ≐ t'⟩st)])"
| Inequality: "semest_d M0 I S
  ⇒ ineq_model I X F
  ⇒ semest_d M0 I (S@[Step (∀X(∀≠: F)st)])"
| Decompose: "semest_d M0 I S ⇒ (ikest S ∪ M0) ·set I ⊢ t · I ⇒ Ana t = (K, M)
  ⇒ (∧k. k ∈ set K ⇒ (ikest S ∪ M0) ·set I ⊢ k · I) ⇒ semest_d M0 I (S@[Decompose t])"

private inductive semest_c:: "('fun, 'var) terms ⇒ ('fun, 'var) subst ⇒ ('fun, 'var) extstrand ⇒ bool"
where
  Nil[simp]: "semest_c M0 I []"
| Send: "semest_c M0 I S ⇒ ∀t ∈ set ts. (ikest S ∪ M0) ·set I ⊢c t · I
  ⇒ semest_c M0 I (S@[Step (send⟨ts⟩st)])"
| Receive: "semest_c M0 I S ⇒ semest_c M0 I (S@[Step (receive⟨t⟩st)])"
| Equality: "semest_c M0 I S ⇒ t · I = t' · I ⇒ semest_c M0 I (S@[Step (⟨a: t ≐ t'⟩st)])"
| Inequality: "semest_c M0 I S
  ⇒ ineq_model I X F
  ⇒ semest_c M0 I (S@[Step (∀X(∀≠: F)st)])"
| Decompose: "semest_c M0 I S ⇒ (ikest S ∪ M0) ·set I ⊢c t · I ⇒ Ana t = (K, M)
  ⇒ (∧k. k ∈ set K ⇒ (ikest S ∪ M0) ·set I ⊢c k · I) ⇒ semest_c M0 I (S@[Decompose t])"

```

Preliminary Lemmata

```

private lemma wfsts_wfsts':
  "wfsts S = wfsts' S []"

```

```

⟨proof⟩ lemma decomp_ik:
  assumes "Ana t = (K,M)"
  shows "ikst (decomp t) = set M"
⟨proof⟩ lemma decomp_assignment_rhs_empty:
  assumes "Ana t = (K,M)"
  shows "assignment_rhsst (decomp t) = {}"
⟨proof⟩ lemma decomp_tfrstp:
  "list_all tfrstp (decomp t)"
⟨proof⟩ lemma trmsest_ikI:
  "t ∈ ikest A ⇒ t ∈ subtermsset (trmsest A)"
⟨proof⟩ lemma trmsest_ik_assignment_rhsI:
  "t ∈ ikest A ∪ assignment_rhsest A ⇒ t ∈ subtermsset (trmsest A)"
⟨proof⟩ lemma trmsest_ik_subtermsI:
  assumes "t ∈ subtermsset (ikest A)"
  shows "t ∈ subtermsset (trmsest A)"
⟨proof⟩ lemma trmsestD:
  assumes "t ∈ trmsest A"
  shows "t ∈ trmsst (tost A)"
⟨proof⟩ lemma subst_apply_extstrand_nil[simp]:
  "[] ·est ∅ = []"
⟨proof⟩ lemma subst_apply_extstrand_singleton[simp]:
  "[Step (receive(ts)st)] ·est ∅ = [Step (Receive (ts ·list ∅))]"
  "[Step (send(ts)st)] ·est ∅ = [Step (Send (ts ·list ∅))]"
  "[Step (⟨a: t ≐ t'⟩st)] ·est ∅ = [Step (Equality a (t · ∅) (t' · ∅))]"
  "[Decomp t] ·est ∅ = [Decomp (t · ∅)]"
⟨proof⟩ lemma extstrand_subst_hom:
  "(S@S') ·est ∅ = (S ·est ∅)@(S' ·est ∅)" "(x#S) ·est ∅ = (x ·estp ∅)#(S ·est ∅)"
⟨proof⟩ lemma decomp_vars:
  "wfrestrictedvarsst (decomp t) = fv t" "varsst (decomp t) = fv t" "bvarsst (decomp t) = {}"
  "fvst (decomp t) = fv t"
⟨proof⟩ lemma bvarsest_cons: "bvarsest (x#X) = bvarsest [x] ∪ bvarsest X"
⟨proof⟩ lemma bvarsest_append: "bvarsest (A@B) = bvarsest A ∪ bvarsest B"
⟨proof⟩ lemma fvest_cons: "fvest (x#X) = fvest [x] ∪ fvest X"
⟨proof⟩ lemma fvest_append: "fvest (A@B) = fvest A ∪ fvest B"
⟨proof⟩ lemma bvars_decomp: "bvarsest (A@[Decomp t]) = bvarsest A" "bvarsest (Decomp t#A) = bvarsest A"
⟨proof⟩ lemma bvars_decomp_rm: "bvarsest (decomp_rmest A) = bvarsest A"
⟨proof⟩ lemma fv_decomp_rm: "fvest (decomp_rmest A) ⊆ fvest A"
⟨proof⟩ lemma ik_assignment_rhs_decomp_fv:
  assumes "t ∈ subtermsset (ikest A ∪ assignment_rhsest A)"
  shows "fvest (A@[Decomp t]) = fvest A"
⟨proof⟩ lemma wfrestrictedvarsest_decomp_rmest_subset:
  "wfrestrictedvarsest (decomp_rmest A) ⊆ wfrestrictedvarsest A"
⟨proof⟩ lemma wfrestrictedvarsest_eq_wfrestrictedvarsst:
  "wfrestrictedvarsest A = wfrestrictedvarsst (tost A)"
⟨proof⟩ lemma decomp_set_unfold:
  assumes "Ana t = (K, M)"
  shows "set (decomp t) = {send⟨[t]⟩st, send⟨K⟩st, receive⟨M⟩st}"
⟨proof⟩ lemma ikest_finite: "finite (ikest A)"
⟨proof⟩ lemma assignment_rhsest_finite: "finite (assignment_rhsest A)"
⟨proof⟩ lemma toest_append: "toest (A@B) = toest A@toest B"
⟨proof⟩ lemma tost_toest_inv: "tost (toest A) = A"
⟨proof⟩ lemma tost_append: "tost (A@B) = (tost A)@(tost B)"
⟨proof⟩ lemma tost_cons: "tost (a#B) = (tost [a])@(tost B)"
⟨proof⟩ lemma wfrestrictedvarsest_split:
  "wfrestrictedvarsest (x#S) = wfrestrictedvarsest [x] ∪ wfrestrictedvarsest S"
  "wfrestrictedvarsest (S@S') = wfrestrictedvarsest S ∪ wfrestrictedvarsest S'"
⟨proof⟩ lemma ikest_append: "ikest (A@B) = ikest A ∪ ikest B"
⟨proof⟩ lemma assignment_rhsest_append:
  "assignment_rhsest (A@B) = assignment_rhsest A ∪ assignment_rhsest B"
⟨proof⟩ lemma ikest_cons: "ikest (a#A) = ikest [a] ∪ ikest A"
⟨proof⟩ lemma ikest_append_subst:
  "ikest (A@B) ·est ∅ = ikest (A ·est ∅) ∪ ikest (B ·est ∅)"
  "ikest (A@B) ·set ∅ = (ikest A ·set ∅) ∪ (ikest B ·set ∅)"

```

```

⟨proof⟩ lemma assignment_rhs_est_append_subst:
  "assignment_rhs_est (A@B ·est ∅) = assignment_rhs_est (A ·est ∅) ∪ assignment_rhs_est (B ·est ∅)"
  "assignment_rhs_est (A@B) ·set ∅ = (assignment_rhs_est A ·set ∅) ∪ (assignment_rhs_est B ·set ∅)"
⟨proof⟩ lemma ik_est_cons_subst:
  "ik_est (a#A ·est ∅) = ik_est ([a ·estp ∅]) ∪ ik_est (A ·est ∅)"
  "ik_est (a#A) ·set ∅ = (ik_est [a] ·set ∅) ∪ (ik_est A ·set ∅)"
⟨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⟨ts⟩st)] = [Step (send⟨ts⟩st)]"
  "decomp_rm_est [Step (receive⟨ts⟩st)] = [Step (receive⟨ts⟩st)]"
  "decomp_rm_est [Decomp t] = []"
⟨proof⟩ lemma decomp_rm_est_ik_subset: "ik_est (decomp_rm_est S) ⊆ ik_est S"
⟨proof⟩ lemma decomp_rm_est_ik_subset: "D ∈ decomp_rm_est M N I ⇒ ik_est D ⊆ subtermsset (M ∪ N)"
⟨proof⟩ lemma decomp_rm_est_decomp_rm_est_empty: "D ∈ decomp_rm_est M N I ⇒ decomp_rm_est D = []"
⟨proof⟩ lemma decomp_rm_est_append:
  assumes "A ∈ decomp_rm_est S N I" "B ∈ decomp_rm_est S N I"
  shows "A@B ∈ decomp_rm_est S N I"
⟨proof⟩ lemma decomp_rm_est_subterms:
  assumes "A' ∈ decomp_rm_est M N I"
  shows "subtermsset (ik_est A') ⊆ subtermsset (M ∪ N)"
⟨proof⟩ lemma decomp_rm_est_assignment_rhs_empty:
  assumes "A' ∈ decomp_rm_est M N I"
  shows "assignment_rhs_est A' = {}"
⟨proof⟩ lemma decomp_rm_est_finite_ik_append:
  assumes "finite M" "M ⊆ decomp_rm_est A N I"
  shows "∃D ∈ decomp_rm_est A N 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 ∈ subtermsset (ik_est A ∪ assignment_rhs_est A) - (Var ` 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⟨ts⟩st)]" "well_analyzed [Step (receive⟨ts⟩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 ⇒ fvest (decomp_rm_est A) = fvest A"
⟨proof⟩ lemma sem_est_d_split_left: assumes "sem_est_d M0 I (A@A)" shows "sem_est_d M0 I A"
⟨proof⟩ lemma sem_est_d_eq_sem_st: "sem_est_d M0 I A = ⟦M0; to_st A⟧d' I"
⟨proof⟩ lemma sem_est_c_eq_sem_st: "sem_est_c M0 I A = ⟦M0; to_st A⟧c' I"
⟨proof⟩ lemma sem_est_c_decomp_rm_est_deduct_aux:
  assumes "sem_est_c M0 I A" "t ∈ ik_est A ·set I" "t ∉ ik_est (decomp_rm_est A) ·set I"
  shows "ik_est (decomp_rm_est A) ∪ M0 ·set I ⊢ t"
⟨proof⟩ lemma sem_est_c_decomp_rm_est_deduct:
  assumes "sem_est_c M0 I A" "ik_est A ∪ M0 ·set I ⊢c t"
  shows "ik_est (decomp_rm_est A) ∪ M0 ·set I ⊢ t"
⟨proof⟩ lemma sem_est_d_decomp_rm_est_if_sem_est_c: "sem_est_c M0 I A ⇒ sem_est_d M0 I (decomp_rm_est A)"
⟨proof⟩ lemma sem_est_c_decomp_rm_est_append:
  assumes "sem_est_c {} I A" "D ∈ decomp_rm_est (ik_est A) (assignment_rhs_est A) I"
  shows "sem_est_c {} I (A@D)"
⟨proof⟩ lemma decomp_rm_est_preserves_wf:
  assumes "D ∈ decomp_rm_est (ik_est A) (assignment_rhs_est A) I" "wfest V A"
  shows "wfest V (A@D)"
⟨proof⟩ lemma decomp_rm_est_preserves_model_c:
  assumes "D ∈ decomp_rm_est (ik_est A) (assignment_rhs_est A) I" "sem_est_c M0 I A"

```

`shows "semest_c M0 I (A@D)"`
`<proof> lemma decompest_exist_aux:`
`assumes "D ∈ decompest M N I" "M ∪ ikest D ⊢ t" "¬(M ∪ (ikest D) ⊢c t)"`
`obtains D' where`
`"D@D' ∈ decompest M N I" "M ∪ ikest (D@D') ⊢c t" "M ∪ ikest D ⊆ M ∪ ikest (D@D')"`
`<proof> lemma decompest_ik_max_exist:`
`assumes "finite A" "finite N"`
`shows "∃D ∈ decompest A N I. ∀D' ∈ decompest A N I. ikest D' ⊆ ikest D"`
`<proof> lemma decompest_exist:`
`assumes "finite A" "finite N"`
`shows "∃D ∈ decompest A N I. ∀t. A ⊢ t → A ∪ ikest D ⊢c t"`
`<proof> lemma decompest_exist_subst:`
`assumes "ikest A ·set I ⊢ t · I"`
`and "semest_c {} I A" "wfest {} A" "interpretationsubst I"`
`and "Ana_invar_subst (ikest A ∪ assignment_rhsest A)"`
`and "well_analyzed A"`
`shows "∃D ∈ decompest (ikest A) (assignment_rhsest A) I. ikest (A@D) ·set I ⊢c t · I"`
`<proof> lemma decompest_exist_subst_list:`
`assumes "∀t ∈ set ts. ikest A ·set I ⊢ t · I"`
`and "semest_c {} I A" "wfest {} A" "interpretationsubst I"`
`and "Ana_invar_subst (ikest A ∪ assignment_rhsest A)"`
`and "well_analyzed A"`
`shows "∃D ∈ decompest (ikest A) (assignment_rhsest A) I.`
`∀t ∈ set ts. ikest (A@D) ·set I ⊢c t · I"`
`(is "∃D ∈ ?A. ?B D ts")`
`<proof> lemma wfsts'_updatest_nil: assumes "wfsts' S A" shows "wfsts' (updatest S []) A"`
`<proof> lemma wfsts'_updatest_snd:`
`assumes "wfsts' S A" "send(ts)st#S ∈ S"`
`shows "wfsts' (updatest S (send(ts)st#S)) (A@[Step (receive(ts)st)])"`
`<proof> lemma wfsts'_updatest_rcv:`
`assumes "wfsts' S A" "receive(ts)st#S ∈ S"`
`shows "wfsts' (updatest S (receive(ts)st#S)) (A@[Step (send(ts)st)])"`
`<proof> lemma wfsts'_updatest_eq:`
`assumes "wfsts' S A" "<a: t ≐ t'>st#S ∈ S"`
`shows "wfsts' (updatest S (<a: t ≐ t'>st#S)) (A@[Step (<a: t ≐ t'>st)])"`
`<proof> lemma wfsts'_updatest_ineq:`
`assumes "wfsts' S A" "∀X<V≠: F>st#S ∈ S"`
`shows "wfsts' (updatest S (∀X<V≠: F>st#S)) (A@[Step (∀X<V≠: F>st)])"`
`<proof> lemma trmsst_updatest_eq:`
`assumes "x#S ∈ S"`
`shows "⋃(trmsst ` updatest S (x#S)) ∪ trmsstp x = ⋃(trmsst ` S)" (is "?A = ?B")`
`<proof> lemma trmsst_updatest_eq_snd:`
`assumes "send(ts)st#S ∈ S" "S' = updatest S (send(ts)st#S)" "A' = A@[Step (receive(ts)st)]"`
`shows "(⋃(trmsst ` S)) ∪ (trmsest A) = (⋃(trmsst ` S')) ∪ (trmsest A')"`
`<proof> lemma trmsst_updatest_eq_rcv:`
`assumes "receive(ts)st#S ∈ S" "S' = updatest S (receive(ts)st#S)" "A' = A@[Step (send(ts)st)]"`
`shows "(⋃(trmsst ` S)) ∪ (trmsest A) = (⋃(trmsst ` S')) ∪ (trmsest A')"`
`<proof> lemma trmsst_updatest_eq_eq:`
`assumes "<a: t ≐ t'>st#S ∈ S" "S' = updatest S (<a: t ≐ t'>st#S)" "A' = A@[Step (<a: t ≐ t'>st)]"`
`shows "(⋃(trmsst ` S)) ∪ (trmsest A) = (⋃(trmsst ` S')) ∪ (trmsest A')"`
`<proof> lemma trmsst_updatest_eq_ineq:`
`assumes "∀X<V≠: F>st#S ∈ S" "S' = updatest S (∀X<V≠: F>st#S)" "A' = A@[Step (∀X<V≠: F>st)]"`
`shows "(⋃(trmsst ` S)) ∪ (trmsest A) = (⋃(trmsst ` S')) ∪ (trmsest A')"`
`<proof> lemma ikst_updatest_subset:`
`assumes "x#S ∈ S"`
`shows "⋃(ikst ` dualst ` (updatest S (x#S))) ⊆ ⋃(ikst ` dualst ` S)" (is ?A)`
`"⋃(assignment_rhsst ` (updatest S (x#S))) ⊆ ⋃(assignment_rhsst ` S)" (is ?B)`
`<proof> lemma ikst_updatest_subset_snd:`
`assumes "send(ts)st#S ∈ S"`
`"S' = updatest S (send(ts)st#S)"`
`"A' = A@[Step (receive(ts)st)]"`
`shows "(⋃(ikst ` dualst ` S')) ∪ (ikest A) ⊆`
`(⋃(ikst ` dualst ` S)) ∪ (ikest A)" (is ?A)`

```

"( $\bigcup$  (assignment_rhsst ` S'))  $\cup$  (assignment_rhsest A')  $\subseteq$ 
( $\bigcup$  (assignment_rhsst ` S))  $\cup$  (assignment_rhsest A)" (is ?B)
<proof> lemma ikst_updatest_subset_rcv:
  assumes "receive⟨t⟩st#S  $\in$  S"
    "S' = updatest S (receive⟨t⟩st#S)"
    "A' = A@[Step (send⟨t⟩st)]"
  shows " $\bigcup$  (ikst ` dualst ` S')  $\cup$  (ikest A')  $\subseteq$ 
( $\bigcup$  (ikst ` dualst ` S))  $\cup$  (ikest A)" (is ?A)
    " $\bigcup$  (assignment_rhsst ` S')  $\cup$  (assignment_rhsest A')  $\subseteq$ 
( $\bigcup$  (assignment_rhsst ` S))  $\cup$  (assignment_rhsest A)" (is ?B)
<proof> lemma ikst_updatest_subset_eq:
  assumes "(a: t  $\doteq$  t')st#S  $\in$  S"
    "S' = updatest S ((a: t  $\doteq$  t')st#S)"
    "A' = A@[Step ((a: t  $\doteq$  t')st)]"
  shows " $\bigcup$  (ikst ` dualst ` S')  $\cup$  (ikest A')  $\subseteq$ 
( $\bigcup$  (ikst ` dualst ` S))  $\cup$  (ikest A)" (is ?A)
    " $\bigcup$  (assignment_rhsst ` S')  $\cup$  (assignment_rhsest A')  $\subseteq$ 
( $\bigcup$  (assignment_rhsst ` S))  $\cup$  (assignment_rhsest A)" (is ?B)
<proof> lemma ikst_updatest_subset_ineq:
  assumes " $\forall X(\forall \neq: F)$ st#S  $\in$  S"
    "S' = updatest S ( $\forall X(\forall \neq: F)$ st#S)"
    "A' = A@[Step ( $\forall X(\forall \neq: F)$ st)]"
  shows " $\bigcup$  (ikst ` dualst ` S')  $\cup$  (ikest A')  $\subseteq$ 
( $\bigcup$  (ikst ` dualst ` S))  $\cup$  (ikest A)" (is ?A)
    " $\bigcup$  (assignment_rhsst ` S')  $\cup$  (assignment_rhsest A')  $\subseteq$ 
( $\bigcup$  (assignment_rhsst ` S))  $\cup$  (assignment_rhsest A)" (is ?B)
<proof>

```

Transition Systems Definitions

inductive pts_symbolic::

```

"((fun, 'var) strands  $\times$  (fun, 'var) strand)  $\Rightarrow$ 
((fun, 'var) strands  $\times$  (fun, 'var) strand)  $\Rightarrow$  bool"

```

(infix <=> 50) where

```

Nil[simp]:      "[ ]  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*$  (updatest S [ ], A)"
| Send[simp]:   "send⟨t⟩st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*$  (updatest S (send⟨t⟩st#S), A@[Step (receive⟨t⟩st)])"
| Receive[simp]: "receive⟨t⟩st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*$  (updatest S (receive⟨t⟩st#S), A@[Step (send⟨t⟩st)])"
| Equality[simp]: "(a: t  $\doteq$  t')st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*$  (updatest S ((a: t  $\doteq$  t')st#S), A@[Step ((a: t  $\doteq$  t')st)])"
| Inequality[simp]: " $\forall X(\forall \neq: F)$ st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*$  (updatest S ( $\forall X(\forall \neq: F)$ st#S), A@[Step ( $\forall X(\forall \neq: F)$ st)])"

```

private inductive pts_symbolic_c::

```

"((fun, 'var) strands  $\times$  (fun, 'var) extstrand)  $\Rightarrow$ 
((fun, 'var) strands  $\times$  (fun, 'var) extstrand)  $\Rightarrow$  bool"

```

(infix <=> 50) where

```

Nil[simp]:      "[ ]  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*_c$  (updatest S [ ], A)"
| Send[simp]:   "send⟨t⟩st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*_c$  (updatest S (send⟨t⟩st#S), A@[Step (receive⟨t⟩st)])"
| Receive[simp]: "receive⟨t⟩st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*_c$  (updatest S (receive⟨t⟩st#S), A@[Step (send⟨t⟩st)])"
| Equality[simp]: "(a: t  $\doteq$  t')st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*_c$  (updatest S ((a: t  $\doteq$  t')st#S), A@[Step ((a: t  $\doteq$  t')st)])"
| Inequality[simp]: " $\forall X(\forall \neq: F)$ st#S  $\in$  S  $\implies$  (S, A)  $\Rightarrow^*_c$  (updatest S ( $\forall X(\forall \neq: F)$ st#S), A@[Step ( $\forall X(\forall \neq: F)$ st)])"
| Decompose[simp]: "Fun f T  $\in$  subtermsset (ikest A  $\cup$  assignment_rhsest A)
 $\implies$  (S, A)  $\Rightarrow^*_c$  (S, A@[Decomp (Fun f T)])"

```

abbreviation pts_symbolic_rtrancl (infix <=> 50) where "a \Rightarrow^{**} b \equiv pts_symbolic** a b"

private abbreviation pts_symbolic_c_rtrancl (infix <=> 50) where "a \Rightarrow^{*_c} b \equiv pts_symbolic_c** a b"

lemma pts_symbolic_induct[consumes 1, case_names Nil Send Receive Equality Inequality]:

```

  assumes "(S, A)  $\Rightarrow^*$  (S', A')"
  and "[ [ ]  $\in$  S; S' = updatest S [ ]; A' = A ]  $\implies$  P"

```

3 The Typing Result for Non-Stateful Protocols

```

and " $\wedge t S. \llbracket \text{send}(t)_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S (\text{send}(t)_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\text{receive}(t)_{st}] \rrbracket \implies P''$ "
and " $\wedge t S. \llbracket \text{receive}(t)_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S (\text{receive}(t)_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\text{send}(t)_{st}] \rrbracket \implies P''$ "
and " $\wedge a t t' S. \llbracket (a: t \doteq t')_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S ((a: t \doteq t')_{st} \# S); \mathcal{A}' = \mathcal{A} @ [(a: t \doteq t')_{st}] \rrbracket \implies P''$ "
 $\implies P''$ 
and " $\wedge X F S. \llbracket \forall X (\forall \neq: F)_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S (\forall X (\forall \neq: F)_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\forall X (\forall \neq: F)_{st}] \rrbracket \implies P''$ "
shows "P"
<proof> lemma pts_symbolic_c_induct[consumes 1, case_names Nil Send Receive Equality Inequality Decompose]:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ "
  and " $\llbracket [] \in \mathcal{S}; S' = \text{update}_{st} S []; \mathcal{A}' = \mathcal{A} \rrbracket \implies P''$ "
  and " $\wedge t S. \llbracket \text{send}(t)_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S (\text{send}(t)_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\text{Step} (\text{receive}(t)_{st})] \rrbracket \implies P''$ "
  and " $\wedge t S. \llbracket \text{receive}(t)_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S (\text{receive}(t)_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\text{Step} (\text{send}(t)_{st})] \rrbracket \implies P''$ "
  and " $\wedge a t t' S. \llbracket (a: t \doteq t')_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S ((a: t \doteq t')_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\text{Step} ((a: t \doteq t')_{st})] \rrbracket \implies P''$ "
  and " $\wedge X F S. \llbracket \forall X (\forall \neq: F)_{st} \# S \in \mathcal{S}; S' = \text{update}_{st} S (\forall X (\forall \neq: F)_{st} \# S); \mathcal{A}' = \mathcal{A} @ [\text{Step} (\forall X (\forall \neq: F)_{st})] \rrbracket \implies P''$ "
  and " $\wedge f T. \llbracket \text{Fun } f T \in \text{subterms}_{set} (\text{ik}_{est} A \cup \text{assignment\_rhs}_{est} A); S' = S; \mathcal{A}' = \mathcal{A} @ [\text{Decomp} (\text{Fun } f T)] \rrbracket \implies P''$ "
  shows "P"
<proof> lemma pts_symbolic_c_preserves_wf_prot:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ " "wfsts' S A"
  shows "wfsts' S' A"
<proof> lemma pts_symbolic_c_preserves_wf_is:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ " "wfsts' S A" "wfst V (tost A)"
  shows "wfst V (tost A)'"
<proof> lemma pts_symbolic_c_preserves_tfrset:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ "
  and "tfrset (( $\bigcup$  (trmsst ` S))  $\cup$  (trmsest A))"
  and "wftrms (( $\bigcup$  (trmsst ` S))  $\cup$  (trmsest A))"
  shows "tfrset (( $\bigcup$  (trmsst ` S'))  $\cup$  (trmsest A'))  $\wedge$  wftrms (( $\bigcup$  (trmsst ` S'))  $\cup$  (trmsest A'))"
<proof> lemma pts_symbolic_c_preserves_tfrstp:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ " " $\forall S \in \mathcal{S} \cup \{\text{to}_{st} A\}. \text{list\_all } \text{tfr}_{stp} S$ "
  shows " $\forall S \in \mathcal{S}' \cup \{\text{to}_{st} A'\}. \text{list\_all } \text{tfr}_{stp} S$ "
<proof> lemma pts_symbolic_c_preserves_well_analyzed:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ " "well_analyzed A"
  shows "well_analyzed A'"
<proof> lemma pts_symbolic_c_preserves_Ana_invar_subst:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ "
  and "Ana_invar_subst (
    ( $\bigcup$  (ikst ` dualst ` S)  $\cup$  (ikest A))  $\cup$ 
    ( $\bigcup$  (assignment_rhsst ` S)  $\cup$  (assignment_rhsest A)))"
  shows "Ana_invar_subst (
    ( $\bigcup$  (ikst ` dualst ` S')  $\cup$  (ikest A'))  $\cup$ 
    ( $\bigcup$  (assignment_rhsst ` S')  $\cup$  (assignment_rhsest A')))"
<proof> lemma pts_symbolic_c_preserves_constr_disj_vars:
  assumes "(S,A)  $\Rightarrow^{\bullet_c} (S',A)''$ " "wfsts' S A" "fvest A  $\cap$  bvarsest A = {}"
  shows "fvest A'  $\cap$  bvarsest A' = {}"
<proof>

```

Theorem: The Typing Result Lifted to the Transition System Level

```

private lemma wfsts'_decomp_rm:
  assumes "well_analyzed A" "wfsts' S (decomp_rmest A)" shows "wfsts' S A"
<proof> lemma decompest_pts_symbolic_c:
  assumes "D  $\in$  decompest (ikest A) (assignment_rhsest A)  $\mathcal{I}$ "
  shows "(S,A)  $\Rightarrow^{\bullet_c} (S, A @ D)$ "
<proof> lemma pts_symbolic_to_pts_symbolic_c:
  assumes "(S, tost (decomp_rmest Ad))  $\Rightarrow^{**} (S', A)''$ " "semest_d {}  $\mathcal{I}$  (toest A)'" "semest_c {}  $\mathcal{I}$  Ad"
  and wf: "wfsts' S (decomp_rmest Ad)" "wfest {} Ad"
  and tar: "Ana_invar_subst (( $\bigcup$  (ikst ` dualst ` S)  $\cup$  (ikest Ad))
     $\cup$  ( $\bigcup$  (assignment_rhsst ` S)  $\cup$  (assignment_rhsest Ad)))"
  and wa: "well_analyzed Ad"
  and  $\mathcal{I}$ : "interpretationsubst  $\mathcal{I}$ "

```

```

shows "∃ A_d'. A' = to_st (decomp_rm_est A_d') ∧ (S, A_d) ⇒•* (S', A_d') ∧ sem_est_c {} I A_d'"
⟨proof⟩ lemma pts_symbolic_c_to_pts_symbolic:
  assumes "(S, A) ⇒•* (S', A)" "sem_est_c {} I A'"
  shows "(S, to_st (decomp_rm_est A)) ⇒•* (S', to_st (decomp_rm_est A'))"
    "sem_est_d {} I (decomp_rm_est A)'"
⟨proof⟩ lemma pts_symbolic_to_pts_symbolic_c_from_initial:
  assumes "(S_0, []) ⇒•* (S, A)" "I ⊨ ⟨A⟩" "wf_sts' S_0 []"
  and "Ana_invar_subst (⋃ (ik_st ` dual_st ` S_0) ∪ ⋃ (assignment_rhs_st ` S_0))" "interpretation_subst I"
  shows "∃ A_d. A = to_st (decomp_rm_est A_d) ∧ (S_0, []) ⇒•* (S, A_d) ∧ (I ⊨_c ⟨to_st A_d⟩)"
⟨proof⟩ lemma pts_symbolic_c_to_pts_symbolic_from_initial:
  assumes "(S_0, []) ⇒•* (S, A)" "I ⊨_c ⟨to_st A⟩"
  shows "(S_0, []) ⇒•* (S, to_st (decomp_rm_est A))" "I ⊨ ⟨to_st (decomp_rm_est 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 S_0" "tfr_set (⋃ (trms_st ` S_0))" "wf_trms (⋃ (trms_st ` S_0))" "∀ S ∈ S_0. list_all tfr_stp S"
  and "Ana_invar_subst (⋃ (ik_st ` dual_st ` S_0) ∪ ⋃ (assignment_rhs_st ` S_0))"
  and "(S_0, []) ⇒•* (S, A)" "interpretation_subst I" "I ⊨ ⟨A, Var⟩"
  shows "∃ I_τ. interpretation_subst I_τ ∧ (I_τ ⊨ ⟨A, Var⟩) ∧ wt_subst I_τ ∧ wf_trms (subst_range I_τ)"
⟨proof⟩

```

Corollary: The Typing Result on the Level of Constraints

There exists well-typed models of satisfiable type-flaw resistant constraints

```

corollary wt_attack_if_tfr_attack_d:
  assumes "wf_st {} A" "fv_st A ∩ bvars_st A = {}" "tfr_st A" "wf_trms (trms_st A)"
  and "Ana_invar_subst (ik_st A ∪ assignment_rhs_st A)"
  and "interpretation_subst I" "I ⊨ ⟨A⟩"
  shows "∃ I_τ. interpretation_subst I_τ ∧ (I_τ ⊨ ⟨A⟩) ∧ wt_subst I_τ ∧ wf_trms (subst_range I_τ)"
⟨proof⟩

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

```
theory Stateful_Strands
imports Strands_and_Constraints
begin
```

4.1.1 Stateful Constraints

```
datatype (funssstp: 'a, varssstp: 'b) stateful_strand_step =
  Send (the_msgs: "('a,'b) term list") (<send<_>> 80)
| Receive (the_msgs: "('a,'b) term list") (<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 (bvarssstp: "'b list")
  (the_eqs: "'('a,'b) term × ('a,'b) term) list")
  (the_ins: "'('a,'b) term × ('a,'b) term) list")
  (<∀_ (∀≠: _ ∉ _)>> [80,80])
where
  "bvarssstp (Send _) = []"
| "bvarssstp (Receive _) = []"
| "bvarssstp (Equality _ _ _) = []"
| "bvarssstp (Insert _ _) = []"
| "bvarssstp (Delete _ _) = []"
| "bvarssstp (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_Check_or_Assignment x ≡ is_Equality x ∨ is_InSet x ∨ 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
```

4 The Typing Result for Stateful Protocols

" $\langle t \neq s \rangle \equiv \text{NegChecks } [] [(t,s)] []$ "

abbreviation *NegChecks_Inequality2* ($\langle \forall _ \langle _ \neq _ \rangle \rangle$) where

" $\forall x \langle t \neq s \rangle \equiv \text{NegChecks } [x] [(t,s)] []$ "

abbreviation *NegChecks_Inequality3* ($\langle \forall _ , _ \langle _ \neq _ \rangle \rangle$) where

" $\forall x,y \langle t \neq s \rangle \equiv \text{NegChecks } [x,y] [(t,s)] []$ "

abbreviation *NegChecks_Inequality4* ($\langle \forall _ , _ , _ \langle _ \neq _ \rangle \rangle$) where

" $\forall x,y,z \langle t \neq s \rangle \equiv \text{NegChecks } [x,y,z] [(t,s)] []$ "

abbreviation *NegChecks_NotInSet1* ($\langle _ \text{ not in } _ \rangle$) where

" $\langle t \text{ not in } s \rangle \equiv \text{NegChecks } [] [] [(t,s)]$ "

abbreviation *NegChecks_NotInSet2* ($\langle \forall _ \langle _ \text{ not in } _ \rangle \rangle$) where

" $\forall x \langle t \text{ not in } s \rangle \equiv \text{NegChecks } [x] [] [(t,s)]$ "

abbreviation *NegChecks_NotInSet3* ($\langle \forall _ , _ \langle _ \text{ not in } _ \rangle \rangle$) where

" $\forall x,y \langle t \text{ not in } s \rangle \equiv \text{NegChecks } [x,y] [] [(t,s)]$ "

abbreviation *NegChecks_NotInSet4* ($\langle \forall _ , _ , _ \langle _ \text{ not in } _ \rangle \rangle$) where

" $\forall x,y,z \langle t \text{ not in } s \rangle \equiv \text{NegChecks } [x,y,z] [] [(t,s)]$ "

fun *trms_{sstp}* where

"*trms_{sstp}* (Send *ts*) = set *ts*"
| "*trms_{sstp}* (Receive *ts*) = set *ts*"
| "*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* \cup *trms_{pairs}* *F'*"

definition *trms_{sst}* where "*trms_{sst}* *S* $\equiv \bigcup (\text{trms}_{sstp} \ ` \ \text{set } S)$ "

declare *trms_{sst_def}*[simp]

fun *trms_list_{sstp}* where

"*trms_list_{sstp}* (Send *ts*) = *ts*"
| "*trms_list_{sstp}* (Receive *ts*) = *ts*"
| "*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 ($\lambda(t,t').$ [*t*,*t'*]) (*F@F'*))"

definition *trms_list_{sst}* where "*trms_list_{sst}* *S* $\equiv \text{remdups } (\text{concat } (\text{map } \text{trms_list}_{sstp} \ S))$ "

definition *ik_{sst}* where "*ik_{sst}* *A* $\equiv \{t \mid t \text{ ts. Receive } ts \in \text{set } A \wedge t \in \text{set } ts\}$ "

definition *bvars_{sst}* :: "('a,'b) stateful_strand \Rightarrow 'b set" where

"*bvars_{sst}* *S* $\equiv \bigcup (\text{set } (\text{map } (\text{set } \circ \text{bvars}_{sstp}) \ S))$ "

fun *fv_{sstp}* :: "('a,'b) stateful_strand_step \Rightarrow 'b set" where

"*fv_{sstp}* (Send *ts*) = *fv_{set}* (set *ts*)"
| "*fv_{sstp}* (Receive *ts*) = *fv_{set}* (set *ts*)"
| "*fv_{sstp}* (Equality $_ t t'$) = *fv* *t* \cup *fv* *t'*"
| "*fv_{sstp}* (Insert *t t'*) = *fv* *t* \cup *fv* *t'*"
| "*fv_{sstp}* (Delete *t t'*) = *fv* *t* \cup *fv* *t'*"
| "*fv_{sstp}* (InSet $_ t t'$) = *fv* *t* \cup *fv* *t'*"
| "*fv_{sstp}* (NegChecks *X F F'*) = *fv_{pairs}* *F* \cup *fv_{pairs}* *F'* - set *X*"

definition *fv_{sst}* :: "('a,'b) stateful_strand \Rightarrow 'b set" where

"*fv_{sst}* *S* $\equiv \bigcup (\text{set } (\text{map } \text{fv}_{sstp} \ S))$ "

```

fun fv_listsstp where
  "fv_listsstp (send(ts)) = concat (map fv_list ts)"
  | "fv_listsstp (receive⟨ts⟩) = concat (map fv_list ts)"
  | "fv_listsstp (⟨_ : t ≐ s⟩) = fv_list t@fv_list s"
  | "fv_listsstp (insert⟨t,s⟩) = fv_list t@fv_list s"
  | "fv_listsstp (delete⟨t,s⟩) = fv_list t@fv_list s"
  | "fv_listsstp (⟨_ : t ∈ s⟩) = fv_list t@fv_list s"
  | "fv_listsstp (∀X(∀≠: F ∀≠: F')) = filter (λx. x ∉ set X) (fv_listpairs (F@F'))"

definition fv_listsst where
  "fv_listsst S ≡ remdups (concat (map fv_listsstp S))"

declare bvarssst_def[simp]
declare fvsst_def[simp]

definition varssst::("a,'b) stateful_strand ⇒ 'b set" where
  "varssst S ≡ ⋃ (set (map varssstp S))"

abbreviation wfrestrictedvarssstp::("a,'b) stateful_strand_step ⇒ 'b set" where
  "wfrestrictedvarssstp x ≡
    case x of
      NegChecks _ _ _ ⇒ {}
    | Equality Check _ _ ⇒ {}
    | InSet Check _ _ ⇒ {}
    | Delete _ _ ⇒ {}
    | _ ⇒ varssstp x"

definition wfrestrictedvarssst::("a,'b) stateful_strand ⇒ 'b set" where
  "wfrestrictedvarssst S ≡ ⋃ (set (map wfrestrictedvarssstp S))"

abbreviation wfvarsoccssstp where
  "wfvarsoccssstp x ≡
    case x of
      Send ts ⇒ fvset (set ts)
    | Equality Assign s t ⇒ fv s
    | InSet Assign s t ⇒ fv s ∪ fv t
    | _ ⇒ {}"

definition wfvarsoccssst where
  "wfvarsoccssst S ≡ ⋃ (set (map wfvarsoccssstp S))"

fun wf'sst::"b set ⇒ ('a,'b) stateful_strand ⇒ bool" where
  "wf'sst V [] = True"
  | "wf'sst V (Receive ts#S) = (fvset (set ts) ⊆ V ∧ wf'sst V S)"
  | "wf'sst V (Send ts#S) = wf'sst (V ∪ fvset (set ts)) 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 "wfsst S ≡ wf'sst {} S ∧ fvsst S ∩ bvarssst 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⟨ts⟩ ·sstp ϑ = send⟨ts ·list ϑ⟩"
  | "receive⟨ts⟩ ·sstp ϑ = receive⟨ts ·list 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⟩"

```

4 The Typing Result for Stateful Protocols

```
| "delete(t,s) ·sstp ∅ = delete(t · ∅, s · ∅)"
| "∀X(∀≠: F ∨≠: G) ·sstp ∅ = ∀X(∀≠: (F ·pairs rm_vars (set X) ∅) ∨≠: (G ·pairs rm_vars (set X) ∅))"
```

```
definition subst_apply_stateful_strand::
  "('a,'b) stateful_strand ⇒ ('a,'b) subst ⇒ ('a,'b) stateful_strand"
  (infix <·sst> 51) where
  "S ·sst ∅ ≡ map (λx. x ·sstp ∅) 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 is_Check_or_Assignment_iff[simp]:
  "is_Check x ∨ is_Assignment x ↔ is_Check_or_Assignment x"
<proof>
```

```
lemma subst_apply_stateful_strand_step_Inequality[simp]:
  "⟨t != s⟩ ·sstp ∅ = ⟨t · ∅ != s · ∅⟩"
  "∀x⟨t != s⟩ ·sstp ∅ = ∀x⟨t · rm_vars {x} ∅ != s · rm_vars {x} ∅⟩"
  "∀x,y⟨t != s⟩ ·sstp ∅ = ∀x,y⟨t · rm_vars {x,y} ∅ != s · rm_vars {x,y} ∅⟩"
  "∀x,y,z⟨t != s⟩ ·sstp ∅ = ∀x,y,z⟨t · rm_vars {x,y,z} ∅ != s · rm_vars {x,y,z} ∅⟩"
<proof>
```

```
lemma subst_apply_stateful_strand_step_NotInSet[simp]:
  "⟨t not in s⟩ ·sstp ∅ = ⟨t · ∅ not in s · ∅⟩"
  "∀x⟨t not in s⟩ ·sstp ∅ = ∀x⟨t · rm_vars {x} ∅ not in s · rm_vars {x} ∅⟩"
```

" $\forall x,y\langle t \text{ not in } s \rangle \cdot_{sstp} \vartheta = \forall x,y\langle t \cdot \text{rm_vars } \{x,y\} \vartheta \text{ not in } s \cdot \text{rm_vars } \{x,y\} \vartheta \rangle$ "
" $\forall x,y,z\langle t \text{ not in } s \rangle \cdot_{sstp} \vartheta = \forall x,y,z\langle t \cdot \text{rm_vars } \{x,y,z\} \vartheta \text{ not in } s \cdot \text{rm_vars } \{x,y,z\} \vartheta \rangle$ "
<proof>

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 `varssstp_finite[simp]`: "`finite (varssstp x)`"
<proof>

lemma `varssst_finite[simp]`: "`finite (varssst S)`"
<proof>

lemma `fvsstp_finite[simp]`: "`finite (fvsstp x)`"
<proof>

lemma `fvsst_finite[simp]`: "`finite (fvsst S)`"
<proof>

lemma `bvarssstp_finite[simp]`: "`finite (set (bvarssstp x))`"
<proof>

lemma `bvarssst_finite[simp]`: "`finite (bvarssst S)`"
<proof>

lemma `substsst_nil[simp]`: "`[] ·sst δ = []`"
<proof>

lemma `dbsst_nil[simp]`: "`dbsst [] \mathcal{I} = []`"
<proof>

lemma `iksst_nil[simp]`: "`iksst [] = {}`"
<proof>

lemma `in_iksst_iff`: "`t ∈ iksst A \longleftrightarrow (\exists ts. receive⟨ts⟩ ∈ set A ∧ t ∈ set ts)`"
<proof>

lemma `iksst_append[simp]`: "`iksst (A@B) = iksst A ∪ iksst B`"
<proof>

lemma `iksst_concat`: "`iksst (concat xs) = \bigcup (iksst ` set xs)`"
<proof>

lemma `iksst_subst`: "`iksst (A ·sst δ) = iksst A ·set δ`"
<proof>

lemma `iksst_set_subset`:
"`set A ⊆ set B \implies iksst A ⊆ iksst B`"

<proof>

lemma *ik_{sst}_prefix_subset*:

"prefix A B \implies ik_{sst} A \subseteq ik_{sst} B" (is "?P A B \implies ?P' A B")

"prefix A (C@D) \implies \neg prefix A C \implies ik_{sst} C \subseteq ik_{sst} A" (is "?Q \implies ?Q' \implies ?Q''")

<proof>

lemma *ik_{sst}_snoc_no_receive_empty*:

assumes " $\forall a \in \text{set } A. \neg \text{is_Receive } a$ "

shows "ik_{sst} A ·_{set} I = {}"

<proof>

lemma *ik_{sst}_snoc_no_receive_eq*:

assumes " $\nexists s. a = \text{receive}(s)$ "

shows "ik_{sst} (A@[a]) ·_{set} I = ik_{sst} A ·_{set} I"

<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 *db_{sst}_no_upd*:

assumes " $\forall a \in \text{set } A. \neg \text{is_Insert } a \wedge \neg \text{is_Delete } a$ "

shows "db'_{sst} A I D = D"

<proof>

lemma *db_{sst}_no_upd_append*:

assumes " $\forall b \in \text{set } B. \neg \text{is_Insert } b \wedge \neg \text{is_Delete } b$ "

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) \in set (db'_{sst} A I D)"

shows "(t,s) \in set D \vee ($\exists t' s'. \text{insert}(t',s') \in \text{set } A \wedge t = t' \cdot I \wedge s = s' \cdot I$)"

<proof>

lemma *db_{sst}_in_cases'*:

assumes "(t,s) \in set (db'_{sst} A I D)"

and "(t,s) \notin set D"

shows " $\exists B C t' s'. A = B@C \wedge \text{insert}(t',s') \in C \wedge t = t' \cdot I \wedge s = s' \cdot I \wedge$
 $(\forall t'' s''. \text{delete}(t'',s'') \in \text{set } C \longrightarrow t \neq t'' \cdot I \vee s \neq s'' \cdot I)$ "

<proof>

lemma *db_{sst}_filter*:

"db'_{sst} A I D = db'_{sst} (filter is_Update A) I D"

<proof>

lemma *db_{sst}_subst_swap*:

assumes " $\forall x \in \text{fv}_{sst} A. I x = J x$ "

shows "db'_{sst} A I D = db'_{sst} A J D"

<proof>

lemma *dbupd_{sst}_no_upd*:

assumes " $\forall a \in \text{set } A. \neg \text{is_Insert } a \wedge \neg \text{is_Delete } a$ "

shows "dbupd_{sst} A I D = D"

<proof>

lemma *dbupd_{sst}_no_deletes*:

assumes "list_all ($\lambda a. \neg \text{is_Delete } a$) A"

shows "dbupd_{sst} A I D = D \cup {(t · I, s · I) | t s. insert(t,s) \in set A}" (is "?Q A D")

<proof>

lemma $\text{dbupd}_{sst_append}$:

" $\text{dbupd}_{sst} (A@B) I D = \text{dbupd}_{sst} B I (\text{dbupd}_{sst} A I D)$ "

<proof>

lemma $\text{dbupd}_{sst_filter}$:

" $\text{dbupd}_{sst} A I D = \text{dbupd}_{sst} (\text{filter } \text{is_Update } A) I D$ "

<proof>

lemma $\text{dbupd}_{sst_in_cases}$:

assumes " $(t,s) \in \text{dbupd}_{sst} A I D$ "

shows " $(t,s) \in D \vee (\exists t' s'. \text{insert}(t',s') \in \text{set } A \wedge t = t' \cdot I \wedge s = s' \cdot I)$ " (is ?P)

and " $\forall u v B. \text{suffix}(\text{delete}(u,v)\#B) A \wedge (t,s) = (u,v) \cdot_p I \longrightarrow$

$(\exists u' v'. (t,s) = (u',v') \cdot_p I \wedge \text{insert}(u',v') \in \text{set } B)$ " (is ?Q)

<proof>

lemma $\text{dbupd}_{sst_in_iff}$:

" $(t,s) \in \text{dbupd}_{sst} A I D \longleftrightarrow$

$(\forall u v B. \text{suffix}(\text{delete}(u,v)\#B) A \wedge (t,s) = (u,v) \cdot_p I \longrightarrow$

$(\exists u' v'. (t,s) = (u',v') \cdot_p I \wedge \text{insert}(u',v') \in \text{set } B)) \wedge$

$((t,s) \in D \vee (\exists u v. (t,s) = (u,v) \cdot_p I \wedge \text{insert}(u,v) \in \text{set } A))$ "

(is "?P A D \longleftrightarrow ?Q1 A \wedge ?Q2 A D")

<proof>

lemma $\text{dbupd}_{sst_in_cases'}$:

fixes $A::('a, 'b) \text{stateful_strand}$

assumes " $(t,s) \in \text{dbupd}_{sst} A I D$ "

and " $(t,s) \notin D$ "

shows " $\exists B C t' s'. A = B@insert(t',s')\#C \wedge t = t' \cdot I \wedge s = s' \cdot I \wedge$

$(\forall t'' s''. \text{delete}(t'',s'') \in \text{set } C \longrightarrow t \neq t'' \cdot I \vee s \neq s'' \cdot I)$ "

<proof>

lemma dbupd_{sst_mono} :

assumes " $D \subseteq E$ "

shows " $\text{dbupd}_{sst} A I D \subseteq \text{dbupd}_{sst} A I E$ "

<proof>

lemma $\text{dbupd}_{sst_db_narrow}$:

assumes " $(t,s) \in \text{dbupd}_{sst} A I (D \cup E)$ "

and " $(t,s) \notin D$ "

shows " $(t,s) \in \text{dbupd}_{sst} A I E$ "

<proof>

lemma $\text{dbupd}_{sst_set_term_neq_in_iff}$:

assumes $f: "f \neq k"$

and $A: "\forall t s. \text{insert}(t,s) \in \text{set } A \longrightarrow (\exists g ts. s = \text{Fun } g \text{ ts})"$

shows " $(t, \text{Fun } f \text{ ts}) \in \text{dbupd}_{sst} A I D \longleftrightarrow$

$(t, \text{Fun } f \text{ ts}) \in \text{dbupd}_{sst} (\text{filter } (\lambda a. \exists s ss. a = \text{insert}(s, \text{Fun } k \text{ ss})) A) I D$ "

(is "?P A D \longleftrightarrow ?P (?f A) D")

<proof>

lemma $\text{dbupd}_{sst_subst_const_swap}$:

fixes $t s$

defines " $\text{fvs} \equiv \lambda A D. \text{fv}_{sst} A \cup \text{fv } t \cup \text{fv } s \cup \bigcup (\text{fv}_{pair} \ ` D)$ "

assumes " $(t \cdot \delta, s \cdot \delta) \in \text{dbupd}_{sst} A \delta (D \cdot_{pset} \delta)$ " (is "?in δ A D")

and " $\forall x \in \text{fvs } A D.$

$\delta x = \vartheta x \vee$

$(\neg(\delta x \sqsubseteq t) \wedge \neg(\delta x \sqsubseteq s) \wedge \neg(\vartheta x \sqsubseteq t) \wedge \neg(\vartheta x \sqsubseteq s) \wedge$

$(\forall (u,v) \in D. \neg(\delta x \sqsubseteq u) \wedge \neg(\delta x \sqsubseteq v) \wedge \neg(\vartheta x \sqsubseteq u) \wedge \neg(\vartheta x \sqsubseteq v)) \wedge$

$(\forall u v. \text{insert}(u,v) \in \text{set } A \vee \text{delete}(u,v) \in \text{set } A \longrightarrow$

$\neg(\delta x \sqsubseteq u) \wedge \neg(\delta x \sqsubseteq v) \wedge \neg(\vartheta x \sqsubseteq u) \wedge \neg(\vartheta x \sqsubseteq v))$ "

(is "?A $\delta \vartheta$ D")

4 The Typing Result for Stateful Protocols

and " $\forall x \in fvs A D. \exists c. \delta x = \text{Fun } c []$ " (is "?B δ ")
and " $\forall x \in fvs A D. \exists c. \vartheta x = \text{Fun } c []$ " (is "?B ϑ ")
and " $\forall x \in fvs A D. \forall y \in fvs A D. \delta x = \delta y \longleftrightarrow \vartheta x = \vartheta y$ " (is "?C $\delta \vartheta A D$ ")
shows " $(t \cdot \vartheta, s \cdot \vartheta) \in \text{dbupd}_{sst} A \vartheta (D \cdot_{pset} \vartheta)$ " (is "?in $\vartheta A D$ ")
 <proof>

lemma `subst_sst_cons`: " $a\#A \cdot_{sst} \delta = (a \cdot_{sstp} \delta)\#(A \cdot_{sst} \delta)$ "
 <proof>

lemma `subst_sst_snoc`: " $A@[a] \cdot_{sst} \delta = (A \cdot_{sst} \delta)@[a \cdot_{sstp} \delta]$ "
 <proof>

lemma `subst_sst_append[simp]`: " $A@B \cdot_{sst} \delta = (A \cdot_{sst} \delta)@(B \cdot_{sst} \delta)$ "
 <proof>

lemma `subst_sst_list_all`:
 "`list_all is_Send S` \longleftrightarrow `list_all is_Send (S $\cdot_{sst} \delta$)`"
 "`list_all is_Receive S` \longleftrightarrow `list_all is_Receive (S $\cdot_{sst} \delta$)`"
 "`list_all is_Equality S` \longleftrightarrow `list_all is_Equality (S $\cdot_{sst} \delta$)`"
 "`list_all is_Insert S` \longleftrightarrow `list_all is_Insert (S $\cdot_{sst} \delta$)`"
 "`list_all is_Delete S` \longleftrightarrow `list_all is_Delete (S $\cdot_{sst} \delta$)`"
 "`list_all is_InSet S` \longleftrightarrow `list_all is_InSet (S $\cdot_{sst} \delta$)`"
 "`list_all is_NegChecks S` \longleftrightarrow `list_all is_NegChecks (S $\cdot_{sst} \delta$)`"
 "`list_all is_Assignment S` \longleftrightarrow `list_all is_Assignment (S $\cdot_{sst} \delta$)`"
 "`list_all is_Check S` \longleftrightarrow `list_all is_Check (S $\cdot_{sst} \delta$)`"
 "`list_all is_Update S` \longleftrightarrow `list_all is_Update (S $\cdot_{sst} \delta$)`"
 "`list_all is_Check_or_Assignment S` \longleftrightarrow `list_all is_Check_or_Assignment (S $\cdot_{sst} \delta$)`"
 <proof>

lemma `subst_sstp_id_subst`: " $a \cdot_{sstp} \text{Var} = a$ "
 <proof>

lemma `subst_sst_id_subst`: " $A \cdot_{sst} \text{Var} = A$ "
 <proof>

lemma `sst_vars_append_subset`:
 " $fv_{sst} A \subseteq fv_{sst} (A@B)$ " " $bvars_{sst} A \subseteq bvars_{sst} (A@B)$ "
 " $fv_{sst} B \subseteq fv_{sst} (A@B)$ " " $bvars_{sst} B \subseteq bvars_{sst} (A@B)$ "
 <proof>

lemma `sst_vars_disj_cons[simp]`: " $fv_{sst} (a\#A) \cap bvars_{sst} (a\#A) = \{\} \implies fv_{sst} A \cap bvars_{sst} A = \{\}$ "
 <proof>

lemma `fv_sst_cons_subset[simp]`: " $fv_{sst} A \subseteq fv_{sst} (a\#A)$ "
 <proof>

lemma `fv_sstp_subst_cases[simp]`:
 " $fv_{sstp} (\text{send}(ts) \cdot_{sstp} \vartheta) = fv_{set} (\text{set } ts \cdot_{set} \vartheta)$ "
 " $fv_{sstp} (\text{receive}(ts) \cdot_{sstp} \vartheta) = fv_{set} (\text{set } ts \cdot_{set} \vartheta)$ "
 " $fv_{sstp} ((c: t \doteq s) \cdot_{sstp} \vartheta) = fv (t \cdot \vartheta) \cup fv (s \cdot \vartheta)$ "
 " $fv_{sstp} (\text{insert}(t,s) \cdot_{sstp} \vartheta) = fv (t \cdot \vartheta) \cup fv (s \cdot \vartheta)$ "
 " $fv_{sstp} (\text{delete}(t,s) \cdot_{sstp} \vartheta) = fv (t \cdot \vartheta) \cup fv (s \cdot \vartheta)$ "
 " $fv_{sstp} ((c: t \in s) \cdot_{sstp} \vartheta) = fv (t \cdot \vartheta) \cup fv (s \cdot \vartheta)$ "
 " $fv_{sstp} (\forall X (\forall \neq: F \vee \notin: G) \cdot_{sstp} \vartheta) =$
 $fv_{pairs} (F \cdot_{pairs} \text{rm_vars} (\text{set } X) \vartheta) \cup fv_{pairs} (G \cdot_{pairs} \text{rm_vars} (\text{set } X) \vartheta) - \text{set } X$ "
 <proof>

lemma `vars_sstp_cases[simp]`:
 " $vars_{sstp} (\text{send}(ts)) = fv_{set} (\text{set } ts)$ "
 " $vars_{sstp} (\text{receive}(ts)) = fv_{set} (\text{set } ts)$ "
 " $vars_{sstp} ((c: t \doteq s)) = fv t \cup fv s$ "
 " $vars_{sstp} (\text{insert}(t,s)) = fv t \cup fv s$ "
 " $vars_{sstp} (\text{delete}(t,s)) = fv t \cup fv s$ "

```

"varssstp (<c: t ∈ s>) = fv t ∪ fv s"
"varssstp (∀X(∀≠: F ∨≠: G)) = fvpairs F ∪ fvpairs G ∪ set X" (is ?A)
"varssstp (∀X(∀≠: [(t,s)] ∨≠: [])) = fv t ∪ fv s ∪ set X" (is ?B)
"varssstp (∀X(∀≠: [] ∨≠: [(t,s)])) = fv t ∪ fv s ∪ set X" (is ?C)
<proof>

```

lemma vars_{sstp}_subst_cases[simp]:

```

"varssstp (send<ts> .sstp ∅) = fvset (set ts .set ∅)"
"varssstp (receive<ts> .sstp ∅) = fvset (set ts .set ∅)"
"varssstp (<c: t ≐ s> .sstp ∅) = fv (t . ∅) ∪ fv (s . ∅)"
"varssstp (insert<t,s> .sstp ∅) = fv (t . ∅) ∪ fv (s . ∅)"
"varssstp (delete<t,s> .sstp ∅) = fv (t . ∅) ∪ fv (s . ∅)"
"varssstp (<c: t ∈ s> .sstp ∅) = fv (t . ∅) ∪ fv (s . ∅)"
"varssstp (∀X(∀≠: F ∨≠: G) .sstp ∅) =
  fvpairs (F .pairs rm_vars (set X) ∅) ∪ fvpairs (G .pairs rm_vars (set X) ∅) ∪ set X" (is ?A)
"varssstp (∀X(∀≠: [(t,s)] ∨≠: [])) .sstp ∅ =
  fv (t . rm_vars (set X) ∅) ∪ fv (s . rm_vars (set X) ∅) ∪ set X" (is ?B)
"varssstp (∀X(∀≠: [] ∨≠: [(t,s)])) .sstp ∅ =
  fv (t . rm_vars (set X) ∅) ∪ fv (s . rm_vars (set X) ∅) ∪ 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 (bvarssstp (send<ts>)) = {}"
"set (bvarssstp (receive<ts>)) = {}"
"set (bvarssstp (<c: t ≐ s>)) = {}"
"set (bvarssstp (insert<t,s>)) = {}"
"set (bvarssstp (delete<t,s>)) = {}"
"set (bvarssstp (<c: t ∈ s>)) = {}"
"set (bvarssstp (∀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]:

```

"varssst (send<ts>#A) = varssstp (send<ts>) ∪ varssst A"

```

4 The Typing Result for Stateful Protocols

```

"varssst (receive⟨ts⟩#A) = varssstp (receive⟨ts⟩) ∪ varssst A"
"varssst (⟨a: t ≐ s⟩#A) = varssstp (⟨a: t ≐ s⟩) ∪ varssst A"
"varssst (insert⟨t,s⟩#A) = varssstp (insert⟨t,s⟩) ∪ varssst A"
"varssst (delete⟨t,s⟩#A) = varssstp (delete⟨t,s⟩) ∪ varssst A"
"varssst (⟨a: t ∈ s⟩#A) = varssstp (⟨a: t ∈ s⟩) ∪ varssst A"
"varssst (∀X⟨∇≠: F ∇∉: G⟩#A) = varssstp (∀X⟨∇≠: F ∇∉: G⟩) ∪ varssst A"
⟨proof⟩

```

```

lemma fvsstp_subst_if_no_bvars:
  assumes a: "bvarssstp a = []"
  shows "fvsstp (a ·sstp ∅) = fvset (∅ ` fvsstp a)"
⟨proof⟩

```

```

lemma fvsst_subst_if_no_bvars:
  assumes A: "bvarssst A = {}"
  shows "fvsst (A ·sst ∅) = fvset (∅ ` fvsst A)"
⟨proof⟩

```

```

lemma varssstp_is_fvsstp_bvarssstp:
  fixes x: "('a, 'b) stateful_strand_step"
  shows "varssstp x = fvsstp x ∪ set (bvarssstp x)"
⟨proof⟩

```

```

lemma varssst_is_fvsst_bvarssst:
  fixes S: "('a, 'b) stateful_strand"
  shows "varssst S = fvsst S ∪ bvarssst S"
⟨proof⟩

```

```

lemma varssstp_NegCheck[simp]:
  "varssstp (∀X⟨∇≠: F ∇∉: G⟩) = set X ∪ fvpairs F ∪ fvpairs G"
⟨proof⟩

```

```

lemma bvarssstp_NegCheck[simp]:
  "bvarssstp (∀X⟨∇≠: F ∇∉: G⟩) = X"
  "set (bvarssstp (∀ []⟨∇≠: F ∇∉: G⟩)) = {}"
⟨proof⟩

```

```

lemma fvsstp_NegCheck[simp]:
  "fvsstp (∀X⟨∇≠: F ∇∉: G⟩) = fvpairs F ∪ fvpairs G - set X"
  "fvsstp (∀ []⟨∇≠: F ∇∉: G⟩) = fvpairs F ∪ fvpairs G"
  "fvsstp (⟨t != s⟩) = fv t ∪ fv s"
  "fvsstp (⟨t not in s⟩) = fv t ∪ fv s"
⟨proof⟩

```

```

lemma fvsst_append[simp]: "fvsst (A@B) = fvsst A ∪ fvsst B"
⟨proof⟩

```

```

lemma bvarssst_append[simp]: "bvarssst (A@B) = bvarssst A ∪ bvarssst B"
⟨proof⟩

```

```

lemma fvsst_mono: "set A ⊆ set B ⇒ fvsst A ⊆ fvsst B"
⟨proof⟩

```

```

lemma fvsstp_is_subterm_trmssstp:
  assumes "x ∈ fvsstp a"
  shows "Var x ∈ subtermsset (trmssstp a)"
⟨proof⟩

```

```

lemma fvsst_is_subterm_trmssst: "x ∈ fvsst A ⇒ Var x ∈ subtermsset (trmssst A)"
⟨proof⟩

```

```

lemma var_subterm_trmssstp_is_varssstp:
  assumes "Var x ∈ subtermsset (trmssstp a)"

```

shows "x ∈ vars_{sstp} a"
 ⟨proof⟩

lemma var_subterm_trms_{sst}_is_vars_{sst}: "Var x ∈ subterms_{set} (trms_{sst} A) ⇒ x ∈ vars_{sst} A"
 ⟨proof⟩

lemma var_trms_{sst}_is_vars_{sst}: "Var x ∈ trms_{sst} A ⇒ x ∈ vars_{sst} A"
 ⟨proof⟩

lemma ik_{sst}_trms_{sst}_subset: "ik_{sst} A ⊆ trms_{sst} A"
 ⟨proof⟩

lemma var_subterm_ik_{sst}_is_vars_{sst}: "Var x ∈ subterms_{set} (ik_{sst} A) ⇒ x ∈ vars_{sst} A"
 ⟨proof⟩

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

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

lemma fv_trms_{sst}_subset:
 "fv_{set} (trms_{sst} S) ⊆ vars_{sst} S"
 "fv_{sst} S ⊆ 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} ∅)"
 shows "x ∈ vars_{sstp} s ∨ x ∈ fv_{set} (∅ ` vars_{sstp} s)"
 ⟨proof⟩

lemma vars_{sst}_subst_cases:
 assumes "x ∈ vars_{sst} (S ·_{sst} ∅)"
 shows "x ∈ vars_{sst} S ∨ x ∈ fv_{set} (∅ ` 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⟨ts⟩ ∈ set A ⇒ send⟨ts ·_{list} ∅⟩ ∈ set (A ·_{sst} ∅)"
 "receive⟨ts⟩ ∈ set A ⇒ receive⟨ts ·_{list} ∅⟩ ∈ set (A ·_{sst} ∅)"

4 The Typing Result for Stateful Protocols

$\langle c: t \doteq s \rangle \in \text{set } A \implies \langle c: (t \cdot \vartheta) \doteq (s \cdot \vartheta) \rangle \in \text{set } (A \cdot_{sst} \vartheta)$
 $\langle \text{insert}(t, s) \rangle \in \text{set } A \implies \langle \text{insert}(t \cdot \vartheta, s \cdot \vartheta) \rangle \in \text{set } (A \cdot_{sst} \vartheta)$
 $\langle \text{delete}(t, s) \rangle \in \text{set } A \implies \langle \text{delete}(t \cdot \vartheta, s \cdot \vartheta) \rangle \in \text{set } (A \cdot_{sst} \vartheta)$
 $\langle c: t \in s \rangle \in \text{set } A \implies \langle c: (t \cdot \vartheta) \in (s \cdot \vartheta) \rangle \in \text{set } (A \cdot_{sst} \vartheta)$
 $\forall X(\forall \neq: F \vee \notin: G) \in \text{set } A$
 $\implies \forall X(\forall \neq: (F \cdot_{pairs} \text{rm_vars } (\text{set } X) \vartheta) \vee \notin: (G \cdot_{pairs} \text{rm_vars } (\text{set } X) \vartheta)) \in \text{set } (A \cdot_{sst} \vartheta)$
 $\langle t \neq s \rangle \in \text{set } A \implies \langle t \cdot \vartheta \neq s \cdot \vartheta \rangle \in \text{set } (A \cdot_{sst} \vartheta)$
 $\langle t \text{ not in } s \rangle \in \text{set } A \implies \langle t \cdot \vartheta \text{ not in } s \cdot \vartheta \rangle \in \text{set } (A \cdot_{sst} \vartheta)$

$\langle \text{proof} \rangle$

lemma stateful_strand_step_cases_subst:

$\text{"is_Send } a = \text{is_Send } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Receive } a = \text{is_Receive } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Equality } a = \text{is_Equality } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Insert } a = \text{is_Insert } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Delete } a = \text{is_Delete } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_InSet } a = \text{is_InSet } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_NegChecks } a = \text{is_NegChecks } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Assignment } a = \text{is_Assignment } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Check } a = \text{is_Check } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Update } a = \text{is_Update } (a \cdot_{sstp} \vartheta)\text{"}$
 $\text{"is_Check_or_Assignment } a = \text{is_Check_or_Assignment } (a \cdot_{sstp} \vartheta)\text{"}$

$\langle \text{proof} \rangle$

lemma stateful_strand_step_substD:

$\text{"a } \cdot_{sstp} \sigma = \text{send}(ts) \implies \exists ts'. ts = ts' \cdot_{list} \sigma \wedge a = \text{send}(ts')\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \text{receive}(ts) \implies \exists ts'. ts = ts' \cdot_{list} \sigma \wedge a = \text{receive}(ts')\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \langle c: t \doteq s \rangle \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = \langle c: t' \doteq s' \rangle\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \text{insert}(t, s) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = \text{insert}(t', s')\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \text{delete}(t, s) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = \text{delete}(t', s')\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \langle c: t \in s \rangle \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = \langle c: t' \in s' \rangle\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \forall X(\forall \neq: F \vee \notin: G) \implies$
 $\quad \exists F' G'. F = F' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge G = G' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge$
 $\quad a = \forall X(\forall \neq: F' \vee \notin: G')\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \langle t \neq s \rangle \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = \langle t' \neq s' \rangle\text{"}$
 $\text{"a } \cdot_{sstp} \sigma = \langle t \text{ not in } s \rangle \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = \langle t' \text{ not in } s' \rangle\text{"}$

$\langle \text{proof} \rangle$

lemma stateful_strand_step_mem_substD:

$\text{"send}(ts) \in \text{set } (S \cdot_{sst} \sigma) \implies \exists ts'. ts = ts' \cdot_{list} \sigma \wedge \text{send}(ts') \in \text{set } S\text{"}$
 $\text{"receive}(ts) \in \text{set } (S \cdot_{sst} \sigma) \implies \exists ts'. ts = ts' \cdot_{list} \sigma \wedge \text{receive}(ts') \in \text{set } S\text{"}$
 $\langle c: t \doteq s \rangle \in \text{set } (S \cdot_{sst} \sigma) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge \langle c: t' \doteq s' \rangle \in \text{set } S\text{"}$
 $\text{insert}(t, s) \in \text{set } (S \cdot_{sst} \sigma) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge \text{insert}(t', s') \in \text{set } S\text{"}$
 $\text{delete}(t, s) \in \text{set } (S \cdot_{sst} \sigma) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge \text{delete}(t', s') \in \text{set } S\text{"}$
 $\langle c: t \in s \rangle \in \text{set } (S \cdot_{sst} \sigma) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge \langle c: t' \in s' \rangle \in \text{set } S\text{"}$
 $\forall X(\forall \neq: F \vee \notin: G) \in \text{set } (S \cdot_{sst} \sigma) \implies$
 $\quad \exists F' G'. F = F' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge G = G' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge$
 $\quad \forall X(\forall \neq: F' \vee \notin: G') \in \text{set } S\text{"}$
 $\langle t \neq s \rangle \in \text{set } (S \cdot_{sst} \sigma) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge \langle t' \neq s' \rangle \in \text{set } S\text{"}$
 $\langle t \text{ not in } s \rangle \in \text{set } (S \cdot_{sst} \sigma) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge \langle t' \text{ not in } s' \rangle \in \text{set } S\text{"}$

$\langle \text{proof} \rangle$

lemma stateful_strand_step_fv_subset_cases:

$\text{"send}(ts) \in \text{set } S \implies \text{fv}_{set}(\text{set } ts) \subseteq \text{fv}_{sst} S\text{"}$
 $\text{"receive}(ts) \in \text{set } S \implies \text{fv}_{set}(\text{set } ts) \subseteq \text{fv}_{sst} S\text{"}$
 $\langle c: t \doteq s \rangle \in \text{set } S \implies \text{fv } t \cup \text{fv } s \subseteq \text{fv}_{sst} S\text{"}$
 $\text{insert}(t, s) \in \text{set } S \implies \text{fv } t \cup \text{fv } s \subseteq \text{fv}_{sst} S\text{"}$
 $\text{delete}(t, s) \in \text{set } S \implies \text{fv } t \cup \text{fv } s \subseteq \text{fv}_{sst} S\text{"}$
 $\langle c: t \in s \rangle \in \text{set } S \implies \text{fv } t \cup \text{fv } s \subseteq \text{fv}_{sst} S\text{"}$
 $\forall X(\forall \neq: F \vee \notin: G) \in \text{set } S \implies \text{fv}_{pairs} F \cup \text{fv}_{pairs} G - \text{set } X \subseteq \text{fv}_{sst} S\text{"}$
 $\langle t \neq s \rangle \in \text{set } S \implies \text{fv } t \cup \text{fv } s \subseteq \text{fv}_{sst} S\text{"}$
 $\langle t \text{ not in } s \rangle \in \text{set } S \implies \text{fv } t \cup \text{fv } s \subseteq \text{fv}_{sst} S\text{"}$

$\langle \text{proof} \rangle$

```

lemma trmssst_nil[simp]:
  "trmssst [] = {}"
⟨proof⟩

lemma trmssst_mono:
  "set M ⊆ set N ⇒ trmssst M ⊆ trmssst N"
⟨proof⟩

lemma trmssst_memI[intro?]:
  "send⟨ts⟩ ∈ set S ⇒ t ∈ set ts ⇒ t ∈ trmssst S"
  "receive⟨ts⟩ ∈ set S ⇒ t ∈ set ts ⇒ t ∈ trmssst S"
  "(ac: t ≐ s) ∈ set S ⇒ t ∈ trmssst S"
  "(ac: t ≐ s) ∈ set S ⇒ s ∈ trmssst S"
  "insert(t,s) ∈ set S ⇒ t ∈ trmssst S"
  "insert(t,s) ∈ set S ⇒ s ∈ trmssst S"
  "delete(t,s) ∈ set S ⇒ t ∈ trmssst S"
  "delete(t,s) ∈ set S ⇒ s ∈ trmssst S"
  "∀X(∀≠: F ∨≠: G) ∈ set S ⇒ t ∈ trmspairs F ⇒ t ∈ trmssst S"
  "∀X(∀≠: F ∨≠: G) ∈ set S ⇒ t ∈ trmspairs G ⇒ t ∈ trmssst S"
⟨proof⟩

lemma trmssst_in:
  assumes "t ∈ trmssst S"
  shows "∃ a ∈ set S. t ∈ trmssstp a"
⟨proof⟩

lemma trmssst_cons: "trmssst (a#A) = trmssstp a ∪ trmssst A"
⟨proof⟩

lemma trmssst_append[simp]: "trmssst (A@B) = trmssst A ∪ trmssst B"
⟨proof⟩

lemma trmssstp_subst:
  assumes "set (bvarssstp a) ∩ subst_domain ∅ = {}"
  shows "trmssstp (a ·sstp ∅) = trmssstp a ·set ∅"
⟨proof⟩

lemma trmssstp_subst':
  assumes "¬is_NegChecks a"
  shows "trmssstp (a ·sstp ∅) = trmssstp a ·set ∅"
⟨proof⟩

lemma trmssstp_subst'':
  fixes t::('a,'b) term" and δ::('a,'b) subst"
  assumes "t ∈ trmssstp (b ·sstp δ)"
  shows "∃ s ∈ trmssstp b. t = s · rm_vars (set (bvarssstp b)) δ"
⟨proof⟩

lemma trmssstp_subst''':
  fixes t::('a,'b) term" and δ ∅::('a,'b) subst"
  assumes "t ∈ trmssstp (b ·sstp δ) ·set ∅"
  shows "∃ s ∈ trmssstp b. t = s · rm_vars (set (bvarssstp b)) δ os ∅"
⟨proof⟩

lemma trmssst_subst:
  assumes "bvarssst S ∩ subst_domain ∅ = {}"
  shows "trmssst (S ·sst ∅) = trmssst S ·set ∅"
⟨proof⟩

lemma trmssst_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 ·_{set} δ)"
 shows "wf_{trms} (trms_{sstp} (a ·_{sstp} δ))"
 ⟨proof⟩

lemma trms_{sst}_fv_vars_{sst}_subset: "t ∈ trms_{sst} A ⇒ fv t ⊆ vars_{sst} A"
 ⟨proof⟩

lemma trms_{sst}_fv_subst_subset:
 assumes "t ∈ trms_{sst} S" "subst_domain ϑ ∩ bvars_{sst} S = {}"
 shows "fv (t · ϑ) ⊆ vars_{sst} (S ·_{sst} ϑ)"
 ⟨proof⟩

lemma trms_{sst}_fv_subst_subset':
 assumes "t ∈ subterms_{set} (trms_{sst} S)" "fv t ∩ bvars_{sst} S = {}" "fv (t · ϑ) ∩ bvars_{sst} S = {}"
 shows "fv (t · ϑ) ⊆ fv_{sst} (S ·_{sst} ϑ)"
 ⟨proof⟩

lemma trms_{sstp}_funs_term_cases:
 assumes "t ∈ trms_{sstp} (s ·_{sstp} ϑ)" "f ∈ funs_term t"
 shows "(∃ u ∈ trms_{sstp} s. f ∈ funs_term u) ∨ (∃ x ∈ fv_{sstp} s. f ∈ funs_term (ϑ x))"
 ⟨proof⟩

lemma trms_{sst}_funs_term_cases:
 assumes "t ∈ trms_{sst} (S ·_{sst} ϑ)" "f ∈ funs_term t"
 shows "(∃ u ∈ trms_{sst} S. f ∈ funs_term u) ∨ (∃ x ∈ fv_{sst} S. f ∈ funs_term (ϑ x))"
 ⟨proof⟩

lemma fv_{sst}_is_subterm_trms_{sst}_subst:
 assumes "x ∈ fv_{sst} T"
 and "bvars_{sst} T ∩ subst_domain ϑ = {}"
 shows "ϑ x ∈ subterms_{set} (trms_{sst} (T ·_{sst} ϑ))"
 ⟨proof⟩

lemma fv_{sst}_subst_fv_subset:
 assumes "x ∈ fv_{sst} S" "x ∉ bvars_{sst} S" "fv (ϑ x) ∩ bvars_{sst} S = {}"
 shows "fv (ϑ x) ⊆ fv_{sst} (S ·_{sst} ϑ)"
 ⟨proof⟩

lemma (in intruder_model) wf_{trms}_trms_{sst}_subst:
 assumes "wf_{trms} (trms_{sst} A ·_{set} δ)"
 shows "wf_{trms} (trms_{sst} (A ·_{sst} δ))"
 ⟨proof⟩

lemma fv_{sst}_subst_obtain_var:
 assumes "x ∈ fv_{sst} (S ·_{sst} δ)"
 shows "∃ y ∈ fv_{sst} S. x ∈ fv (δ y)"
 ⟨proof⟩

lemma fv_{sst}_subst_subset_range_vars_if_subset_domain:
 assumes "fv_{sst} S ⊆ subst_domain σ"
 shows "fv_{sst} (S ·_{sst} σ) ⊆ range_vars σ"
 ⟨proof⟩

lemma fv_{sst}_in_fv_trms_{sst}: "x ∈ fv_{sst} S ⇒ x ∈ fv_{set} (trms_{sst} S)"
 ⟨proof⟩

lemma fv_{sstp}_ground_subst_compose:
 assumes "subst_domain δ = subst_domain σ"
 and "range_vars δ = {}" "range_vars σ = {}"
 shows "fv_{sstp} (a ·_{sstp} δ ∘_s ϑ) = fv_{sstp} (a ·_{sstp} σ ∘_s ϑ)"
 ⟨proof⟩

```

lemma fv_sst_ground_subst_compose:
  assumes "subst_domain  $\delta$  = subst_domain  $\sigma$ "
    and "range_vars  $\delta$  = {}" "range_vars  $\sigma$  = {}"
  shows "fv_sst (S .sst  $\delta$   $\circ_s$   $\vartheta$ ) = fv_sst (S .sst  $\sigma$   $\circ_s$   $\vartheta$ )"
<proof>

lemma stateful_strand_step_subst_comp:
  assumes "range_vars  $\delta \cap \text{set} (\text{bvars}_{sst} x) = \{\}$ "
  shows " $x$  .sstp  $\delta$   $\circ_s$   $\vartheta = (x$  .sstp  $\delta)$  .sstp  $\vartheta$ "
<proof>

lemma stateful_strand_subst_comp:
  assumes "range_vars  $\delta \cap \text{bvars}_{sst} S = \{\}$ "
  shows " $S$  .sst  $\delta$   $\circ_s$   $\vartheta = (S$  .sst  $\delta)$  .sst  $\vartheta$ "
<proof>

lemma subst_apply_bvars_disj_NegChecks:
  assumes "set X  $\cap$  subst_domain  $\vartheta = \{\}$ "
  shows "NegChecks X F G .sstp  $\vartheta = \text{NegChecks X (F .pairs } \vartheta) (G .pairs } \vartheta)$ "
<proof>

lemma subst_apply_NegChecks_no_bvars[simp]:
  " $\forall [] \langle \forall \neq: F \vee \notin: F' \rangle$  .sstp  $\vartheta = \forall [] \langle \forall \neq: (F$  .pairs  $\vartheta) \vee \notin: (F'$  .pairs  $\vartheta) \rangle$ "
  " $\forall [] \langle \forall \neq: [] \vee \notin: F' \rangle$  .sstp  $\vartheta = \forall [] \langle \forall \neq: [] \vee \notin: (F'$  .pairs  $\vartheta) \rangle$ "
  " $\forall [] \langle \forall \neq: F \vee \notin: [] \rangle$  .sstp  $\vartheta = \forall [] \langle \forall \neq: (F$  .pairs  $\vartheta) \vee \notin: [] \rangle$ "
  " $\forall [] \langle \forall \neq: [] \vee \notin: [(t,s)] \rangle$  .sstp  $\vartheta = \forall [] \langle \forall \neq: [] \vee \notin: [(t \cdot \vartheta, s \cdot \vartheta)] \rangle$ "
  " $\forall [] \langle \forall \neq: [(t,s)] \vee \notin: [] \rangle$  .sstp  $\vartheta = \forall [] \langle \forall \neq: [(t \cdot \vartheta, s \cdot \vartheta)] \vee \notin: [] \rangle$ "
<proof>

lemma setops_sst_mono:
  "set M  $\subseteq$  set N  $\implies$  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:
  " $(t,s) \in \text{setops}_{sstp} x \iff$ 
  ( $x = \text{Insert } t \ s \vee x = \text{Delete } t \ s \vee (\exists ac. x = \text{InSet } ac \ t \ s) \vee$ 
  ( $\exists X \ F \ F'. x = \text{NegChecks } X \ F \ F' \wedge (t,s) \in \text{set } F')$ )"
<proof>

lemma setops_sst_member_iff:
  " $(t,s) \in \text{setops}_{sst} A \iff$ 
  ( $\text{Insert } t \ s \in \text{set } A \vee \text{Delete } t \ s \in \text{set } A \vee (\exists ac. \text{InSet } ac \ t \ s \in \text{set } A) \vee$ 
  ( $\exists X \ F \ F'. \text{NegChecks } X \ F \ F' \in \text{set } A \wedge (t,s) \in \text{set } F')$ )"
  (is "?P  $\iff$  ?Q")
<proof>

lemma setops_sstp_subst:
  assumes "set (bvars_sstp a)  $\cap$  subst_domain  $\vartheta = \{\}$ "
  shows "setops_sstp (a .sstp  $\vartheta) = \text{setops}_{sstp} a$  .pset  $\vartheta$ "
<proof>

```

```

lemma setopssstp_subst':
  assumes "¬is_NegChecks a"
  shows "setopssstp (a ·sstp ∅) = setopssstp a ·pset ∅"
⟨proof⟩

lemma setopssstp_subst'':
  fixes t:: "('a,'b) term × ('a,'b) term" and δ:: "('a,'b) subst"
  assumes t: "t ∈ setopssstp (b ·sstp δ)"
  shows "∃ s ∈ setopssstp b. t = s ·p rm_vars (set (bvarssstp b)) δ"
⟨proof⟩

lemma setopssst_subst:
  assumes "bvarssst S ∩ subst_domain ∅ = {}"
  shows "setopssst (S ·sst ∅) = setopssst S ·pset ∅"
⟨proof⟩

lemma setopssst_subst':
  fixes p:: "('a,'b) term × ('a,'b) term" and δ:: "('a,'b) subst"
  assumes "p ∈ setopssst (S ·sst δ)"
  shows "∃ s ∈ setopssst S. ∃ X. set X ⊆ bvarssst S ∧ p = s ·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) →
    (∃ (t,s) ∈ set F. t · δ ◦s I ≠ s · δ ◦s I) ∨
    (∃ (t,s) ∈ set G. (t,s) ·p δ ◦s I ∉ D))"

```

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 ts#S]]s = (λI. (∀ t ∈ set ts. M ⊢ t · I) ∧ [[M; D; S]]s I)"
| "[M; D; Receive ts#S]]s = (λI. [(set ts ·set I) ∪ M; D; S]]s I)"
| "[M; D; Equality _ t t'#S]]s = (λI. t · I = t' · I ∧ [[M; D; S]]s I)"
| "[M; D; Insert t s#S]]s = (λI. [[M; insert ((t,s) ·p I) D; S]]s I)"
| "[M; D; Delete t s#S]]s = (λI. [[M; D - {(t,s) ·p I}; S]]s I)"
| "[M; D; InSet _ t s#S]]s = (λI. (t,s) ·p I ∈ D ∧ [[M; D; S]]s I)"
| "[M; D; NegChecks X F F'#S]]s = (λI. negchecks_model I D X F F' ∧ [[M; D; S]]s I)"

```

lemmas strand_sem_stateful_induct =

```

strand_sem_stateful.induct[case_names Nil ConsSnd ConsRcv ConsEq
  ConsIns ConsDel ConsIn ConsNegChecks]

```

abbreviation constr_sem_stateful (infix <|=_s> 91) where "I |=_s A ≡ [[{}; {}; A]]_s I"

lemma stateful_strand_sem_NegChecks_no_bvars:

```

"[[M; D; [⟨t not in s⟩]]]s I ↔ (t · I, s · I) ∉ D"
"[[M; D; [⟨t != s⟩]]]s I ↔ t · I ≠ s · I"
⟨proof⟩

```

lemma strand_sem_ik_mono_stateful:

```

"[[M; D; A]]s I ⇒ [[M ∪ M'; D; A]]s I"
⟨proof⟩

```

lemma strand_sem_append_stateful:

" $\llbracket M; D; A @ B \rrbracket_s \mathcal{I} \longleftrightarrow \llbracket M; D; A \rrbracket_s \mathcal{I} \wedge \llbracket M \cup (ik_{sst} A \cdot_{set} \mathcal{I}); dbupd_{sst} A \mathcal{I} D; B \rrbracket_s \mathcal{I}$ "
 (is "?P \longleftrightarrow ?Q \wedge ?R")

<proof>

lemma negchecks_model_db_subset:

fixes $F F'::((\text{'a','b'} \text{ term} \times (\text{'a','b'} \text{ term}) \text{ list})$ "
 assumes " $D' \subseteq D$ "
 and "negchecks_model $\mathcal{I} D X F F'$ "
 shows "negchecks_model $\mathcal{I} D' X F F'$ "

<proof>

lemma negchecks_model_db_supset:

fixes $F F'::((\text{'a','b'} \text{ term} \times (\text{'a','b'} \text{ term}) \text{ list})$ "
 assumes " $D' \subseteq D$ "
 and " $\forall f \in \text{set } F'. \forall \delta. \text{subst_domain } \delta = \text{set } X \wedge \text{ground } (\text{subst_range } \delta) \longrightarrow f \cdot_p (\delta \circ_s \mathcal{I}) \notin D - D'$ "
 and "negchecks_model $\mathcal{I} D' X F F'$ "
 shows "negchecks_model $\mathcal{I} D X F F'$ "

<proof>

lemma negchecks_model_subst:

fixes $F F'::((\text{'a','b'} \text{ term} \times (\text{'a','b'} \text{ term}) \text{ list})$ "
 assumes " $(\text{subst_domain } \delta \cup \text{range_vars } \delta) \cap \text{set } X = \{\}$ "
 shows "negchecks_model $(\delta \circ_s \vartheta) D X F F' \longleftrightarrow \text{negchecks_model } \vartheta D X (F \cdot_{pairs} \delta) (F' \cdot_{pairs} \delta)$ "

<proof>

lemma strand_sem_subst_stateful:

fixes $\delta::(\text{'fun','var'} \text{ subst})$ "
 assumes " $(\text{subst_domain } \delta \cup \text{range_vars } \delta) \cap \text{bvars}_{sst} S = \{\}$ "
 shows " $\llbracket M; D; S \rrbracket_s (\delta \circ_s \vartheta) \longleftrightarrow \llbracket M; D; S \cdot_{sst} \delta \rrbracket_s \vartheta$ "

<proof>

lemma strand_sem_receive_prepend_stateful:

assumes " $\llbracket M; D; S \rrbracket_s \vartheta$ "
 and "list_all is_Receive S' "
 shows " $\llbracket M; D; S @ S' \rrbracket_s \vartheta$ "

<proof>

lemma negchecks_model_model_swap:

fixes $I J::(\text{'a','b'} \text{ subst})$ "
 assumes " $\forall x \in (\text{fv}_{pairs} F \cup \text{fv}_{pairs} G) - \text{set } X. I x = J x$ "
 and "negchecks_model $I D X F G$ "
 shows "negchecks_model $J D X F G$ "

<proof>

lemma strand_sem_model_swap:

assumes " $\forall x \in \text{fv}_{sst} S. I x = J x$ "
 and " $\llbracket M; D; S \rrbracket_s I$ "
 shows " $\llbracket M; D; S \rrbracket_s J$ "

<proof>

lemma strand_sem_receive_send_append:

assumes $A: \llbracket M; D; A \rrbracket_s I$ "
 shows " $\llbracket M; D; A @ [\text{receive}\langle [t] \rangle, \text{send}\langle [t] \rangle] \rrbracket_s I$ "

<proof>

lemma strand_sem_stateful_if_no_send_or_check:

assumes $A: \text{list_all } (\lambda a. \neg \text{is_Send } a \wedge \neg \text{is_Check_or_Assignment } a) A$ "
 shows " $\llbracket M; D; A \rrbracket_s I$ "

<proof>

```

lemma strand_sem_stateful_if_sends_deduct:
  assumes "list_all is_Send A"
    and "∀ ts. send(ts) ∈ set A ⟶ (∀ t ∈ set ts. M ⊢ t · I)"
  shows "[M; D; A]s I"
⟨proof⟩

lemma strand_sem_stateful_if_checks:
  assumes "list_all is_Check_or_Assignment A"
    and "∀ ac t s. ⟨ac: t ≐ s⟩ ∈ set A ⟶ t · I = s · I"
    and "∀ ac t s. ⟨ac: t ∈ s⟩ ∈ set A ⟶ (t · I, s · I) ∈ D"
    and "∀ X F G. ∀ X (∀ ≠: F ∨ ∉: G) ∈ set A ⟶ negchecks_model I D X F G"
  shows "[M; D; A]s I"
⟨proof⟩

lemma strand_sem_stateful_sends_deduct:
  assumes A: "[M; D; A]s I"
    and ts: "send(ts) ∈ set A"
    and t: "t ∈ set ts"
  shows "M ∪ (iksst A ·set I) ⊢ t · I"
⟨proof⟩

end

```

4.1.4 Well-Formedness Lemmata

```

lemma wfvarsoccssst_subset_wfrestrictedvarssst[simp]:
  "wfvarsoccssst S ⊆ wfrestrictedvarssst S"
⟨proof⟩

lemma wfvarsoccssst_append: "wfvarsoccssst (S@S') = wfvarsoccssst S ∪ wfvarsoccssst S'"
⟨proof⟩

lemma wfrestrictedvarssst_union[simp]:
  "wfrestrictedvarssst (S@T) = wfrestrictedvarssst S ∪ wfrestrictedvarssst T"
⟨proof⟩

lemma wfrestrictedvarssst_singleton:
  "wfrestrictedvarssst [s] = wfrestrictedvarssstp s"
⟨proof⟩

lemma iksst_fv_subset_wfrestrictedvarssst[simp]:
  "fvset (iksst S) ⊆ wfrestrictedvarssst S"
⟨proof⟩

lemma wfsst_prefix[dest]: "wf'sst V (S@S') ⟹ wf'sst V S"
⟨proof⟩

lemma wfsst_vars_mono: "wf'sst V S ⟹ wf'sst (V ∪ W) S"
⟨proof⟩

lemma wfsstI[intro]: "wfrestrictedvarssst S ⊆ V ⟹ wf'sst V S"
⟨proof⟩

lemma wfsstI'[intro]:
  assumes "⋃ ((λx. case x of
    Receive ts ⇒ fvset (set ts)
  | Equality Assign _ t' ⇒ fv t'
  | Insert t t' ⇒ fv t ∪ fv t'
  | _ ⇒ {}) ` set S) ⊆ V"
  shows "wf'sst V S"
⟨proof⟩

lemma wfsst_append_exec: "wf'sst V (S@S') ⟹ wf'sst (V ∪ wfvarsoccssst S) S'"

```

<proof>

lemma wf_{sst_append} :

" $wf'_{sst} X S \implies wf'_{sst} Y T \implies wf'_{sst} (X \cup Y) (S@T)$ "

<proof>

lemma $wf_{sst_append_suffix}$:

" $wf'_{sst} V S \implies wf_{restrictedvars_{sst}} S' \subseteq wf_{restrictedvars_{sst}} S \cup V \implies wf'_{sst} V (S@S')$ "

<proof>

lemma $wf_{sst_append_suffix'}$:

assumes " $wf'_{sst} V S$ "

and " $\bigcup ((\lambda x. \text{case } x \text{ of}$
 Receive $ts \Rightarrow fv_{set} (\text{set } ts)$
 | Equality Assign $_ t' \Rightarrow fv t'$
 | Insert $t t' \Rightarrow fv t \cup fv t'$
 | $_ \Rightarrow \{\}$) ` set $S')$ $\subseteq wf_{varsoccs_{sst}} S \cup V$ "

shows " $wf'_{sst} V (S@S')$ "

<proof>

lemma $wf_{sst_subst_apply}$:

" $wf'_{sst} V S \implies wf'_{sst} (fv_{set} (\delta \setminus V)) (S \cdot_{sst} \delta)$ "

<proof>

lemma $wf_{sst_induct}[consumes 1,$

 case_names Nil ConsSnd ConsRcv ConsEq ConsEq2 ConsIn ConsIns ConsDel

 ConsNegChecks]:

fixes $S::('a, 'b) \text{stateful_strand}$ "

assumes " $wf'_{sst} V S$ "

" $P []$ "

" $\bigwedge ts S. \llbracket wf'_{sst} V S; P S \rrbracket \implies P (S@[Send ts])$ "

" $\bigwedge ts S. \llbracket wf'_{sst} V S; P S; fv_{set} (\text{set } ts) \subseteq V \cup wf_{varsoccs_{sst}} S \rrbracket \implies P (S@[Receive ts])$ "

" $\bigwedge t t' S. \llbracket wf'_{sst} V S; P S; fv t' \subseteq V \cup wf_{varsoccs_{sst}} S \rrbracket \implies P (S@[Equality Assign t t'])$ "

" $\bigwedge t t' S. \llbracket wf'_{sst} V S; P S \rrbracket \implies P (S@[Equality Check t t'])$ "

" $\bigwedge ac t t' S. \llbracket wf'_{sst} V S; P S \rrbracket \implies P (S@[InSet ac t t'])$ "

" $\bigwedge t t' S. \llbracket wf'_{sst} V S; P S; fv t \cup fv t' \subseteq V \cup wf_{varsoccs_{sst}} S \rrbracket \implies P (S@[Insert t t'])$ "

" $\bigwedge t t' S. \llbracket wf'_{sst} V S; P S \rrbracket \implies P (S@[Delete t t'])$ "

" $\bigwedge X F G S. \llbracket wf'_{sst} V S; P S \rrbracket \implies P (S@[NegChecks X F G])$ "

shows " $P S$ "

<proof>

lemma $wf_{sst_strand_first_Send_var_split}$:

assumes " $wf'_{sst} \{\} S$ " " $\exists v \in wf_{restrictedvars_{sst}} S. t \cdot I \sqsubseteq I v$ "

shows " $\exists S_{pre} S_{suf}. \neg(\exists w \in wf_{restrictedvars_{sst}} S_{pre}. t \cdot I \sqsubseteq I w) \wedge$
 $(\exists ts. S = S_{pre}@send(ts)\#S_{suf} \wedge t \cdot I \sqsubseteq_{set} \text{set } ts \cdot_{set} I) \vee$
 $(\exists s u. S = S_{pre}@assign: s \dot{=} u)\#S_{suf} \wedge t \cdot I \sqsubseteq s \cdot I \wedge \neg(t \cdot I \sqsubseteq_{set} I \setminus fv u) \vee$
 $(\exists s u. S = S_{pre}@assign: s \in u)\#S_{suf} \wedge (t \cdot I \sqsubseteq s \cdot I \vee t \cdot I \sqsubseteq u \cdot I))$ "

(is " $\exists S_{pre} S_{suf}. ?P S_{pre} \wedge ?Q S S_{pre} S_{suf}$ ")

<proof>

lemma $wf_{sst_vars_mono}$: " $wf'_{sst} V S \implies V \subseteq W \implies wf'_{sst} W S$ "

<proof>

lemma $wf_{restrictedvars_{sst}_receives_only_eq}$:

assumes " $list_all \text{ is_Receive } S$ "

shows " $wf_{restrictedvars_{sst}} S = fv_{sst} S$ "

<proof>

lemma $wf_{varsoccs_{sst}_receives_only_empty}$:

assumes " $list_all \text{ is_Receive } S$ "

shows " $wf_{varsoccs_{sst}} S = \{\}$ "

<proof>

```

lemma wfsst_sends_only:
  assumes "list_all is_Send S"
  shows "wf'sst V S"
⟨proof⟩

lemma wfsst_sends_only_prepend:
  assumes "wf'sst V S"
  and "list_all is_Send S'"
  shows "wf'sst V (S'@S)"
⟨proof⟩

lemma wfsst_receives_only_fv_subset:
  assumes "wf'sst V S"
  and "list_all is_Receive S"
  shows "fvsst S ⊆ V"
⟨proof⟩

lemma wfsst_append_suffix'':
  assumes "wf'sst V S"
  and "wfrestrictedvarssst S' ⊆ wfvarsoccssst S ∪ V"
  shows "wf'sst V (S@S')"
⟨proof⟩

end

```

4.2 Extending the Typing Result to Stateful Constraints

```

theory Stateful_Typing
imports Typing_Result Stateful_Strands
begin

  Locale setup

  locale stateful_typed_model = typed_model arity public Ana  $\Gamma$ 
    for arity::"'fun  $\Rightarrow$  nat"
    and public::"'fun  $\Rightarrow$  bool"
    and Ana::"'(fun, 'var) term  $\Rightarrow$  (('fun, 'var) term list  $\times$  ('fun, 'var) term list)"
    and  $\Gamma$ ::"'(fun, 'var) term  $\Rightarrow$  ('fun, 'atom::finite) term_type"
  +
  fixes Pair::"'fun"
  assumes Pair_arity: "arity Pair = 2"
  and Ana_subst': " $\bigwedge f T \delta K M. \text{Ana } (\text{Fun } f T) = (K, M) \implies \text{Ana } (\text{Fun } f T \cdot \delta) = (K \cdot_{\text{list}} \delta, M \cdot_{\text{list}} \delta)"$ "
begin

  lemma Ana_invar_subst'[simp]: "Ana_invar_subst S"
  ⟨proof⟩

  definition pair where
    "pair d  $\equiv$  case d of (t, t')  $\Rightarrow$  Fun Pair [t, t']"

  fun trpairs::
    "'(('fun, 'var) term  $\times$  ('fun, 'var) term) list  $\Rightarrow$ 
      ('fun, 'var) dbstatelist  $\Rightarrow$ 
      (('fun, 'var) term  $\times$  ('fun, 'var) term) list list"
  where
    "trpairs [] D = [[]]"
  | "trpairs ((s, t)#F) D =
    concat (map ( $\lambda d. \text{map } ((\#) (\text{pair } (s, t), \text{pair } d)) (\text{tr}_{\text{pairs}} F D)) D)"$ "

```

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⟨ts⟩#A) D = map ((#) (send⟨ts⟩st)) (tr A D)"
| "tr (receive⟨ts⟩#A) D = map ((#) (receive⟨ts⟩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 Minor 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 ∅) = pair ` set F ·set ∅"
⟨proof⟩

```

lemma Ana_subst_subterms_cases:

```

fixes ∅::('fun,'var) subst"
assumes t: "t ⊆set M ·set ∅"
and s: "s ∈ set (snd (Ana t))"
shows "(∃u ∈ subtermsset M. t = u · ∅ ∧ s ∈ set (snd (Ana u)) ·set ∅) ∨ (∃x ∈ fvset M. t ⊆ ∅ x)"
⟨proof⟩

```

lemma Ana_snd_subst_nth_inv:

```

fixes ∅::('fun,'var) subst" and f ts
defines "R ≡ snd (Ana (Fun f ts · ∅))"
assumes r: "r = R ! i" "i < length R"
shows "r = snd (Ana (Fun f ts)) ! i · ∅"
⟨proof⟩

```

lemma Ana_snd_subst_inv:

```

fixes ∅::('fun,'var) subst"
assumes r: "r ∈ set (snd (Ana (Fun f ts · ∅)))"
shows "∃t ∈ set (snd (Ana (Fun f ts))). r = t · ∅"
⟨proof⟩

```

lemma fun_pair_eq[dest]: "pair d = pair d' ⇒ d = d'"

⟨proof⟩

4 The Typing Result for Stateful Protocols

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 ·_p δ = d' ·_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⟨ts⟩#S) = pair ` setops_{sst} S"
 "pair ` setops_{sst} (receive⟨ts⟩#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 ·_{sst} ϑ) = pair ` setops_{sstp} (x ·_{sstp} ϑ) ∪ pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (send⟨ts⟩#S ·_{sst} ϑ) = pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (receive⟨ts⟩#S ·_{sst} ϑ) = pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (⟨ac: t ≐ t'⟩#S ·_{sst} ϑ) = pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (insert⟨t,s⟩#S ·_{sst} ϑ) = {pair (t,s) · ϑ} ∪ pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (delete⟨t,s⟩#S ·_{sst} ϑ) = {pair (t,s) · ϑ} ∪ pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (⟨ac: t ∈ s⟩#S ·_{sst} ϑ) = {pair (t,s) · ϑ} ∪ pair ` setops_{sst} (S ·_{sst} ϑ)"
 "pair ` setops_{sst} (∀X(∀≠: F ∨≠: G)#S ·_{sst} ϑ) =
 pair ` set (G ·_{pairs} rm_vars (set X) ϑ) ∪ pair ` setops_{sst} (S ·_{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 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)) ≐ (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 setopssst_wt_subst_ex:
  assumes  $\vartheta$ : "wtsubst  $\vartheta$ " "wftrms (subst_range  $\vartheta$ )"
    and t: "t  $\in$  pair  $\setminus$  setopssst (S  $\cdot$ sst  $\vartheta$ )"
  shows  $\exists s \delta. s \in$  pair  $\setminus$  setopssst S  $\wedge$  wtsubst  $\delta \wedge$  wftrms (subst_range  $\delta$ )  $\wedge$  t = s  $\cdot$   $\delta$ "
<proof>

lemma setopssst_wftrms:
  "wftrms (trmssst A)  $\implies$  wftrms (pair  $\setminus$  setopssst A)"
  "wftrms (trmssst A)  $\implies$  wftrms (trmssst A  $\cup$  pair  $\setminus$  setopssst A)"
<proof>

lemma SMP_MP_split:
  assumes "t  $\in$  SMP M"
    and M: " $\forall m \in M. is\_Fun\ m$ "
  shows "( $\exists \delta. wt_{subst}\ \delta \wedge wf_{trms}\ (subst\_range\ \delta) \wedge t \in M \cdot_{set}\ \delta$ )  $\vee$ 
    t  $\in$  SMP ((subtermsset M  $\cup$   $\bigcup$  ((set  $\circ$  fst  $\circ$  Ana)  $\setminus$  M)) - M)"
    (is "?P t  $\vee$  ?Q t")
<proof>

lemma setops_subterm_trms:
  assumes t: "t  $\in$  pair  $\setminus$  setopssst S"
    and s: "s  $\sqsubset$  t"
  shows "s  $\in$  subtermsset (trmssst S)"
<proof>

lemma setops_subterms_cases:
  assumes t: "t  $\in$  subtermsset (pair  $\setminus$  setopssst S)"
  shows "t  $\in$  subtermsset (trmssst S)  $\vee$  t  $\in$  pair  $\setminus$  setopssst S"
<proof>

lemma setops_SMP_cases:
  assumes "t  $\in$  SMP (pair  $\setminus$  setopssst S)"
    and " $\forall p. Ana\ (pair\ p) = ([], [])$ "
  shows "( $\exists \delta. wt_{subst}\ \delta \wedge wf_{trms}\ (subst\_range\ \delta) \wedge t \in$  pair  $\setminus$  setopssst S  $\cdot_{set}\ \delta$ )  $\vee$  t  $\in$  SMP (trmssst S)"
<proof>

lemma constraint_model_priv_const_in_constr_prefix:
  assumes A: "wfsst A"
    and I: "I  $\models_s$  A"
      "interpretationsubst I"
      "wftrms (subst_range I)"
      "wtsubst I"
    and c: "-public c"
      "arity c = 0"
      "Fun c []  $\sqsubseteq_{set}$  iksst A  $\cdot_{set}$  I"
  shows "Fun c []  $\sqsubseteq_{set}$  trmssst A"
<proof>

lemma trpairs_empty_case:
  assumes "trpairs F D = []"
  shows "D = []" "F  $\neq$  []"
<proof>

lemma trpairs_elem_length_eq:
  assumes "G  $\in$  set (trpairs F D)"
  shows "length G = length F"
<proof>

lemma trpairs_index:
  assumes "G  $\in$  set (trpairs F D)" "i < length F"
  shows " $\exists d \in$  set D. G ! i = (pair (F ! i), pair d)"

```

$\langle proof \rangle$

lemma tr_{pairs_cons} :

assumes " $G \in set (tr_{pairs} F D)$ " " $d \in set D$ "

shows " $(pair (s,t), pair d) \# G \in set (tr_{pairs} ((s,t) \# F) D)$ "

$\langle proof \rangle$

lemma $tr_{pairs_has_pair_lists}$:

assumes " $G \in set (tr_{pairs} F D)$ " " $g \in set G$ "

shows " $\exists f \in set F. \exists d \in set D. g = (pair f, pair d)$ "

$\langle proof \rangle$

lemma $tr_{pairs_is_pair_lists}$:

assumes " $f \in set F$ " " $d \in set D$ "

shows " $\exists G \in set (tr_{pairs} F D). (pair f, pair d) \in set G$ "
(is " $?P F D f d$ ")

$\langle proof \rangle$

lemma $tr_{pairs_db_append_subset}$:

" $set (tr_{pairs} F D) \subseteq set (tr_{pairs} F (D \# E))$ " (is ?A)

" $set (tr_{pairs} F E) \subseteq set (tr_{pairs} F (D \# E))$ " (is ?B)

$\langle proof \rangle$

lemma $tr_{pairs_trms_subset}$:

" $G \in set (tr_{pairs} F D) \implies trms_{pairs} G \subseteq pair \setminus set F \cup pair \setminus set D$ "

$\langle proof \rangle$

lemma $tr_{pairs_trms_subset'}$:

" $\bigcup (trms_{pairs} \setminus set (tr_{pairs} F D)) \subseteq pair \setminus set F \cup pair \setminus set D$ "

$\langle proof \rangle$

lemma $tr_{pairs_vars_subset}$:

" $G \in set (tr_{pairs} F D) \implies fv_{pairs} G \subseteq fv_{pairs} F \cup fv_{pairs} D$ "

$\langle proof \rangle$

lemma $tr_{pairs_vars_subset'}$: " $\bigcup (fv_{pairs} \setminus set (tr_{pairs} F D)) \subseteq fv_{pairs} F \cup fv_{pairs} D$ "

$\langle proof \rangle$

lemma tr_trms_subset :

" $A' \in set (tr A D) \implies trms_{st} A' \subseteq trms_{sst} A \cup pair \setminus setops_{sst} A \cup pair \setminus set D$ "

$\langle proof \rangle$

lemma tr_vars_subset :

assumes " $A' \in set (tr A D)$ "

shows " $fv_{st} A' \subseteq fv_{sst} A \cup (\bigcup (t, t') \in set D. fv t \cup fv t')$ " (is ?P)

and " $bvars_{st} A' \subseteq bvars_{sst} A$ " (is ?Q)

$\langle proof \rangle$

lemma tr_vars_disj :

assumes " $A' \in set (tr A D)$ " " $\forall (t, t') \in set D. (fv t \cup fv t') \cap bvars_{sst} A = \{\}$ "

and " $fv_{sst} A \cap bvars_{sst} A = \{\}$ "

shows " $fv_{st} A' \cap bvars_{st} A' = \{\}$ "

$\langle proof \rangle$

lemma $tfr_{sstp_alt_def}$:

"list_all $tfr_{sstp} S =$

$(\forall ac t t'. Equality ac t t' \in set S \wedge (\exists \delta. Unifier \delta t t') \longrightarrow \Gamma t = \Gamma t') \wedge$

$(\forall X F F'. NegChecks X F F' \in set S \longrightarrow ($

$(F = [] \wedge (\forall x \in fv_{pairs} F\text{-set } X. \exists a. \Gamma (Var x) = TAtom a)) \vee$

$(\forall f T. Fun f T \in subterms_{set} (trms_{pairs} F \cup pair \setminus set F') \longrightarrow$

$T = [] \vee (\exists s \in set T. s \notin Var \setminus set X))))$ "

```

(is "?P S = ?Q S")
⟨proof⟩

lemma tfrsst_Nil[simp]: "tfrsst []"
⟨proof⟩

lemma tfrsst_append: "tfrsst (A@B) ⇒ tfrsst A"
⟨proof⟩

lemma tfrsst_append': "tfrsst (A@B) ⇒ tfrsst B"
⟨proof⟩

lemma tfrsst_cons: "tfrsst (a#A) ⇒ tfrsst A"
⟨proof⟩

lemma tfrsstp_subst:
  assumes s: "tfrsstp s"
  and  $\vartheta$ : "wtsubst  $\vartheta$ " "wftrms (subst_range  $\vartheta$ )" "set (bvarssstp s) ∩ range_vars  $\vartheta$  = {}"
  shows "tfrsstp (s ·sstp  $\vartheta$ )"
⟨proof⟩

lemma tfrsstp_all_wt_subst_apply:
  assumes S: "list_all tfrsstp S"
  and  $\vartheta$ : "wtsubst  $\vartheta$ " "wftrms (subst_range  $\vartheta$ )" "bvarssst S ∩ range_vars  $\vartheta$  = {}"
  shows "list_all tfrsstp (S ·sst  $\vartheta$ )"
⟨proof⟩

lemma tfr_setops_if_tfr_trms:
  assumes "Pair ∉ ⋃ (funs_term ` SMP (trmssst S))"
  and "∀p. Ana (pair p) = ([], [])"
  and "∀s ∈ pair ` setopssst S. ∀t ∈ pair ` setopssst S. (∃δ. Unifier δ s t) ⇒ Γ s = Γ t"
  and "∀s ∈ pair ` setopssst S. ∀t ∈ pair ` setopssst S.
    (∃σ  $\vartheta$  ρ. wtsubst σ ∧ wtsubst  $\vartheta$  ∧ wftrms (subst_range σ) ∧ wftrms (subst_range  $\vartheta$ ) ∧
      Unifier ρ (s · σ) (t ·  $\vartheta$ ))
    ⇒ (∃δ. Unifier δ s t)"
  and tfr: "tfrset (trmssst S)"
  shows "tfrset (trmssst S ∪ pair ` setopssst S)"
⟨proof⟩

end

```

4.2.2 The Typing Result for Stateful Constraints

Correctness of the Constraint Reduction

```

context stateful_typed_model
begin

context
begin
private lemma tr_wf':
  assumes "∀(t,t') ∈ set D. (fv t ∪ fv t') ∩ bvarssst A = {}"
  and "∀(t,t') ∈ set D. fv t ∪ fv t' ⊆ X"
  and "wf'sst X A" "fvsst A ∩ bvarssst A = {}"
  and "A' ∈ set (tr A D)"
  shows "wfst X A'"
⟨proof⟩ lemma tr_wftrms:
  assumes "A' ∈ set (tr A [])" "wftrms (trmssst A)"
  shows "wftrms (trmsst A)"
⟨proof⟩

lemma tr_wf:
  assumes "A' ∈ set (tr A [])"

```

```

  and "wfsst A"
  and "wftrms (trmssst A)"
shows "wfst {} A'"
  and "wftrms (trmsst A)"
  and "fvst A' ∩ bvarsst A' = {}"
⟨proof⟩ lemma fun_pair_ineqs:
  assumes "d ·p δ ·p ∅ ≠ d' ·p I"
  shows "pair d · δ · ∅ ≠ 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') ∩ bvarssst A = {}"
  and "fvsst A ∩ bvarssst A = {}"
  and "ground M"
  and I: "interpretationsubst 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 "fvsst A ∩ bvarssst A = {}" and "interpretationsubst I"
  shows "I ⊨s A ⟷ (∃ A' ∈ set (tr A []). (I ⊨ ⟨A'⟩))"
⟨proof⟩

end

end

```

Typing Result Locale Definition

```
locale stateful_typing_result =
```

```

stateful_typed_model arity public Ana  $\Gamma$  Pair
+ typing_result arity public Ana  $\Gamma$ 
  for arity::"'fun  $\Rightarrow$  nat"
    and public::"'fun  $\Rightarrow$  bool"
    and Ana::"'(fun, 'var) term  $\Rightarrow$  (('fun, 'var) term list  $\times$  ('fun, 'var) term list)"
    and  $\Gamma$ ::"'(fun, 'var) term  $\Rightarrow$  ('fun, 'atom::finite) term_type"
    and Pair::"'fun"

```

Type-Flaw Resistance Preservation of the Constraint Reduction

```

context stateful_typing_result
begin

```

```

context
begin

```

```

private lemma tr_tfrsstp:
  assumes "A'  $\in$  set (tr A D)" "list_all tfrsstp A"
  and "fvsst A  $\cap$  bvarssst A = {}" (is "?P0 A D")
  and " $\forall (t, s) \in$  set D. (fv t  $\cup$  fv s)  $\cap$  bvarssst A = {}" (is "?P1 A D")
  and " $\forall t \in$  pair ` setopssst A  $\cup$  pair ` set D.  $\forall t' \in$  pair ` setopssst A  $\cup$  pair ` set D.
    ( $\exists \delta$ . Unifier  $\delta$  t t')  $\longrightarrow$   $\Gamma$  t =  $\Gamma$  t'" (is "?P3 A D")
  shows "list_all tfrstp A'"
<proof>

```

```

lemma tr_tfr:
  assumes "A'  $\in$  set (tr A [])" and "tfrsst A" and "fvsst A  $\cap$  bvarssst A = {}"
  shows "tfrst A'"
<proof>

```

```

end

```

```

end

```

Theorem: The Stateful Typing Result

```

context stateful_typing_result
begin

```

```

theorem stateful_typing_result:
  assumes "wfsst A"
  and "tfrsst A"
  and "wftrms (trmssst A)"
  and "interpretationsubst I"
  and "I  $\models_s$  A"
  obtains  $\mathcal{I}_\tau$ 
  where "interpretationsubst  $\mathcal{I}_\tau$ "
  and " $\mathcal{I}_\tau \models_s$  A"
  and "wtsubst  $\mathcal{I}_\tau$ "
  and "wftrms (subst_range  $\mathcal{I}_\tau$ )"
<proof>

```

```

end

```

4.2.3 Proving Type-Flaw Resistance Automatically

definition pair' where

```
"pair' pair_fun d  $\equiv$  case d of (t, t')  $\Rightarrow$  Fun pair_fun [t, t']"
```

fun comp_tfr_{sstp} where

```
"comp_tfrsstp  $\Gamma$  pair_fun ( $\langle \_ : t \doteq t' \rangle$ ) = (mgu t t'  $\neq$  None  $\longrightarrow$   $\Gamma$  t =  $\Gamma$  t')"
```

```
| "comp_tfrsstp  $\Gamma$  pair_fun ( $\forall X \langle \forall \neq : F \vee \notin : F' \rangle$ ) = (
  (F' = []  $\wedge$  ( $\forall x \in$  fvpairs F - set X. is_Var ( $\Gamma$  (Var x))))  $\vee$ 
```

4 The Typing Result for Stateful Protocols

```

  (∀ u ∈ subtermsset (trmspairs F ∪ pair' pair_fun ` set F').
    is_Fun u → (args u = [] ∨ (∃ s ∈ set (args u). s ∉ Var ` set X)))"
| "comp_tfrsstp _ _ _ = True"

definition comp_tfrsst where
  "comp_tfrsst arity Ana Γ pair_fun M S ≡
    list_all (comp_tfrsstp Γ pair_fun) S ∧
    list_all (wftrm' arity) (trmslistsst S) ∧
    has_all_wt_instances_of Γ (trmssst S ∪ pair' pair_fun ` setopssst S) M ∧
    comp_tfrset 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⟩

end

locale stateful_typing_result' =
  stateful_typed_model' arity public Ana Γ Pair + stateful_typing_result 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"
begin

lemma tfrsstp_is_comp_tfrsstp: "tfrsstp a = comp_tfrsstp Γ Pair a"
⟨proof⟩

lemma tfrsst_if_comp_tfrsst:
  assumes "comp_tfrsst arity Ana Γ Pair M S"
  shows "tfrsst S"
⟨proof⟩

lemma tfrsst_if_comp_tfrsst':
  fixes S
  defines "M ≡ SMP0 Ana Γ (trmslistsst S@map pair (setopslistsst S))"
  assumes comp_tfr: "comp_tfrsst arity Ana Γ Pair (set M) S"
  shows "tfrsst 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

```
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 has_LabelN where "has_LabelN n x ≡ fst x = ln n"
abbreviation has_LabelS where "has_LabelS x ≡ fst x = *"
```

```
definition unlabel where "unlabel S ≡ map snd S"
definition proj where "proj n S ≡ filter (λs. has_LabelN n s ∨ has_LabelS s) S"
abbreviation proj_unl where "proj_unl n S ≡ unlabel (proj n S)"
```

```
abbreviation wfrestrictedvarslst where "wfrestrictedvarslst S ≡ wfrestrictedvarsst (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 trmslst where "trmslst S ≡ trmsst (unlabel S)"
abbreviation trms_projlst where "trms_projlst n S ≡ trmsst (proj_unl n S)"
```

```
abbreviation varslst where "varslst S ≡ varsst (unlabel S)"
abbreviation vars_projlst where "vars_projlst n S ≡ varsst (proj_unl n S)"
```

```
abbreviation bvarslst where "bvarslst S ≡ bvarsst (unlabel S)"
abbreviation fvlst where "fvlst S ≡ fvst (unlabel S)"
```

```
abbreviation wflst where "wflst V S ≡ wfst V (unlabel S)"
```

5.1.2 Lemmata: Projections

```
lemma has_LabelS_proj_iff_not_has_LabelN:
  "list_all has_LabelS (proj l A) ↔ ¬list_ex (has_LabelN l) A"
<proof>
```

```
lemma proj_subset_if_no_label:
```

```

assumes "¬list_ex (has_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_ident:
assumes "list_all (λs. has_LabelN l s ∨ has_LabelS s) S"
shows "proj l S = S"
⟨proof⟩

lemma proj_elims_label:
assumes "k ≠ l"
shows "¬list_ex (has_LabelN l) (proj k S)"
⟨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"

```

" $l = (\ln n) \implies \text{proj } n ((l,a)\#A) = (l,a)\#\text{proj } n A$ "
" $l = \star \implies \text{proj } n ((l,a)\#A) = (l,a)\#\text{proj } n A$ "
" $\text{fst } b \neq \star \implies \text{fst } b \neq (\ln n) \implies \text{proj } n (b\#A) = \text{proj } n A$ "
⟨proof⟩

lemma `proj_append[simp]`:
"`proj 1 (A'@B')` = `proj 1 A'@proj 1 B'`"
"`proj_unl 1 (A@B)` = `proj_unl 1 A@proj_unl 1 B`"
⟨proof⟩

lemma `proj_unl_cons[simp]`:
"`proj_unl 1 ((ln l, a)\#A)` = `a#\proj_unl 1 A`"
" $l \neq l' \implies \text{proj_unl } l' ((\ln l, a)\#A) = \text{proj_unl } l' A$ "
"`proj_unl 1 ((\star, a)\#A)` = `a#\proj_unl 1 A`"
⟨proof⟩

lemma `trms_unlabel_proj[simp]`:
"`trmsstp (snd (\ln l, x))` \subseteq `trmsprojlst 1 [(\ln l, x)]`"
⟨proof⟩

lemma `trms_unlabel_star[simp]`:
"`trmsstp (snd (\star, x))` \subseteq `trmsprojlst 1 [(\star, x)]`"
⟨proof⟩

lemma `trmslst_union[simp]`: "`trmslst A` = $(\bigcup l. \text{trms_{projlst} } 1 A)$ "
⟨proof⟩

lemma `trmslst_append[simp]`: "`trmslst (A@B)` = `trmslst A` \cup `trmslst B`"
⟨proof⟩

lemma `trmsprojlst_append[simp]`: "`trmsprojlst 1 (A@B)` = `trmsprojlst 1 A` \cup `trmsprojlst 1 B`"
⟨proof⟩

lemma `trmsprojlst_subset[simp]`:
"`trmsprojlst 1 A` \subseteq `trmsprojlst 1 (A@B)`"
"`trmsprojlst 1 B` \subseteq `trmsprojlst 1 (A@B)`"
⟨proof⟩

lemma `trmslst_subset[simp]`:
"`trmslst A` \subseteq `trmslst (A@B)`"
"`trmslst B` \subseteq `trmslst (A@B)`"
⟨proof⟩

lemma `varslst_union`: "`varslst A` = $(\bigcup l. \text{vars_{projlst} } 1 A)$ "
⟨proof⟩

lemma `unlabel_Cons_inv`:
"`unlabel A = b\#B` $\implies \exists A'. (\exists n. A = (\ln n, b)\#A') \vee A = (\star, b)\#A'$ "
⟨proof⟩

lemma `unlabel_snoc_inv`:
"`unlabel A = B@[b]` $\implies \exists A'. (\exists n. A = A'@[(\ln n, b)]) \vee A = A'@[(\star, b)]$ "
⟨proof⟩

lemma `proj_idem[simp]`: "`proj 1 (proj 1 A)` = `proj 1 A`"
⟨proof⟩

lemma `proj_ikst_is_proj_rcv_set`:
"`ikst (proj_unl n A)` =
 $\{t. \exists ts. ((\ln n, \text{Receive } ts) \in \text{set } A \vee (\star, \text{Receive } ts) \in \text{set } A) \wedge t \in \text{set } ts\}$ "
⟨proof⟩

lemma `unlabel_ikst_is_rcv_set`:

```
"ikst (unlabel A) = {t | ∃ ts. (l, Receive ts) ∈ set A ∧ t ∈ set ts}"
⟨proof⟩
```

```
lemma proj_ik_union_is_unlabel_ik:
  "ikst (unlabel A) = (⋃ l. ikst (proj_unl l A))"
⟨proof⟩
```

```
lemma proj_ik_append[simp]:
  "ikst (proj_unl l (A@B)) = ikst (proj_unl l A) ∪ ikst (proj_unl l B)"
⟨proof⟩
```

```
lemma proj_ik_append_subst_all:
  "ikst (proj_unl l (A@B)) ·set I = (ikst (proj_unl l A) ·set I) ∪ (ikst (proj_unl l B) ·set I)"
⟨proof⟩
```

```
lemma ik_proj_subset[simp]: "ikst (proj_unl n A) ⊆ trms_projlst n A"
⟨proof⟩
```

```
lemma prefix_unlabel:
  "prefix A B ⇒ prefix (unlabel A) (unlabel B)"
⟨proof⟩
```

```
lemma prefix_proj:
  "prefix A B ⇒ prefix (proj n A) (proj n B)"
  "prefix A B ⇒ prefix (proj_unl n A) (proj_unl n B)"
⟨proof⟩
```

```
lemma suffix_unlabel:
  "suffix A B ⇒ suffix (unlabel A) (unlabel B)"
⟨proof⟩
```

```
lemma suffix_proj:
  "suffix A B ⇒ suffix (proj n A) (proj n B)"
  "suffix A B ⇒ suffix (proj_unl n A) (proj_unl n B)"
⟨proof⟩
```

5.1.3 Lemmata: Well-formedness

```
lemma wfvarsoccsst_proj_union:
  "wfvarsoccsst (unlabel A) = (⋃ l. wfvarsoccsst (proj_unl l A))"
⟨proof⟩
```

```
lemma wf_if_wf_proj:
  assumes "∀ l. wfst V (proj_unl l A)"
  shows "wfst V (unlabel A)"
⟨proof⟩
```

end

5.2 Parallel Compositionality of Security Protocols

```
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  $\Gamma::('fun, 'var) term \Rightarrow ('fun, 'atom::finite) term\_type"$ 
+
fixes label_witness1 and label_witness2::"lbl"
assumes at_least_2_labels: "label_witness1  $\neq$  label_witness2"
begin

```

The Ground Sub-Message Patterns (GSMP)

```

definition GSMP::('fun, 'var) terms  $\Rightarrow$  ('fun, 'var) terms" where
"GSMP P  $\equiv$  {t  $\in$  SMP P. fv t = {}}"

```

```

definition typing_cond where
"typing_cond  $\mathcal{A} \equiv$ 
wfst {}  $\mathcal{A} \wedge$ 
fvst  $\mathcal{A} \cap$  bvarsst  $\mathcal{A} = \{\} \wedge$ 
tfrst  $\mathcal{A} \wedge$ 
wftrms (trmsst  $\mathcal{A}) \wedge$ 
Ana_invar_subst (ikst  $\mathcal{A} \cup$  assignment_rhsst  $\mathcal{A})"$ 

```

5.2.2 Definitions: GSMP Disjointness and Parallel Composability

```

definition GSMP_disjoint where
"GSMP_disjoint P1 P2 Secrets  $\equiv$  GSMP P1  $\cap$  GSMP P2  $\subseteq$  Secrets  $\cup$  {m. {}  $\vdash_c$  m}"

```

```

definition declassifiedlst where
"declassifiedlst ( $\mathcal{A}::('fun, 'var, 'lbl)$  labeled_strand)  $\mathcal{I} \equiv$ 
{s.  $\bigcup$  {set ts | ts. ( $\star$ , Receive ts)  $\in$  set ( $\mathcal{A} \cdot_{lst}$   $\mathcal{I}$ )}  $\vdash_s$  s}"

```

```

definition par_comp where
"par_comp ( $\mathcal{A}::('fun, 'var, 'lbl)$  labeled_strand) (Secrets::('fun, 'var) terms)  $\equiv$ 
( $\forall$  l1 l2. l1  $\neq$  l2  $\longrightarrow$  GSMP_disjoint (trms_projlst l1  $\mathcal{A}$ ) (trms_projlst l2  $\mathcal{A}$ ) Secrets)  $\wedge$ 
( $\forall$  s  $\in$  Secrets.  $\neg$ { $\} \vdash_c$  s)  $\wedge$ 
ground Secrets"

```

```

definition strand_leakslst where
"strand_leakslst  $\mathcal{A}$  Sec  $\mathcal{I} \equiv$  ( $\exists$  t  $\in$  Sec - declassifiedlst  $\mathcal{A}$   $\mathcal{I}$ .  $\exists$  l. ( $\mathcal{I} \models$   $\langle$ proj_unl l  $\mathcal{A}$ @[Send1 t] $\rangle$ )"

```

5.2.3 Definitions: GSMP-Restricted Intruder Deduction Variant

```

definition intruder_deduct_hom::
"('fun, 'var) terms  $\Rightarrow$  ('fun, 'var, 'lbl) labeled_strand  $\Rightarrow$  ('fun, 'var) term  $\Rightarrow$  bool"
( $\langle$ _;_  $\rangle$   $\vdash_{GSMP}$  _  $>$  50)

```

```

where
" $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} t \equiv$   $\langle$ M;  $\lambda t. t \in$  GSMP (trmslst  $\mathcal{A}) \rangle \vdash_r t"$ 

```

```

lemma intruder_deduct_hom_AxiomH[simp]:
assumes "t  $\in$  M"
shows " $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} t"$ 
 $\langle$ proof $\rangle$ 

```

```

lemma intruder_deduct_hom_ComposeH[simp]:
assumes "length X = arity f" "public f" " $\bigwedge$ x. x  $\in$  set X  $\implies$   $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} x"$ 
and "Fun f X  $\in$  GSMP (trmslst  $\mathcal{A})"$ 
shows " $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP}$  Fun f X"
 $\langle$ proof $\rangle$ 

```

```

lemma intruder_deduct_hom_DecomposeH:
assumes " $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} t"$  "Ana t = (K, T)" " $\bigwedge$ k. k  $\in$  set K  $\implies$   $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} k"$  "ti  $\in$  set T"
shows " $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} t_i"$ 
 $\langle$ proof $\rangle$ 

```

```

lemma intruder_deduct_hom_induct[consumes 1, case_names AxiomH ComposeH DecomposeH]:
assumes " $\langle$ M;  $\mathcal{A}$  $\rangle \vdash_{GSMP} t"$  " $\bigwedge$ t. t  $\in$  M  $\implies$  P M t"
" $\bigwedge$ X f. [ $\text{length X = arity f; public f;}$ 

```

$$\begin{aligned} & \bigwedge x. x \in \text{set } X \implies \langle M; \mathcal{A} \rangle \vdash_{GSMP} x; \\ & \bigwedge x. x \in \text{set } X \implies P M x; \\ & \text{Fun } f X \in GSMP (\text{trms}_{lst} \mathcal{A}) \\ & \quad \Big] \implies P M (\text{Fun } f X)'' \\ & \text{"}\bigwedge t K T t_i. \llbracket \langle M; \mathcal{A} \rangle \vdash_{GSMP} t; P M t; \text{Ana } t = (K, T); \\ & \quad \bigwedge k. k \in \text{set } K \implies \langle M; \mathcal{A} \rangle \vdash_{GSMP} k; \\ & \quad \bigwedge k. k \in \text{set } K \implies P M k; t_i \in \text{set } T \rrbracket \implies P M t_i\text{"} \end{aligned}$$

shows "P M t"
 <proof>

lemma ideduct_hom_mono:
 "⟦⟨M; A⟩ ⊢_{GSMP} t; M ⊆ M'⟧ ⟹ ⟨M'; A⟩ ⊢_{GSMP} 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_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. {} ⊢_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 ∨ {} ⊢_c t"

<proof>

lemma *GSMP_wt_subst_subset*:
 assumes "t ∈ GSMP (M ·_{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>

5.2.5 Lemmata: Intruder Knowledge and Declassification

lemma *declassified_{lst}_alt_def*:
 "declassified_{lst} A I = {s. (⋃{set ts | ts. (★, Receive ts) ∈ set A}) ·_{set} I ⊢ s}"
<proof>

lemma *declassified_{lst}_star_receive_supset*:
 "{t | t ts. (★, Receive ts) ∈ set A ∧ t ∈ set ts} ·_{set} I ⊆ declassified_{lst} A I"
<proof>

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 *ik_proj_subst_subterms_GSMP_subset*:
 assumes I: "wt_{subst} I" "wf_{trms} (subst_range I)" "interpretation_{subst} I"
 shows "subterms_{set} (ik_{st} (proj_unl n A) ·_{set} I) ⊆ GSMP (trms_proj_{lst} n A)" (is "?A ⊆ ?B")
<proof>

lemma *declassified_proj_eq*: "declassified_{lst} A I = declassified_{lst} (proj n A) I"
<proof>

lemma *declassified_prefix_subset*:
 assumes AB: "prefix A B"
 shows "declassified_{lst} A I ⊆ declassified_{lst} B I"
<proof>

lemma *declassified_proj_ik_subset*:
 "declassified_{lst} A I ⊆ {s. ik_{st} (proj_unl n A) ·_{set} I ⊢ s}"
 (is "?A A ⊆ ?P A A")
<proof>

lemma *deduct_proj_priv_term_prefix_ex*:
 assumes A: "ik_{st} (proj_unl l A) ·_{set} I ⊢ t"
 and t: "¬{} ⊢_c t"
 shows "∃B k s. (k = ★ ∨ k = ln l) ∧ prefix (B@[k, receive(s)_{st}]) A ∧
 declassified_{lst} ((B@[k, receive(s)_{st}])) I = declassified_{lst} A I ∧
 ik_{st} (proj_unl l (B@[k, receive(s)_{st}])) = ik_{st} (proj_unl l A)"
<proof>

5.2.6 Lemmata: Homogeneous and Heterogeneous Terms (Deprecated Theory)

The following theory is no longer needed for the compositionality result

context

```

begin
private definition proj_specific where
  "proj_specific n t A Secrets  $\equiv$  t  $\in$  GSMP (trms_projlst n A) - (Secrets  $\cup$  {m. {}  $\vdash_c$  m})"

private definition heterogeneouslst where
  "heterogeneouslst t A Secrets  $\equiv$  (
    ( $\exists$  l1 l2.  $\exists$  s1  $\in$  subterms t.  $\exists$  s2  $\in$  subterms t.
      l1  $\neq$  l2  $\wedge$  proj_specific l1 s1 A Secrets  $\wedge$  proj_specific l2 s2 A Secrets))"

private abbreviation homogeneouslst where
  "homogeneouslst t A Secrets  $\equiv$   $\neg$ heterogeneouslst t A Secrets"

private definition intruder_deduct_hom'::
  "('fun,'var) terms  $\Rightarrow$  ('fun,'var,'lbl) labeled_strand  $\Rightarrow$  ('fun,'var) terms  $\Rightarrow$  ('fun,'var) term
   $\Rightarrow$  bool" (<(_;_i_)  $\vdash_{hom}$  _> 50)
where
  "<M; A; Sec>  $\vdash_{hom}$  t  $\equiv$  <M;  $\lambda$ t. homogeneouslst t A Sec  $\wedge$  t  $\in$  GSMP (trmslst A)>  $\vdash_r$  t"

private lemma GSMP_disjoint_fst_specific_not_snd_specific:
  assumes "GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec" "l  $\neq$  l'"
  and "proj_specific l m A Sec"
  shows " $\neg$ proj_specific l' m A Sec"
<proof> lemma GSMP_disjoint_snd_specific_not_fst_specific:
  assumes "GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and "proj_specific l' m A Sec"
  shows " $\neg$ proj_specific l m A Sec"
<proof> lemma GSMP_disjoint_intersection_not_specific:
  assumes "GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and "t  $\in$  Sec  $\vee$  {}  $\vdash_c$  t"
  shows " $\neg$ proj_specific l t A Sec" " $\neg$ proj_specific l' t A Sec"
<proof> lemma proj_specific_secrets_anti_mono:
  assumes "proj_specific l t A Sec" "Sec'  $\subseteq$  Sec"
  shows "proj_specific l t A Sec'"
<proof> lemma heterogeneous_secrets_anti_mono:
  assumes "heterogeneouslst t A Sec" "Sec'  $\subseteq$  Sec"
  shows "heterogeneouslst t A Sec'"
<proof> lemma homogeneous_secrets_mono:
  assumes "homogeneouslst t A Sec'" "Sec'  $\subseteq$  Sec"
  shows "homogeneouslst t A Sec"
<proof> lemma heterogeneous_supterm:
  assumes "heterogeneouslst t A Sec" "t  $\sqsubseteq$  t'"
  shows "heterogeneouslst t' A Sec"
<proof> lemma homogeneous_subterm:
  assumes "homogeneouslst t A Sec" "t'  $\sqsubseteq$  t"
  shows "homogeneouslst t' A Sec"
<proof> lemma proj_specific_subterm:
  assumes "t  $\sqsubseteq$  t'" "proj_specific l t' A Sec"
  shows "proj_specific l t A Sec  $\vee$  t  $\in$  Sec  $\vee$  {}  $\vdash_c$  t"
<proof> lemma heterogeneous_term_is_Fun:
  assumes "heterogeneouslst t A S" shows " $\exists$  f T. t = Fun f T"
<proof> lemma proj_specific_is_homogeneous:
  assumes A: " $\forall$  l l'. l  $\neq$  l'  $\longrightarrow$  GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and t: "proj_specific l m A Sec"
  shows "homogeneouslst m A Sec"
<proof> lemma deduct_synth_homogeneous:
  assumes "{}  $\vdash_c$  t"
  shows "homogeneouslst t A Sec"
<proof> lemma GSMP_proj_is_homogeneous:
  assumes " $\forall$  l l'. l  $\neq$  l'  $\longrightarrow$  GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and "t  $\in$  GSMP (trms_projlst l A)" "t  $\notin$  Sec"
  shows "homogeneouslst t A Sec"
<proof> lemma homogeneous_is_not_proj_specific:
  assumes "homogeneouslst m A Sec"

```

```

shows "∃l::'lbl. ¬proj_specific l m A Sec"
⟨proof⟩ lemma secrets_are_homogeneous:
  assumes "∀s ∈ Sec. P s → (∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Sec)" "s ∈ Sec" "P s"
  shows "homogeneouslst s A Sec"
⟨proof⟩ lemma GSMP_is_homogeneous:
  assumes A: "∀l l'. l ≠ l' → GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and t: "t ∈ GSMP (trmslst A)" "t ∉ Sec"
  shows "homogeneouslst t A Sec"
⟨proof⟩ lemma GSMP_intersection_is_homogeneous:
  assumes A: "∀l l'. l ≠ l' → GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and t: "t ∈ GSMP (trms_projlst l A) ∩ GSMP (trms_projlst l' A)" "l ≠ l'"
  shows "homogeneouslst t A Sec"
⟨proof⟩ lemma GSMP_is_homogeneous':
  assumes A: "∀l l'. l ≠ l' → GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and t: "t ∈ GSMP (trmslst A)"
  "t ∉ Sec - ∪{GSMP (trms_projlst l1 A) ∩ GSMP (trms_projlst l2 A) | l1 l2. l1 ≠ l2}"
  shows "homogeneouslst t A Sec"
⟨proof⟩ lemma Ana_keys_homogeneous:
  assumes A: "∀l l'. l ≠ l' → GSMP_disjoint (trms_projlst l A) (trms_projlst l' A) Sec"
  and t: "t ∈ GSMP (trmslst A)"
  and k: "Ana t = (K,T)" "k ∈ set K"
  "k ∉ Sec - ∪{GSMP (trms_projlst l1 A) ∩ GSMP (trms_projlst l2 A) | l1 l2. l1 ≠ l2}"
  shows "homogeneouslst k A Sec"
⟨proof⟩
end

```

5.2.7 Lemmata: Intruder Deduction Equivalences

```

lemma deduct_if_hom_deduct: "⟨M;A⟩ ⊢GSMP m ⇒ M ⊢ m"
⟨proof⟩

lemma hom_deduct_if_hom_ik:
  assumes "⟨M;A⟩ ⊢GSMP m" "∀m ∈ M. m ∈ GSMP (trmslst A)"
  shows "m ∈ GSMP (trmslst A)"
⟨proof⟩

lemma deduct_hom_if_synth:
  assumes hom: "m ∈ GSMP (trmslst A)"
  and m: "M ⊢c m"
  shows "⟨M; A⟩ ⊢GSMP m"
⟨proof⟩

lemma hom_deduct_if_deduct:
  assumes M: "∀m ∈ M. m ∈ GSMP (trmslst A)"
  and m: "M ⊢ m" "m ∈ GSMP (trmslst A)"
  shows "⟨M; A⟩ ⊢GSMP 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 l ⊆ GSMP (trms_projlst l A)"
  "∀l. Discl ⊆ {s. M l ⊢ s}"
  and Sec: "∀l. ∀s ∈ Sec - Discl. ¬(⟨M l; A⟩ ⊢GSMP s)"
  and t: "⟨∪l. M l; A⟩ ⊢GSMP t"
  shows "t ∉ Sec - Discl" (is ?A)
  "∀l. t ∈ GSMP (trms_projlst l A) → ⟨M l; A⟩ ⊢GSMP t" (is ?B)
⟨proof⟩

lemma par_comp_deduct_proj:
  assumes A: "par_comp A Sec"

```

```

and M: "∀l. M l ⊆ GSMP (trms_projlst l A)"
      "∀l. Discl ⊆ {s. M l ⊢ s}"
and t: "(∪l. M l) ⊢ t" "t ∈ GSMP (trms_projlst l A)"
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⟩" "interpretationsubst I" "wtsubst I" "wftrms (subst_range I)"
  shows "(∀l. (I ⊨ ⟨proj_unl l A⟩)) ∨
        (∃A' l' t. prefix A' A ∧ suffix [(l', receive(t)st]) A' ∧ (strand_leakslst A' Sec I))"
⟨proof⟩

```

end

```

locale labeled_typing =
  labeled_typed_model arity public Ana Γ label_witness1 label_witness2
+ typing_result 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"
  and label_witness1::"lbl"
  and label_witness2::"lbl"

```

begin

```

theorem par_comp_constr:
  assumes A: "par_comp A Sec" "typing_cond (unlabel A)"
  and I: "I ⊨ ⟨unlabel A⟩" "interpretationsubst I"
  shows "∃Iτ. interpretationsubst Iτ ∧ wtsubst Iτ ∧ wftrms (subst_range Iτ) ∧ (Iτ ⊨ ⟨unlabel A⟩) ∧
        ((∀l. (Iτ ⊨ ⟨proj_unl l A⟩)) ∨
         (∃A' l' t. prefix A' A ∧ suffix [(l', receive(t)st]) A' ∧
          (strand_leakslst A' Sec Iτ)))"
⟨proof⟩

```

end

5.2.10 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) terms"
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}}"
  and "δ ≡ var_rename (max_var_set (fvset A))"
  assumes A'_wf: "wftrms' arity A'"
  and B'_wf: "wftrms' arity B'"
  and A_inst: "has_all_wt_instances_of Γ A' A"
  and B_inst: "has_all_wt_instances_of Γ B' (B·set δ)"

```

```

and A_SMP_repr: "finite_SMP_representation arity Ana  $\Gamma$  A"
and B_SMP_repr: "finite_SMP_representation arity Ana  $\Gamma$  (B ·set  $\delta$ )"
and AB_trms_disj:
  " $\forall t \in A. \forall s \in B \cdot_{set} \delta. \Gamma t = \Gamma s \wedge \text{mgu } t s \neq \text{None} \longrightarrow$ 
  (intruder_synth' public arity {} t)  $\vee$  (( $\exists u \in \text{Sec. is\_wt\_instance\_of\_cond } \Gamma t u$ ))"
and Sec_wf: "wftrms Sec"
shows "GSMP_disjoint A' B' ((f Sec) - {m. {}  $\vdash_c$  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

```
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

```
definition 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 duallsstp :: "('a,'b,'c) labeled_stateful_strand_step ⇒ ('a,'b,'c) labeled_stateful_strand_step"
where
  "duallsstp (l,send(t)) = (l,receive(t))"
| "duallsstp (l,receive(t)) = (l,send(t))"
| "duallsstp x = x"
```

```
definition duallsst :: "('a,'b,'c) labeled_stateful_strand ⇒ ('a,'b,'c) labeled_stateful_strand"
where
```

```
"duallsst ≡ map duallsstp"
```

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 wfrestrictedvarslsst where "wfrestrictedvarslsst S ≡ wfrestrictedvarssst (unlabel S)"
```

```
abbreviation iklsst where "iklsst S ≡ iksst (unlabel S)"
```

abbreviation $db_{l_{sst}}$ where " $db_{l_{sst}} S \equiv db_{sst} (\text{unlabel } S)$ "

abbreviation $db'_{l_{sst}}$ where " $db'_{l_{sst}} S \equiv db'_{sst} (\text{unlabel } S)$ "

abbreviation $trms_{l_{sst}}$ where " $trms_{l_{sst}} S \equiv trms_{sst} (\text{unlabel } S)$ "

abbreviation $trms_proj_{l_{sst}}$ where " $trms_proj_{l_{sst}} n S \equiv trms_{sst} (\text{proj_unl } n S)$ "

abbreviation $vars_{l_{sst}}$ where " $vars_{l_{sst}} S \equiv vars_{sst} (\text{unlabel } S)$ "

abbreviation $vars_proj_{l_{sst}}$ where " $vars_proj_{l_{sst}} n S \equiv vars_{sst} (\text{proj_unl } n S)$ "

abbreviation $bvars_{l_{sst}}$ where " $bvars_{l_{sst}} S \equiv bvars_{sst} (\text{unlabel } S)$ "

abbreviation $fv_{l_{sst}}$ where " $fv_{l_{sst}} S \equiv fv_{sst} (\text{unlabel } S)$ "

Labeled set-operations

fun $setops_{l_{sst}p}$ where

" $setops_{l_{sst}p} (i, \text{insert}(t, s)) = \{(i, t, s)\}$ "
 | " $setops_{l_{sst}p} (i, \text{delete}(t, s)) = \{(i, t, s)\}$ "
 | " $setops_{l_{sst}p} (i, \langle _ : t \in s \rangle) = \{(i, t, s)\}$ "
 | " $setops_{l_{sst}p} (i, \forall _ (\forall \neq : _ \vee \notin : F')) = ((\lambda(t, s). (i, t, s)) \ ` \ \text{set } F')$ "
 | " $setops_{l_{sst}p} _ = \{\}$ "

definition $setops_{l_{sst}}$ where

" $setops_{l_{sst}} S \equiv \bigcup (setops_{l_{sst}p} \ ` \ \text{set } S)$ "

6.1.2 Minor Lemmata

lemma $in_ik_{l_{sst}}_iff$: " $t \in ik_{l_{sst}} A \iff (\exists ! ts. (l, \text{receive}(ts)) \in \text{set } A \wedge t \in \text{set } ts)$ "
 <proof>

lemma $ik_{l_{sst}}_concat$: " $ik_{l_{sst}} (\text{concat } xs) = \bigcup (ik_{l_{sst}} \ ` \ \text{set } xs)$ "
 <proof>

lemma $ik_{l_{sst}}_Cons[simp]$:

" $ik_{l_{sst}} ((l, \text{send}(ts))\#A) = ik_{l_{sst}} A$ " (is ?A)
 " $ik_{l_{sst}} ((l, \text{receive}(ts))\#A) = \text{set } ts \cup ik_{l_{sst}} A$ " (is ?B)
 " $ik_{l_{sst}} ((l, \langle \text{ac}: t \in s \rangle)\#A) = ik_{l_{sst}} A$ " (is ?C)
 " $ik_{l_{sst}} ((l, \text{insert}(t, s))\#A) = ik_{l_{sst}} A$ " (is ?D)
 " $ik_{l_{sst}} ((l, \text{delete}(t, s))\#A) = ik_{l_{sst}} A$ " (is ?E)
 " $ik_{l_{sst}} ((l, \langle \text{ac}: t \in s \rangle)\#A) = ik_{l_{sst}} A$ " (is ?F)
 " $ik_{l_{sst}} ((l, \forall X (\forall \neq : F \vee \notin : G))\#A) = ik_{l_{sst}} A$ " (is ?G)

<proof>

lemma $subst_lsstp_fst_eq$:

" $fst (a \cdot_{lsstp} \delta) = fst a$ "

<proof>

lemma $subst_lsst_map_fst_eq$:

" $\text{map } fst (S \cdot_{lsst} \delta) = \text{map } fst S$ "

<proof>

lemma $subst_lsst_nil[simp]$: " $[] \cdot_{lsst} \delta = []$ "

<proof>

lemma $subst_lsst_cons$: " $a\#A \cdot_{lsst} \delta = (a \cdot_{lsstp} \delta)\#(A \cdot_{lsst} \delta)$ "

<proof>

lemma $subst_lsstp_id_subst$: " $a \cdot_{lsstp} \text{Var} = a$ "

<proof>

lemma $subst_lsst_id_subst$: " $A \cdot_{lsst} \text{Var} = A$ "

<proof>

lemma $subst_lsst_singleton$: " $[(l, s)] \cdot_{lsst} \delta = [(l, s \cdot_{lsstp} \delta)]$ "

<proof>

lemma subst_lsst_append: " $A @ B \cdot_{lsst} \delta = (A \cdot_{lsst} \delta) @ (B \cdot_{lsst} \delta)$ "

<proof>

lemma subst_lsst_append_inv:

assumes " $A \cdot_{lsst} \delta = B1 @ B2$ "

shows " $\exists A1 A2. A = A1 @ A2 \wedge A1 \cdot_{lsst} \delta = B1 \wedge A2 \cdot_{lsst} \delta = B2$ "

<proof>

lemma subst_lsst_memI[*intro*]: " $x \in \text{set } A \implies x \cdot_{lsstp} \delta \in \text{set } (A \cdot_{lsst} \delta)$ "

<proof>

lemma subst_lsstpD:

" $a \cdot_{lsstp} \sigma = (n, \text{send}(ts)) \implies \exists ts'. ts = ts' \cdot_{list} \sigma \wedge a = (n, \text{send}(ts'))$ "

(is "?A \implies ?A'")

" $a \cdot_{lsstp} \sigma = (n, \text{receive}(ts)) \implies \exists ts'. ts = ts' \cdot_{list} \sigma \wedge a = (n, \text{receive}(ts'))$ "

(is "?B \implies ?B'")

" $a \cdot_{lsstp} \sigma = (n, \langle c: t \doteq s \rangle) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = (n, \langle c: t' \doteq s' \rangle)$ "

(is "?C \implies ?C'")

" $a \cdot_{lsstp} \sigma = (n, \text{insert}(t, s)) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = (n, \text{insert}(t', s'))$ "

(is "?D \implies ?D'")

" $a \cdot_{lsstp} \sigma = (n, \text{delete}(t, s)) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = (n, \text{delete}(t', s'))$ "

(is "?E \implies ?E'")

" $a \cdot_{lsstp} \sigma = (n, \langle c: t \in s \rangle) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = (n, \langle c: t' \in s' \rangle)$ "

(is "?F \implies ?F'")

" $a \cdot_{lsstp} \sigma = (n, \forall X(\forall \neq: F \vee \notin: G)) \implies$

$\exists F' G'. F = F' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge G = G' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge$

$a = (n, \forall X(\forall \neq: F' \vee \notin: G'))$ "

(is "?G \implies ?G'")

" $a \cdot_{lsstp} \sigma = (n, \langle t \neq s \rangle) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = (n, \langle t' \neq s' \rangle)$ "

(is "?H \implies ?H'")

" $a \cdot_{lsstp} \sigma = (n, \langle t \text{ not in } s \rangle) \implies \exists t' s'. t = t' \cdot \sigma \wedge s = s' \cdot \sigma \wedge a = (n, \langle t' \text{ not in } s' \rangle)$ "

(is "?I \implies ?I'")

<proof>

lemma subst_lsst_memD:

" $(n, \text{receive}(ts)) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists us. (n, \text{receive}(us)) \in \text{set } S \wedge ts = us \cdot_{list} \sigma$ "

" $(n, \text{send}(ts)) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists us. (n, \text{send}(us)) \in \text{set } S \wedge ts = us \cdot_{list} \sigma$ "

" $(n, \langle ac: t \doteq s \rangle) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists u v. (n, \langle ac: u \doteq v \rangle) \in \text{set } S \wedge t = u \cdot \sigma \wedge s = v \cdot \sigma$ "

" $(n, \text{insert}(t, s)) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists u v. (n, \text{insert}(u, v)) \in \text{set } S \wedge t = u \cdot \sigma \wedge s = v \cdot \sigma$ "

" $(n, \text{delete}(t, s)) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists u v. (n, \text{delete}(u, v)) \in \text{set } S \wedge t = u \cdot \sigma \wedge s = v \cdot \sigma$ "

" $(n, \langle ac: t \in s \rangle) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists u v. (n, \langle ac: u \in v \rangle) \in \text{set } S \wedge t = u \cdot \sigma \wedge s = v \cdot \sigma$ "

" $(n, \forall X(\forall \neq: F \vee \notin: G)) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists F' G'. (n, \forall X(\forall \neq: F' \vee \notin: G')) \in \text{set } S \wedge$

$F = F' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma \wedge$

$G = G' \cdot_{pairs} \text{rm_vars } (\text{set } X) \sigma$ "

" $(n, \langle t \neq s \rangle) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists u v. (n, \langle u \neq v \rangle) \in \text{set } S \wedge t = u \cdot \sigma \wedge s = v \cdot \sigma$ "

" $(n, \langle t \text{ not in } s \rangle) \in \text{set } (S \cdot_{lsst} \sigma) \implies$

$\exists u v. (n, \langle u \text{ not in } v \rangle) \in \text{set } S \wedge t = u \cdot \sigma \wedge s = v \cdot \sigma$ "

<proof>

lemma subst_lsst_unlabel_cons: " $\text{unlabel } ((l, b) \# A \cdot_{lsst} \vartheta) = (b \cdot_{sstp} \vartheta) \# (\text{unlabel } (A \cdot_{lsst} \vartheta))$ "

<proof>

lemma subst_lsst_unlabel: " $\text{unlabel } (A \cdot_{lsst} \delta) = \text{unlabel } A \cdot_{sst} \delta$ "

<proof>

lemma *subst_lsst_unlabel_member*[intro]:
 assumes "x ∈ set (unlabel A)"
 shows "x ·_{sstp} δ ∈ set (unlabel (A ·_{lsst} δ))"
<proof>

lemma *subst_lsst_prefix*:
 assumes "prefix B (A ·_{lsst} ϑ)"
 shows "∃ C. C ·_{lsst} ϑ = B ∧ prefix C A"
<proof>

lemma *subst_lsst_tl*:
 "tl (S ·_{lsst} δ) = tl S ·_{lsst} δ"
<proof>

lemma *dual_{lsst}_tl*:
 "tl (dual_{lsst} S) = dual_{lsst} (tl S)"
<proof>

lemma *dual_{lsstp}_fst_eq*:
 "fst (dual_{lsstp} a) = fst a"
<proof>

lemma *dual_{lsst}_map_fst_eq*:
 "map fst (dual_{lsst} S) = map fst S"
<proof>

lemma *dual_{lsst}_nil*[simp]: "dual_{lsst} [] = []"
<proof>

lemma *dual_{lsst}_Cons*[simp]:
 "dual_{lsst} ((l, send⟨ts⟩)#A) = (l, receive⟨ts⟩)#(dual_{lsst} A)"
 "dual_{lsst} ((l, receive⟨ts⟩)#A) = (l, send⟨ts⟩)#(dual_{lsst} A)"
 "dual_{lsst} ((l, ⟨a: t ≐ s⟩)#A) = (l, ⟨a: t ≐ 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 ·_{lsstp} δ) = (dual_{lsstp} s) ·_{lsstp} δ"
<proof>

lemma *dual_{lsst}_subst*: "dual_{lsst} (S ·_{lsst} δ) = (dual_{lsst} S) ·_{lsst} δ"
<proof>

lemma *dual_{lsst}_subst_unlabel*: "unlabel (dual_{lsst} (S ·_{lsst} δ)) = unlabel (dual_{lsst} S) ·_{sst} δ"
<proof>

lemma *dual_{lsst}_subst_cons*: "dual_{lsst} (a#A ·_{lsst} σ) = (dual_{lsstp} a ·_{lsstp} σ)#(dual_{lsst} (A ·_{lsst} σ))"
<proof>

lemma *dual_{lsst}_subst_append*: "dual_{lsst} (A@B ·_{lsst} σ) = (dual_{lsst} A@dual_{lsst} B) ·_{lsst} σ"
<proof>

lemma *dual_{lsst}_subst_snoc*: "dual_{lsst} (A@[a] ·_{lsst} σ) = (dual_{lsst} A ·_{lsst} σ)@[dual_{lsstp} a ·_{lsstp} σ]"
<proof>

lemma *dual_{lsst}_memberD*:

assumes " $(l,a) \in \text{set}(\text{dual}_{l_{sst}} A)$ "
 shows " $\exists b. (l,b) \in \text{set} A \wedge \text{dual}_{l_{sstp}}(l,b) = (l,a)$ "
 <proof>

lemma $\text{dual}_{l_{sst_memberD'}}$:
 assumes $a: "a \in \text{set}(\text{dual}_{l_{sst}} A \cdot_{l_{sst}} \delta)"$
 obtains b where " $b \in \text{set} A$ " " $a = \text{dual}_{l_{sstp}} b \cdot_{l_{sstp}} \delta$ " " $\text{fst } a = \text{fst } b$ "
 <proof>

lemma $\text{dual}_{l_{sstp_inv}}$:
 assumes " $\text{dual}_{l_{sstp}}(l, a) = (k, b)$ "
 shows " $l = k$ "
 and " $a = \text{receive}(t) \implies b = \text{send}(t)$ "
 and " $a = \text{send}(t) \implies b = \text{receive}(t)$ "
 and " $(\exists t. a = \text{receive}(t) \vee a = \text{send}(t)) \implies b = a$ "
 <proof>

lemma $\text{dual}_{l_{sst_self_inverse}}$: " $\text{dual}_{l_{sst}}(\text{dual}_{l_{sst}} A) = A$ "
 <proof>

lemma $\text{dual}_{l_{sst_unlabel_cong}}$:
 assumes " $\text{unlabel } S = \text{unlabel } S'$ "
 shows " $\text{unlabel}(\text{dual}_{l_{sst}} S) = \text{unlabel}(\text{dual}_{l_{sst}} S')$ "
 <proof>

lemma $\text{vars}_{sst_unlabel_dual_{l_{sst}}_eq}$: " $\text{vars}_{l_{sst}}(\text{dual}_{l_{sst}} A) = \text{vars}_{l_{sst}} A$ "
 <proof>

lemma $\text{fv}_{sst_unlabel_dual_{l_{sst}}_eq}$: " $\text{fv}_{l_{sst}}(\text{dual}_{l_{sst}} A) = \text{fv}_{l_{sst}} A$ "
 <proof>

lemma $\text{bvars}_{sst_unlabel_dual_{l_{sst}}_eq}$: " $\text{bvars}_{l_{sst}}(\text{dual}_{l_{sst}} A) = \text{bvars}_{l_{sst}} A$ "
 <proof>

lemma $\text{vars}_{sst_unlabel_Cons}$: " $\text{vars}_{l_{sst}}((l,b)\#A) = \text{vars}_{sstp} b \cup \text{vars}_{l_{sst}} A$ "
 <proof>

lemma $\text{fv}_{sst_unlabel_Cons}$: " $\text{fv}_{l_{sst}}((l,b)\#A) = \text{fv}_{sstp} b \cup \text{fv}_{l_{sst}} A$ "
 <proof>

lemma $\text{bvars}_{sst_unlabel_Cons}$: " $\text{bvars}_{l_{sst}}((l,b)\#A) = \text{set}(\text{bvars}_{sstp} b) \cup \text{bvars}_{l_{sst}} A$ "
 <proof>

lemma $\text{bvars}_{l_{sst_subst}}$: " $\text{bvars}_{l_{sst}}(A \cdot_{l_{sst}} \delta) = \text{bvars}_{l_{sst}} A$ "
 <proof>

lemma $\text{dual}_{l_{sst_member}}$:
 assumes " $(l,x) \in \text{set} A$ "
 and " $\neg \text{is_Receive } x$ " " $\neg \text{is_Send } x$ "
 shows " $(l,x) \in \text{set}(\text{dual}_{l_{sst}} A)$ "
 <proof>

lemma $\text{dual}_{l_{sst_unlabel_member}}$:
 assumes " $x \in \text{set}(\text{unlabel } A)$ "
 and " $\neg \text{is_Receive } x$ " " $\neg \text{is_Send } x$ "
 shows " $x \in \text{set}(\text{unlabel}(\text{dual}_{l_{sst}} A))$ "
 <proof>

lemma $\text{dual}_{l_{sst_steps_iff}}$:
 " $(l, \text{send}(ts)) \in \text{set} A \iff (l, \text{receive}(ts)) \in \text{set}(\text{dual}_{l_{sst}} A)$ "
 " $(l, \text{receive}(ts)) \in \text{set} A \iff (l, \text{send}(ts)) \in \text{set}(\text{dual}_{l_{sst}} A)$ "
 " $(l, \langle c: t \dot{=} s \rangle) \in \text{set} A \iff (l, \langle c: t \dot{=} s \rangle) \in \text{set}(\text{dual}_{l_{sst}} A)$ "
 " $(l, \text{insert}(t,s)) \in \text{set} A \iff (l, \text{insert}(t,s)) \in \text{set}(\text{dual}_{l_{sst}} A)$ "

"(l,delete(t,s)) ∈ set A ↔ (l,delete(t,s)) ∈ set (dual_{l_{sst}} A)"
 "(l,<c: t ∈ s>) ∈ set A ↔ (l,<c: t ∈ s>) ∈ set (dual_{l_{sst}} A)"
 "(l,∀X(∀≠: F ∨≠: G)) ∈ set A ↔ (l,∀X(∀≠: F ∨≠: G)) ∈ set (dual_{l_{sst}} A)"
 <proof>

lemma dual_{l_{sst}}_unlabel_steps_iff:

"send(ts) ∈ set (unlabel A) ↔ receive(ts) ∈ set (unlabel (dual_{l_{sst}} A))"
 "receive(ts) ∈ set (unlabel A) ↔ send(ts) ∈ set (unlabel (dual_{l_{sst}} A))"
 "<c: t ≐ s> ∈ set (unlabel A) ↔ <c: t ≐ s> ∈ set (unlabel (dual_{l_{sst}} A))"
 "insert(t,s) ∈ set (unlabel A) ↔ insert(t,s) ∈ set (unlabel (dual_{l_{sst}} A))"
 "delete(t,s) ∈ set (unlabel A) ↔ delete(t,s) ∈ set (unlabel (dual_{l_{sst}} A))"
 "<c: t ∈ s> ∈ set (unlabel A) ↔ <c: t ∈ s> ∈ set (unlabel (dual_{l_{sst}} A))"
 "∀X(∀≠: F ∨≠: G) ∈ set (unlabel A) ↔ ∀X(∀≠: F ∨≠: G) ∈ set (unlabel (dual_{l_{sst}} A))"
 <proof>

lemma dual_{l_{sst}}_list_all:

"list_all is_Receive (unlabel A) ↔ list_all is_Send (unlabel (dual_{l_{sst}} A))"
 "list_all is_Send (unlabel A) ↔ list_all is_Receive (unlabel (dual_{l_{sst}} A))"
 "list_all is_Equality (unlabel A) ↔ list_all is_Equality (unlabel (dual_{l_{sst}} A))"
 "list_all is_Insert (unlabel A) ↔ list_all is_Insert (unlabel (dual_{l_{sst}} A))"
 "list_all is_Delete (unlabel A) ↔ list_all is_Delete (unlabel (dual_{l_{sst}} A))"
 "list_all is_InSet (unlabel A) ↔ list_all is_InSet (unlabel (dual_{l_{sst}} A))"
 "list_all is_NegChecks (unlabel A) ↔ list_all is_NegChecks (unlabel (dual_{l_{sst}} A))"
 "list_all is_Assignment (unlabel A) ↔ list_all is_Assignment (unlabel (dual_{l_{sst}} A))"
 "list_all is_Check (unlabel A) ↔ list_all is_Check (unlabel (dual_{l_{sst}} A))"
 "list_all is_Update (unlabel A) ↔ list_all is_Update (unlabel (dual_{l_{sst}} A))"
 "list_all is_Check_or_Assignment (unlabel A) ↔
 list_all is_Check_or_Assignment (unlabel (dual_{l_{sst}} A))"
 <proof>

lemma dual_{l_{sst}}_list_all_same:

"list_all is_Equality (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_Insert (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_Delete (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_InSet (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_NegChecks (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_Assignment (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_Check (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_Update (unlabel A) ⇒ dual_{l_{sst}} A = A"
 "list_all is_Check_or_Assignment (unlabel A) ⇒ dual_{l_{sst}} A = A"
 <proof>

lemma dual_{l_{sst}}_in_set_prefix_obtain:

assumes "s ∈ set (unlabel (dual_{l_{sst}} A))"
 shows "∃ l B s'. (l,s) = dual_{l_{sstp}} (l,s') ∧ prefix (B@[l,s']) A"
 <proof>

lemma dual_{l_{sst}}_in_set_prefix_obtain_subst:

assumes "s ∈ set (unlabel (dual_{l_{sst}} (A ·_{l_{sst}} ∅)))"
 shows "∃ l B s'. (l,s) = dual_{l_{sstp}} ((l,s') ·_{l_{sstp}} ∅) ∧ prefix ((B ·_{l_{sst}} ∅)@[l,s'] ·_{l_{sstp}} ∅) (A ·_{l_{sst}} ∅)"
 <proof>

lemma trms_{sst}_unlabel_dual_{l_{sst}}_eq: "trms_{l_{sst}} (dual_{l_{sst}} A) = trms_{l_{sst}} A"

<proof>

lemma trms_{sst}_unlabel_subst_cons:

"trms_{l_{sst}} ((l,b)#A ·_{l_{sst}} δ) = trms_{sstp} (b ·_{sstp} δ) ∪ trms_{l_{sst}} (A ·_{l_{sst}} δ)"
 <proof>

lemma trms_{sst}_unlabel_subst:

assumes "bvars_{l_{sst}} S ∩ subst_domain ∅ = {}"
 shows "trms_{l_{sst}} (S ·_{l_{sst}} ∅) = trms_{l_{sst}} S ·_{set} ∅"
 <proof>

```

lemma trmssst_unlabel_subst':
  fixes t::('a,'b) term and δ::('a,'b) subst
  assumes "t ∈ trmslsst (S ·lsst δ)"
  shows "∃s ∈ trmslsst S. ∃X. set X ⊆ bvarslsst S ∧ t = s · rm_vars (set X) δ"
⟨proof⟩

lemma trmssst_unlabel_subst'':
  fixes t::('a,'b) term and δ ∅::('a,'b) subst
  assumes "t ∈ trmslsst (S ·lsst δ) ·set ∅"
  shows "∃s ∈ trmslsst S. ∃X. set X ⊆ bvarslsst S ∧ t = s · rm_vars (set X) δ ∘s ∅"
⟨proof⟩

lemma trmssst_unlabel_dual_subst_cons:
  "trmslsst (duallsst (a#A ·lsst σ)) = (trmssstp (snd a ·sstp σ)) ∪ (trmslsst (duallsst (A ·lsst σ)))"
⟨proof⟩

lemma duallsst_funs_term:
  "∪ (funs_term ` (trmssst (unlabel (duallsst S)))) = ∪ (funs_term ` (trmssst (unlabel S)))"
⟨proof⟩

lemma duallsst_dblsst:
  "db'lsst (duallsst A) = db'lsst A"
⟨proof⟩

lemma dbsst_unlabel_append:
  "db'lsst (A@B) I D = db'lsst B I (db'lsst A I D)"
⟨proof⟩

lemma dbsst_duallsst:
  "db'sst (unlabel (duallsst (T ·lsst δ))) I D = db'sst (unlabel (T ·lsst δ)) 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 in_proj_set:
  assumes "<l,r> ∈ set A"
  shows "<l,r> ∈ set (proj l A)"
⟨proof⟩

lemma proj_subst: "proj l (A ·lsst δ) = proj l A ·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_mem_iff:

```

"(ln i, d) ∈ set D ↔ (ln i, d) ∈ set (proj i D)"
 "(*, d) ∈ set D ↔ (*, d) ∈ set (proj i 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_{l_{sst}}:

"proj l (dual_{l_{sst}} A) = dual_{l_{sst}} (proj l A)"
 ⟨proof⟩

lemma proj_instance_ex:

assumes B: "∀ b ∈ set B. ∃ a ∈ set A. ∃ δ. b = a ·_{l_{sst}} δ ∧ P δ"
 and b: "b ∈ set (proj l B)"
 shows "∃ a ∈ set (proj l A). ∃ δ. b = a ·_{l_{sst}} 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 ∅ = (S ·lsst δ) ·lsst ∅"`

<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 `setopslsstpD`:

`assumes p: "p ∈ setopslsstp 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 $\text{setops}_{l_{sst}}\text{-nil}[\text{simp}]$:

" $\text{setops}_{l_{sst}} [] = \{\}$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sst}}\text{-cons}[\text{simp}]$:

" $\text{setops}_{l_{sst}} (x\#S) = \text{setops}_{l_{sstp}} x \cup \text{setops}_{l_{sst}} S$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{sst}\text{-proj_subset}$:

" $\text{setops}_{sst} (\text{proj_unl } n \ A) \subseteq \text{setops}_{sst} (\text{unlabel } A)$ "

" $\text{setops}_{sst} (\text{proj_unl } m \ (\text{proj } n \ A)) \subseteq \text{setops}_{sst} (\text{proj_unl } n \ A)$ "

" $\text{setops}_{sst} (\text{proj_unl } m \ (\text{proj } n \ A)) \subseteq \text{setops}_{sst} (\text{proj_unl } m \ A)$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{sst}\text{-unlabel_prefix_subset}$:

" $\text{setops}_{sst} (\text{unlabel } A) \subseteq \text{setops}_{sst} (\text{unlabel } (A@B))$ "

" $\text{setops}_{sst} (\text{proj_unl } n \ A) \subseteq \text{setops}_{sst} (\text{proj_unl } n \ (A@B))$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{sst}\text{-unlabel_suffix_subset}$:

" $\text{setops}_{sst} (\text{unlabel } B) \subseteq \text{setops}_{sst} (\text{unlabel } (A@B))$ "

" $\text{setops}_{sst} (\text{proj_unl } n \ B) \subseteq \text{setops}_{sst} (\text{proj_unl } n \ (A@B))$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sst}}\text{-proj_subset}$:

" $\text{setops}_{l_{sst}} (\text{proj } n \ A) \subseteq \text{setops}_{l_{sst}} A$ "

" $\text{setops}_{l_{sst}} (\text{proj } m \ (\text{proj } n \ A)) \subseteq \text{setops}_{l_{sst}} (\text{proj } n \ A)$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sst}}\text{-prefix_subset}$:

" $\text{setops}_{l_{sst}} A \subseteq \text{setops}_{l_{sst}} (A@B)$ "

" $\text{setops}_{l_{sst}} (\text{proj } n \ A) \subseteq \text{setops}_{l_{sst}} (\text{proj } n \ (A@B))$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sst}}\text{-suffix_subset}$:

" $\text{setops}_{l_{sst}} B \subseteq \text{setops}_{l_{sst}} (A@B)$ "

" $\text{setops}_{l_{sst}} (\text{proj } n \ B) \subseteq \text{setops}_{l_{sst}} (\text{proj } n \ (A@B))$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sst}}\text{-mono}$:

" $\text{set } M \subseteq \text{set } N \implies \text{setops}_{l_{sst}} M \subseteq \text{setops}_{l_{sst}} N$ "

$\langle \text{proof} \rangle$

lemma $\text{trms}_{sst}\text{-unlabel_subset_if_no_label}$:

" $\neg \text{list_ex } (\text{has_LabelN } l) \ A \implies \text{trms}_{l_{sst}} (\text{proj } l \ A) \subseteq \text{trms}_{l_{sst}} (\text{proj } l' \ A)$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{sst}\text{-unlabel_subset_if_no_label}$:

" $\neg \text{list_ex } (\text{has_LabelN } l) \ A \implies \text{setops}_{sst} (\text{proj_unl } l \ A) \subseteq \text{setops}_{sst} (\text{proj_unl } l' \ A)$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sst}}\text{-proj_subset_if_no_label}$:

" $\neg \text{list_ex } (\text{has_LabelN } l) \ A \implies \text{setops}_{l_{sst}} (\text{proj } l \ A) \subseteq \text{setops}_{l_{sst}} (\text{proj } l' \ A)$ "

$\langle \text{proof} \rangle$

lemma $\text{setops}_{l_{sstp}}\text{-subst_cases}[\text{simp}]$:

" $\text{setops}_{l_{sstp}} ((l, \text{send}(ts)) \cdot_{l_{sstp}} \delta) = \{\}$ "

" $\text{setops}_{l_{sstp}} ((l, \text{receive}(ts)) \cdot_{l_{sstp}} \delta) = \{\}$ "

" $\text{setops}_{l_{sstp}} ((l, (\text{ac}: s \doteq t)) \cdot_{l_{sstp}} \delta) = \{\}$ "

" $\text{setops}_{l_{sstp}} ((l, \text{insert}(t, s)) \cdot_{l_{sstp}} \delta) = \{(l, t \cdot \delta, s \cdot \delta)\}$ "

" $\text{setops}_{l_{sstp}} ((l, \text{delete}(t, s)) \cdot_{l_{sstp}} \delta) = \{(l, t \cdot \delta, s \cdot \delta)\}$ "

" $\text{setops}_{l_{sstp}} ((l, (\text{ac}: t \in s)) \cdot_{l_{sstp}} \delta) = \{(l, t \cdot \delta, s \cdot \delta)\}$ "

```

"setopslsstp ((1,∀X(∇≠: F ∇≠: F')) ·lsstp δ) =
  ((λ(t,s). (1,t · rm_vars (set X) δ,s · rm_vars (set X) δ)) ` set F)" (is "?A = ?B")
⟨proof⟩

lemma setopslsstp_subst:
  assumes "set (bvarssstp (snd a)) ∩ subst_domain ∅ = {}"
  shows "setopslsstp (a ·lsstp ∅) = (λp. (fst a,snd p ·p ∅)) ` setopslsstp a"
⟨proof⟩

lemma setopslsstp_subst':
  assumes "set (bvarssstp (snd a)) ∩ subst_domain ∅ = {}"
  shows "setopslsstp (a ·lsstp ∅) = (λ(i,p). (i,p ·p ∅)) ` setopslsstp a"
⟨proof⟩

lemma setopslsst_subst:
  assumes "bvarslsst S ∩ subst_domain ∅ = {}"
  shows "setopslsst (S ·lsst ∅) = (λp. (fst p,snd p ·p ∅)) ` setopslsst S"
⟨proof⟩

lemma setopslsstp_in_subst:
  assumes p: "p ∈ setopslsstp (a ·lsstp δ)"
  shows "∃q ∈ setopslsstp a. fst p = fst q ∧ snd p = snd q ·p rm_vars (set (bvarssstp (snd a))) δ"
  (is "∃q ∈ setopslsstp a. ?P q")
⟨proof⟩

lemma setopslsst_in_subst:
  assumes "p ∈ setopslsst (A ·lsst δ)"
  shows "∃q ∈ setopslsst A. fst p = fst q ∧ (∃X ⊆ bvarslsst A. snd p = snd q ·p rm_vars X δ)"
  (is "∃q ∈ setopslsst A. ?P A q")
⟨proof⟩

lemma setopslsst_duallsst_eq:
  "setopslsst (duallsst A) = setopslsst A"
⟨proof⟩

end

```

6.2 Stateful Protocol 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 "wtsubst δ"
  and "∀t ∈ subst_range δ. fv t = {} ∨ (∃x. t = Var x)"
  shows "Γ ` Var ` fvsstp (a ·sstp δ) ⊆ Γ ` Var ` fvsstp a" (is ?A)
  and "Γ ` Var ` set (bvarssstp (a ·sstp δ)) = Γ ` Var ` set (bvarssstp a)" (is ?B)
  and "Γ ` Var ` varssstp (a ·sstp δ) ⊆ Γ ` Var ` varssstp a" (is ?C)
⟨proof⟩

lemma (in typed_model) wt_subst_lsst_vars_type_subset:
  fixes A:: "('fun, 'var, 'a) labeled_stateful_strand"
  assumes "wtsubst δ"
  and "∀t ∈ subst_range δ. fv t = {} ∨ (∃x. t = Var x)"
  shows "Γ ` Var ` fvlsst (A ·lsst δ) ⊆ Γ ` Var ` fvlsst A" (is ?A)
  and "Γ ` Var ` bvarslsst (A ·lsst δ) = Γ ` Var ` bvarslsst A" (is ?B)
  and "Γ ` Var ` varslsst (A ·lsst δ) ⊆ Γ ` Var ` varslsst A" (is ?C)

```

<proof>

lemma (in *stateful_typed_model*) *fv_pair_fv_pairs_subset*:

assumes " $d \in \text{set } D$ "

shows " $\text{fv}(\text{pair}(\text{snd } d)) \subseteq \text{fv}_{\text{pairs}}(\text{unlabel } D)$ "

<proof>

lemma (in *stateful_typed_model*) *labeled_sat_ineq_lift*:

assumes " $\llbracket M; \text{map}(\lambda d. \forall X(\forall \neq: [(\text{pair}(t,s), \text{pair}(\text{snd } d))]))_{st} [d \leftarrow \text{dbproj } i D. d \notin \text{set } Di] \rrbracket_d \mathcal{I}$ "
(is "*?R1 D*")

and " $\forall (j,p) \in \{(i,t,s)\} \cup \text{set } D \cup \text{set } Di. \forall (k,q) \in \{(i,t,s)\} \cup \text{set } D \cup \text{set } Di.$
 $(\exists \delta. \text{Unifier } \delta(\text{pair } p)(\text{pair } q)) \longrightarrow j = k$ " (is "*?R2 D*")

shows " $\llbracket M; \text{map}(\lambda d. \forall X(\forall \neq: [(\text{pair}(t,s), \text{pair}(\text{snd } d))]))_{st} [d \leftarrow D. d \notin \text{set } Di] \rrbracket_d \mathcal{I}$ "

<proof>

lemma (in *stateful_typed_model*) *labeled_sat_ineq_dbproj*:

assumes " $\llbracket M; \text{map}(\lambda d. \forall X(\forall \neq: [(\text{pair}(t,s), \text{pair}(\text{snd } d))]))_{st} [d \leftarrow D. d \notin \text{set } Di] \rrbracket_d \mathcal{I}$ "
(is "*?P D*")

shows " $\llbracket M; \text{map}(\lambda d. \forall X(\forall \neq: [(\text{pair}(t,s), \text{pair}(\text{snd } d))]))_{st} [d \leftarrow \text{dbproj } i D. d \notin \text{set } Di] \rrbracket_d \mathcal{I}$ "
(is "*?Q D*")

<proof>

lemma (in *stateful_typed_model*) *labeled_sat_ineq_dbproj_sem_equiv*:

assumes " $\forall (j,p) \in ((\lambda(t,s). (i,t,s)) \setminus \text{set } F') \cup \text{set } D.$

$\forall (k,q) \in ((\lambda(t,s). (i,t,s)) \setminus \text{set } F') \cup \text{set } D.$

$(\exists \delta. \text{Unifier } \delta(\text{pair } p)(\text{pair } q)) \longrightarrow j = k$ "

and " $\text{fv}_{\text{pairs}}(\text{map } \text{snd } D) \cap \text{set } X = \{\}$ "

shows " $\llbracket M; \text{map}(\lambda G. \forall X(\forall \neq: (F \circ G))_{st}) (\text{tr}_{\text{pairs}} F'(\text{map } \text{snd } D)) \rrbracket_d \mathcal{I} \longleftrightarrow$

$\llbracket M; \text{map}(\lambda G. \forall X(\forall \neq: (F \circ G))_{st}) (\text{tr}_{\text{pairs}} F'(\text{map } \text{snd } (\text{dbproj } i D))) \rrbracket_d \mathcal{I}$ "

<proof>

lemma (in *stateful_typed_model*) *labeled_sat_eqs_list_all*:

assumes " $\forall (j,p) \in \{(i,t,s)\} \cup \text{set } D. \forall (k,q) \in \{(i,t,s)\} \cup \text{set } D.$

$(\exists \delta. \text{Unifier } \delta(\text{pair } p)(\text{pair } q)) \longrightarrow j = k$ " (is "*?P D*")

and " $\llbracket M; \text{map}(\lambda d. \langle \text{ac}: (\text{pair}(t,s)) \doteq (\text{pair}(\text{snd } d)) \rangle_{st}) D \rrbracket_d \mathcal{I}$ " (is "*?Q D*")

shows " $\text{list_all}(\lambda d. \text{fst } d = i) D$ "

<proof>

lemma (in *stateful_typed_model*) *labeled_sat_eqs_subseqs*:

assumes " $Di \in \text{set}(\text{subseqs } D)$ "

and " $\forall (j,p) \in \{(i,t,s)\} \cup \text{set } D. \forall (k,q) \in \{(i,t,s)\} \cup \text{set } D.$

$(\exists \delta. \text{Unifier } \delta(\text{pair } p)(\text{pair } q)) \longrightarrow j = k$ " (is "*?P D*")

and " $\llbracket M; \text{map}(\lambda d. \langle \text{ac}: (\text{pair}(t,s)) \doteq (\text{pair}(\text{snd } d)) \rangle_{st}) Di \rrbracket_d \mathcal{I}$ "

shows " $Di \in \text{set}(\text{subseqs}(\text{dbproj } i D))$ "

<proof>

lemma (in *stateful_typing_result*) *dual_{lst}_tfr_{sstp}*:

assumes " $\text{list_all } \text{tfr}_{\text{sstp}}(\text{unlabel } S)$ "

shows " $\text{list_all } \text{tfr}_{\text{sstp}}(\text{unlabel}(\text{dual}_{\text{lst}} S))$ "

<proof>

lemma (in *stateful_typed_model*) *setops_{sst}_unlabel_{dual_{lst}}_eq*:

" $\text{setops}_{\text{sst}}(\text{unlabel}(\text{dual}_{\text{lst}} A)) = \text{setops}_{\text{sst}}(\text{unlabel } A)$ "

<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 \Rightarrow nat"

and public: "'fun \Rightarrow bool"

and Ana: "('fun, 'var) term \Rightarrow (('fun, 'var) term list \times ('fun, 'var) term list)"

```

and  $\Gamma::('fun, 'var) \text{ term} \Rightarrow ('fun, 'atom::finite) \text{ term\_type}$ 
and Pair::"'fun"
and label_witness1::"'lbl"
and label_witness2::"'lbl"
begin

```

```

definition lpair where

```

```

"lpair lp  $\equiv$  case lp of (i,p)  $\Rightarrow$  (i,pair p)"

```

```

lemma setopsl_sstp_pair_image[simp]:

```

```

"lpair  $\setminus$  (setopsl_sstp (i,send<ts>)) = {}"
"lpair  $\setminus$  (setopsl_sstp (i,receive<ts>)) = {}"
"lpair  $\setminus$  (setopsl_sstp (i,<ac: t  $\doteq$  t'>)) = {}"
"lpair  $\setminus$  (setopsl_sstp (i,insert(t,s))) = {(i, pair (t,s))}"
"lpair  $\setminus$  (setopsl_sstp (i,delete(t,s))) = {(i, pair (t,s))}"
"lpair  $\setminus$  (setopsl_sstp (i,<ac: t  $\in$  s>)) = {(i, pair (t,s))}"
"lpair  $\setminus$  (setopsl_sstp (i, $\forall X(\forall \neq: F \vee \notin: F')$ )) = (( $\lambda(t,s). (i, pair (t,s))$ )  $\setminus$  set F)"
<proof>

```

```

definition par_compl_sst where

```

```

"par_compl_sst (A::('fun, 'var, 'lbl) labeled_stateful_strand) (Secrets::('fun, 'var) terms)  $\equiv$ 
( $\forall l1 l2. l1 \neq l2 \rightarrow$ 
  GSMP_disjoint (trmssst (proj_unl l1 A)  $\cup$  pair  $\setminus$  setopssst (proj_unl l1 A))
    (trmssst (proj_unl l2 A)  $\cup$  pair  $\setminus$  setopssst (proj_unl l2 A)) Secrets)  $\wedge$ 
( $\forall s \in$  Secrets.  $\neg\{s\} \vdash_c s$ )  $\wedge$  ground Secrets  $\wedge$ 
( $\forall (i,p) \in$  setopsl_sst A.  $\forall (j,q) \in$  setopsl_sst A.
  ( $\exists \delta. \text{Unifier } \delta$  (pair p) (pair q))  $\rightarrow i = j$ )")

```

```

definition declassifiedl_sst where

```

```

"declassifiedl_sst A  $\mathcal{I} \equiv$  {s.  $\bigcup$  {set ts | ts. (<*, receive<ts>)>  $\in$  set (A  $\cdot$ l_sst  $\mathcal{I}$ )}  $\vdash s$ }"

```

```

definition strand_leaksl_sst (<_ leaks _ under _>) where

```

```

"(A::('fun, 'var, 'lbl) labeled_stateful_strand) leaks Secrets under  $\mathcal{I} \equiv$ 
( $\exists t \in$  Secrets - declassifiedl_sst A  $\mathcal{I}. \exists n. \mathcal{I} \models_s$  (proj_unl n A@[send<[t]>]))"

```

```

type_synonym ('a, 'b, 'c) labeleddbstate = "('c strand_label  $\times$  ((('a, 'b) term  $\times$  ('a, 'b) term)) set"

```

```

type_synonym ('a, 'b, 'c) labeleddbstatelist = "('c strand_label  $\times$  ((('a, 'b) term  $\times$  ('a, 'b) term))
list"

```

```

definition typing_condsst where

```

```

"typing_condsst A  $\equiv$  wfsst A  $\wedge$  wftrms (trmssst A)  $\wedge$  tfrsst A"

```

For proving the compositionality theorem for stateful constraints the idea is to first define a variant of the reduction technique that was used to establish the stateful typing result. This variant performs database-state projections, and it allows us to reduce the compositionality problem for stateful constraints to ordinary constraints.

```

fun trpc::

```

```

('fun, 'var, 'lbl) labeled_stateful_strand  $\Rightarrow$  ('fun, 'var, 'lbl) labeleddbstatelist
 $\Rightarrow$  ('fun, 'var, 'lbl) labeled_strand list"

```

```

where

```

```

"trpc [] D = [[]]"
| "trpc ((i,send<ts>)#A) D = map ((#) (i,send<ts>)st) (trpc A D)"
| "trpc ((i,receive<ts>)#A) D = map ((#) (i,receive<ts>)st) (trpc A D)"
| "trpc ((i,<ac: t  $\doteq$  t'>)#A) D = map ((#) (i,<ac: t  $\doteq$  t'>)st) (trpc A D)"
| "trpc ((i,insert(t,s))#A) D = trpc A (List.insert (i,(t,s)) D)"
| "trpc ((i,delete(t,s))#A) D = (
  concat (map ( $\lambda Di. \text{map } (\lambda B. (\text{map } (\lambda d. (i,<check: (pair (t,s)) \doteq (pair (snd d))>)st) Di)@
    (\text{map } (\lambda d. (i,\forall [](\forall \neq: [(pair (t,s), pair (snd d))])st)
      [d $\leftarrow$ dbproj i D. d  $\notin$  set Di])@B)
    (trpc A [d $\leftarrow$ D. d  $\notin$  set Di]))
    (subseqs (dbproj i D))))"
| "trpc ((i,<ac: t  $\in$  s>)#A) D =
  concat (map ( $\lambda B. \text{map } (\lambda d. (i,<ac: (pair (t,s)) \doteq (pair (snd d))>)st)#B$ ) (dbproj i D)) (trpc A D))"$ 
```

```

| "trpc ((i,∀X(∀≠: F ∨≠: F' ))#A) D =
  map ((@) (map (λG. (i,∀X(∀≠: (F@G)st))) (trpairs F' (map snd (dbproj i D)))))) (trpc A D)"

end

locale labeled_stateful_typing =
  labeled_stateful_typed_model arity public Ana Γ Pair label_witness1 label_witness2
+ stateful_typing_result arity public Ana Γ Pair
  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

sublocale labeled_typing
⟨proof⟩

end

```

6.2.3 Small Lemmata

```

context labeled_stateful_typed_model
begin

lemma declassifiedlsst_alt_def:
  "declassifiedlsst A I = {s. ∪ {set ts | ts. ⟨*, receive⟨ts⟩⟩ ∈ set A} ·set I ⊢ s}"
⟨proof⟩

lemma declassifiedlsst_prefix_subset:
  assumes AB: "prefix A B"
  shows "declassifiedlsst A I ⊆ declassifiedlsst B I"
⟨proof⟩

lemma declassifiedlsst_star_receive_supset:
  "{t | t ts. ⟨*, receive⟨ts⟩⟩ ∈ set A ∧ t ∈ set ts} ·set I ⊆ declassifiedlsst A I"
⟨proof⟩

lemma declassifiedlsst_proj_eq:
  "declassifiedlsst A I = declassifiedlsst (proj n A) I"
⟨proof⟩

lemma par_complsst_nil:
  assumes "ground Sec" "∀s ∈ Sec. ∀s' ∈ subterms s. {} ⊢c s' ∨ s' ∈ Sec" "∀s ∈ Sec. ¬{} ⊢c s"
  shows "par_complsst [] Sec"
⟨proof⟩

lemma par_complsst_subset:
  assumes A: "par_complsst A Sec"
  and BA: "set B ⊆ set A"
  shows "par_complsst B Sec"
⟨proof⟩

lemma par_complsst_split:
  assumes "par_complsst (A@B) Sec"
  shows "par_complsst A Sec" "par_complsst B Sec"
⟨proof⟩

lemma par_complsst_proj:
  assumes "par_complsst A Sec"
  shows "par_complsst (proj n A) Sec"

```

<proof>

lemma *par_comp_{l_{sst}}_dual_{l_{sst}}*:
 assumes *A*: "par_comp_{l_{sst}} *A* *S*"
 shows "par_comp_{l_{sst}} (dual_{l_{sst}} *A*) *S*"
<proof>

lemma *par_comp_{l_{sst}}_subst*:
 assumes *A*: "par_comp_{l_{sst}} *A* *S*"
 and δ : "wt_{subst} δ " "wf_{trms} (subst_range δ)" "subst_domain $\delta \cap \text{bvars}_{l_{sst}} A = \{\}$ "
 shows "par_comp_{l_{sst}} (*A* ·_{l_{sst}} δ) *S*"
<proof>

lemma *wf_pair_negchecks_map'*:
 assumes "wf_{st} *X* (unlabel *A*)"
 shows "wf_{st} *X* (unlabel (map ($\lambda G. (i, \forall Y (\forall \neq: (F@G)_{st})) M@A$)))"
<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 ($\lambda d. (i, \langle \text{check: (pair (t,s)) \doteq (pair (snd d)) \rangle_{st})$) *Di*)@
 (map ($\lambda d. (i, \forall [] (\forall \neq: [(pair (t,s), pair (snd d))])_{st})$) [*d* ← dbproj *i* *D*. *d* ∉ set *Di*])@*A*))"
<proof>

lemma *trms_{sst}_setops_{sst}_wt_instance_ex*:
 defines "*M* ≡ $\lambda A. \text{trms}_{l_{sst}} A \cup \text{pair} \ ` \ \text{setops}_{sst} (\text{unlabel } A)$ "
 assumes *B*: " $\forall b \in \text{set } B. \exists a \in \text{set } A. \exists \delta. b = a \cdot_{l_{sstp}} \delta \wedge \text{wt}_{subst} \delta \wedge \text{wf}_{trms} (\text{subst_range } \delta)$ "
 shows " $\forall t \in M B. \exists s \in M A. \exists \delta. t = s \cdot \delta \wedge \text{wt}_{subst} \delta \wedge \text{wf}_{trms} (\text{subst_range } \delta)$ "
<proof>

lemma *setops_{l_{sst}}_wt_instance_ex*:
 assumes *B*: " $\forall b \in \text{set } B. \exists a \in \text{set } A. \exists \delta. b = a \cdot_{l_{sstp}} \delta \wedge \text{wt}_{subst} \delta \wedge \text{wf}_{trms} (\text{subst_range } \delta)$ "
 shows " $\forall p \in \text{setops}_{l_{sst}} B. \exists q \in \text{setops}_{l_{sst}} A. \exists \delta. \text{fst } p = \text{fst } q \wedge \text{snd } p = \text{snd } q \cdot_p \delta \wedge \text{wt}_{subst} \delta \wedge \text{wf}_{trms} (\text{subst_range } \delta)$ "
<proof>

lemma *deduct_proj_priv_term_prefix_ex_stateful*:
 assumes *A*: "ik_{sst} (proj_unl 1 *A*) ·_{set} *I* ⊢ *t*"
 and *t*: "¬{ } ⊢_c *t*"
 shows "∃ *B* *k* *s*. (*k* = * ∨ *k* = ln 1) ∧ prefix (*B*@[(*k*, receive(*s*))] *A*) ∧
 declassified_{l_{sst}} ((*B*@[(*k*, receive(*s*))]) *I*) = declassified_{l_{sst}} *A* *I* ∧
 ik_{sst} (proj_unl 1 (*B*@[(*k*, receive(*s*))])) = ik_{sst} (proj_unl 1 *A*)"
<proof>

lemma *constr_sem_stateful_proj_priv_term_prefix_obtain*:
 assumes *A'*: "prefix *A'* *A*" "constr_sem_stateful \mathcal{I}_τ (proj_unl *n* *A'*@[send([*t*])])"
 and *t*: "*t* ∈ Sec - declassified_{l_{sst}} *A'* \mathcal{I}_τ " "¬{ } ⊢_c *t*" "*t* · $\mathcal{I}_\tau = t$ "
 obtains *B* *k'* *s* where
 "*k'* = * ∨ *k'* = ln *n*" "prefix *B* *A'*" "suffix [(*k'*, receive(*s*))] *B*"
 "declassified_{l_{sst}} *B* $\mathcal{I}_\tau = \text{declassified}_{l_{sst}} A' \mathcal{I}_\tau$ "
 "ik_{l_{sst}} (proj *n* *B*) = ik_{l_{sst}} (proj *n* *A')*"
 "constr_sem_stateful \mathcal{I}_τ (proj_unl *n* *B*@[send([*t*])])"
 "prefix (proj *n* *B*) (proj *n* *A*)" "suffix [(*k'*, receive(*s*))] (proj *n* *B*)"
 "*t* ∈ Sec - declassified_{l_{sst}} (proj *n* *B*) \mathcal{I}_τ "
<proof>

lemma *constr_sem_stateful_star_proj_no_leakage*:
 fixes Sec *P* *lbls* *k*
 defines "no_leakage ≡ $\lambda A. \# \mathcal{I}_\tau B s$."

```

    prefix B A ∧ s ∈ Sec - declassifiedlsst B Iτ ∧ Iτ ⊨s (unlabel B@[send([s])])"
  assumes Sec: "ground Sec"
    and A: "∀(l,a) ∈ set A. l = ★"
  shows "no_leakage A"
⟨proof⟩

end

```

6.2.4 Lemmata: Properties of the Constraint Translation Function

```

context labeled_stateful_typed_model
begin

```

```

lemma tr_par_labeled_rcv_iff:

```

```

  "B ∈ set (trpc A D) ⇒ (i, receive(t)st) ∈ set B ⇔ (i, receive(t)) ∈ set A"
⟨proof⟩

```

```

lemma tr_par_declassified_eq:

```

```

  "B ∈ set (trpc A D) ⇒ declassifiedlst B I = declassifiedlsst A I"
⟨proof⟩

```

```

lemma tr_par_ik_eq:

```

```

  assumes "B ∈ set (trpc A D)"
  shows "ikst (unlabel B) = iksst (unlabel A)"
⟨proof⟩

```

```

lemma tr_par_deduct_iff:

```

```

  assumes "B ∈ set (trpc A D)"
  shows "ikst (unlabel B) ·set I ⊢ t ⇔ iksst (unlabel A) ·set I ⊢ t"
⟨proof⟩

```

```

lemma tr_par_vars_subset:

```

```

  assumes "A' ∈ set (trpc A D)"
  shows "fvlst A' ⊆ fvsst (unlabel A) ∪ fvpairs (unlabel D)" (is ?P)
  and "bvarslst A' ⊆ bvarssst (unlabel A)" (is ?Q)
⟨proof⟩

```

```

lemma tr_par_vars_disj:

```

```

  assumes "A' ∈ set (trpc A D)" "fvpairs (unlabel D) ∩ bvarssst (unlabel A) = {}"
  and "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}"
  shows "fvlst A' ∩ bvarslst A' = {}"
⟨proof⟩

```

```

lemma tr_par_trms_subset:

```

```

  assumes "A' ∈ set (trpc A D)"
  shows "trmslst A' ⊆ trmssst (unlabel A) ∪ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D"
⟨proof⟩

```

```

lemma tr_par_wf_trms:

```

```

  assumes "A' ∈ set (trpc A [])" "wftrms (trmssst (unlabel A))"
  shows "wftrms (trmslst A)"
⟨proof⟩

```

```

lemma tr_par_wf':

```

```

  assumes "fvpairs (unlabel D) ∩ bvarssst (unlabel A) = {}"
  and "fvpairs (unlabel D) ⊆ X"
  and "wf'sst X (unlabel A)" "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}"
  and "A' ∈ set (trpc A D)"
  shows "wf'lst X A'"
⟨proof⟩

```

```

lemma tr_par_wf:

```

```

  assumes "A' ∈ set (trpc A [])"

```

```

    and "wfsst (unlabel A)"
    and "wftrms (trmssst A)"
  shows "wflst {} A'"
    and "wftrms (trmssst A)"
    and "fvlst A' ∩ bvarssst A' = {}"
⟨proof⟩

lemma tr_par_proj:
  assumes "B ∈ set (trpc A D)"
  shows "proj n B ∈ set (trpc (proj n A) (proj n D))"
⟨proof⟩

lemma tr_par_preserves_par_comp:
  assumes "par_compsst A Sec" "A' ∈ set (trpc A [])"
  shows "par_comp A' Sec"
⟨proof⟩

lemma tr_preserves_receives:
  assumes "E ∈ set (trpc F D)" "(l, receive(t)) ∈ set F"
  shows "(l, receive(t)st) ∈ set E"
⟨proof⟩

lemma tr_preserves_last_receive:
  assumes "E ∈ set (trpc F D)" "suffix [(l, receive(t)st)] E"
  shows "∃ G. suffix ((l, receive(t))#G) F ∧ list_all (Not ∘ is_Receive ∘ snd) G"
    (is "∃ G. ?P G F ∧ ?Q G")
⟨proof⟩

lemma tr_leaking_prefix_exists:
  assumes "A' ∈ set (trpc A [])" "prefix B A'" "ikst (proj_unl n B) ·set I ⊢ t"
  shows "∃ C D. prefix C B ∧ prefix D A ∧ C ∈ set (trpc D []) ∧ (ikst (proj_unl n C) ·set I ⊢ t) ∧
    (¬{} ⊢c t → (∃ l s G. suffix ((l, receive(s))#G) D ∧ list_all (Not ∘ is_Receive ∘
    snd) G))"
⟨proof⟩

end

context labeled_stateful_typing
begin

lemma tr_par_tfrsstp:
  assumes "A' ∈ set (trpc A D)" "list_all tfrsstp (unlabel A)"
  and "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}" (is "?P0 A D")
  and "fvpairs (unlabel D) ∩ bvarssst (unlabel A) = {}" (is "?P1 A D")
  and "∀ t ∈ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D.
    ∀ t' ∈ pair ` setopssst (unlabel A) ∪ pair ` snd ` set D.
    (∃ δ. Unifier δ t t') → Γ t = Γ t'" (is "?P3 A D")
  shows "list_all tfrstp (unlabel A)"
⟨proof⟩

lemma tr_par_tfr:
  assumes "A' ∈ set (trpc A [])" and "tfrsst (unlabel A)"
  and "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}"
  shows "tfrst (unlabel A)"
⟨proof⟩

lemma tr_par_preserves_typing_cond:
  assumes "par_compsst A Sec" "typing_condsst (unlabel A)" "A' ∈ set (trpc A [])"
  shows "typing_cond (unlabel A)"
⟨proof⟩

end

```

6.2.5 Theorem: Semantic Equivalence of Translation

```
context labeled_stateful_typed_model
begin
```

```
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/correctness.

```
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⟨ts⟩)#A) D = map ((#) (i, send⟨ts⟩st)) (tr'_pc A D)"
| "tr'_pc ((i, receive⟨ts⟩)#A) D = map ((#) (i, receive⟨ts⟩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) ∈ setopsl_sst A ∪ set D.
    ∀ (j, q) ∈ setopsl_sst 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) ∈ setopsl_sst A ∪ set D. ∀ (j, q) ∈ setopsl_sst 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) ∩ bvarssst (unlabel A) = {}" (is "?R4 A D")
  and "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}" (is "?R5 A D")
  shows "(∃ B ∈ set (trpc 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) ∩ bvarssst (unlabel A) = {}"
  and "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}" "ground M"
  and "∀ (i, p) ∈ setopsl_sst A ∪ set D. ∀ (j, q) ∈ setopsl_sst A ∪ set D.
    (∃ δ. Unifier δ (pair p) (pair q)) ⟶ i = j" (is "?R A D")
  and I: "interpretationsubst 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) ∩ bvarssst (unlabel A) = {}"
  and "fvsst (unlabel A) ∩ bvarssst (unlabel A) = {}" "ground M"
  and "∀(i,p) ∈ setopslst A ∪ set D. ∀(j,q) ∈ setopslst A ∪ set D.
    (∃δ. Unifier δ (pair p) (pair q)) → i = j"
  and I: "interpretationsubst I"
  shows "[M; set (unlabel D) ·pset I; unlabel A]s I ↔ (∃B ∈ set (trpc A D). [M; unlabel B]d I)"
  (is "?P ↔ ?Q")
⟨proof⟩

end

end

```

6.2.6 Theorem: The Stateful Compositionality Result, on the Constraint Level

```

theorem (in labeled_stateful_typed_model) par_comp_constr_stateful_typed:
  assumes A: "par_complst A Sec" "fvlst A ∩ bvarslst A = {}"
  and I: "I ⊨s unlabel A" "interpretationsubst I" "wtsubst I" "wftrms (subst_range I)"
  shows "(∀n. I ⊨s proj_unl n A) ∨
    (∃A' l' ts. prefix A' A ∧ suffix [(l', receive(ts))] A' ∧ (A' leaks Sec under I))"
⟨proof⟩

```

```

theorem (in labeled_stateful_typing) par_comp_constr_stateful:
  assumes A: "par_complst A Sec" "typing_condsst (unlabel A)"
  and I: "I ⊨s unlabel A" "interpretationsubst I"
  shows "∃Iτ. interpretationsubst Iτ ∧ wtsubst Iτ ∧ wftrms (subst_range Iτ) ∧ (Iτ ⊨s unlabel A) ∧
    ((∀n. Iτ ⊨s proj_unl n A) ∨
    (∃A' l' ts. prefix A' A ∧ suffix [(l', receive(ts))] A' ∧ (A' leaks Sec under
Iτ)))"
⟨proof⟩

```

6.2.7 Theorem: The Stateful Compositionality Result, on the Protocol Level

```

context labeled_stateful_typing
begin

```

```

context
begin

```

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.

```

private definition wflst :: "('fun, 'var, 'lbl) labeled_strand set ⇒ bool" where
  "wflst S ≡ (∀A ∈ S. wflst {} A) ∧ (∀A ∈ S. ∀A' ∈ S. fvlst A ∩ bvarslst A' = {})"

```

```

private definition wflst' ::
  "('fun, 'var, 'lbl) labeled_strand set ⇒ ('fun, 'var, 'lbl) labeled_strand ⇒ bool"

```

```

where

```

```

  "wflst' S A ≡ (∀A' ∈ S. wfst (wfrestrictedvarslst A) (unlabel A')) ∧
    (∀A' ∈ S. ∀A'' ∈ S. fvlst A' ∩ bvarslst A'' = {}) ∧
    (∀A' ∈ S. fvlst A' ∩ bvarslst A = {}) ∧
    (∀A' ∈ S. fvlst A ∩ bvarslst A' = {})"

```

```

private definition typing_cond_prot where

```

```

  "typing_cond_prot P ≡
    wflst P ∧
    tfrset (∪(trmslst ` P)) ∧
    wftrms (∪(trmslst ` P)) ∧

```

$$(\forall \mathcal{A} \in \mathcal{P}. \text{list_all } \text{tfr}_{stp} (\text{unlabel } \mathcal{A})) \wedge$$

$$\text{Ana_invar_subst } (\bigcup (\text{ik}_{st} \setminus \text{unlabel } \setminus \mathcal{P}) \cup \bigcup (\text{assignment_rhs}_{st} \setminus \text{unlabel } \setminus \mathcal{P}))"$$

private definition *par_comp_prot* where

```
"par_comp_prot P Sec ≡
  (∀ l1 l2. l1 ≠ l2 →
    GSMP_disjoint (⋃ A ∈ P. trms_projlst l1 A) (⋃ A ∈ P. trms_projlst l2 A) Sec) ∧
  ground Sec ∧ (∀ s ∈ Sec. ¬{ } ⊢c s) ∧
  typing_cond_prot P"
```

Lemmata: Labeled Protocols

private lemma *wflsts_eqs_wflsts'*: "*wflsts S = wflsts' 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

private definition *component_prot* where

```
"component_prot n P ≡ (∀ l ∈ P. ∀ s ∈ set l. has_LabelN n s ∨ has_LabelS s)"
```

private definition *composed_prot* where

```
"composed_prot Pi ≡ {A. ∀ n. proj n A ∈ Pi n}"
```

private definition *component_secure_prot* where

```
"component_secure_prot n P Sec attack ≡ (∀ A ∈ P. suffix [(ln n, Send1 (Fun attack []))]) A →
  (∀ Iτ. (interpretationsubst Iτ ∧ wtsubst Iτ ∧ wftrms (subst_range Iτ)) →
    ¬(Iτ ⊢ ⟨proj_unl n A⟩) ∧
    (∀ A'. prefix A' A →
      (∀ t ∈ Sec-declassifiedlst A' Iτ. ¬(Iτ ⊢ ⟨proj_unl n A'@[Send1 t]⟩))))))"
```

private definition *component_leaks* where

```
"component_leaks n A Sec ≡ (∃ A' Iτ. interpretationsubst Iτ ∧ wtsubst Iτ ∧ wftrms (subst_range Iτ)
  ∧
  prefix A' A ∧ (∃ t ∈ Sec - declassifiedlst A' Iτ. (Iτ ⊢ ⟨proj_unl n A'@[Send1 t]⟩)))"
```

private definition *unsat* where

```
"unsat A ≡ (∀ I. interpretationsubst I → ¬(I ⊢ ⟨unlabel A⟩))"
```

private 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, Send1 (Fun attack []))]) A →
    unsat A ∨ (∃ m. n ≠ m ∧ component_leaks m A Sec)"
```

<proof>

Theorem: Parallel Compositionality for Stateful Protocols

private abbreviation *wflsst* where

```
"wflsst 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.

private definition *wflsst*: "*(fun, 'var, 'lbl) labeled_stateful_strand set ⇒ bool*" where

```
"wflsst S ≡ (∀ A ∈ S. wflst { } A) ∧ (∀ A ∈ S. ∀ A' ∈ S. fvlst A ∩ bvarslst A' = { })"
```

private definition *wflsst'*:

```

("fun, 'var, 'lbl) labeled_stateful_strand set  $\Rightarrow$  ('fun, 'var, 'lbl) labeled_stateful_strand  $\Rightarrow$  bool"
where
  "wflssst' S A  $\equiv$  ( $\forall A' \in S. wf'_{sst} (wfrestrictedvars_{lssst} A) (unlabel A')$ )  $\wedge$ 
    ( $\forall A' \in S. \forall A'' \in S. fv_{lssst} A' \cap bvars_{lssst} A'' = \{\}$ )  $\wedge$ 
    ( $\forall A' \in S. fv_{lssst} A' \cap bvars_{lssst} A = \{\}$ )  $\wedge$ 
    ( $\forall A' \in S. fv_{lssst} A \cap bvars_{lssst} A' = \{\}$ )"

private definition typing_cond_prot_stateful where
  "typing_cond_prot_stateful P  $\equiv$ 
    wflssst P  $\wedge$ 
    tfrset ( $\bigcup (trms_{lssst} \ ` P) \cup pair \ ` \bigcup (setops_{sst} \ ` unlabel \ ` P)$ )  $\wedge$ 
    wfttrms ( $\bigcup (trms_{lssst} \ ` P)$ )  $\wedge$ 
    ( $\forall S \in P. list\_all\ tfr_{sstp} (unlabel S)$ )"

private definition par_comp_prot_stateful where
  "par_comp_prot_stateful P Sec  $\equiv$ 
    ( $\forall l1\ l2. l1 \neq l2 \rightarrow$ 
      GSMP_disjoint ( $\bigcup A \in P. trms_{sst} (proj\_unl\ l1\ A) \cup pair \ ` \ setops_{sst} (proj\_unl\ l1\ A)$ )
        ( $\bigcup A \in P. trms_{sst} (proj\_unl\ l2\ A) \cup pair \ ` \ setops_{sst} (proj\_unl\ l2\ A)$ ) Sec)  $\wedge$ 
    ground Sec  $\wedge$  ( $\forall s \in Sec. \neg \{\} \vdash_c s$ )  $\wedge$ 
    ( $\forall (i,p) \in \bigcup A \in P. setops_{lssst} A. \forall (j,q) \in \bigcup A \in P. setops_{lssst} A.
      (\exists \delta. Unifier\ \delta (pair\ p) (pair\ q)) \rightarrow i = j$ )  $\wedge$ 
    typing_cond_prot_stateful P"

private definition component_secure_prot_stateful where
  "component_secure_prot_stateful n P Sec attack  $\equiv$ 
    ( $\forall A \in P. suffix [(ln\ n, Send [Fun\ attack\ []])] A \rightarrow$ 
      ( $\forall \mathcal{I}_\tau. (interpretation_{subst} \mathcal{I}_\tau \wedge wt_{subst} \mathcal{I}_\tau \wedge wf_{trms} (subst\_range \mathcal{I}_\tau)) \rightarrow$ 
        ( $\neg (\mathcal{I}_\tau \models_s (proj\_unl\ n\ A)) \wedge$ 
          ( $\forall A'. prefix\ A'\ A \rightarrow$ 
            ( $\forall t \in Sec\text{-declassified}_{lssst} A' \mathcal{I}_\tau. \neg (\mathcal{I}_\tau \models_s (proj\_unl\ n\ A'@[Send\ [t]]))$ ))))))"

private definition component_leaks_stateful where
  "component_leaks_stateful n A Sec  $\equiv$ 
    ( $\exists A' \mathcal{I}_\tau. interpretation_{subst} \mathcal{I}_\tau \wedge wt_{subst} \mathcal{I}_\tau \wedge wf_{trms} (subst\_range \mathcal{I}_\tau) \wedge prefix\ A'\ A \wedge$ 
      ( $\exists t \in Sec\text{-declassified}_{lssst} A' \mathcal{I}_\tau. (\mathcal{I}_\tau \models_s (proj\_unl\ n\ A'@[Send\ [t]]))$ ))"

private definition unsat_stateful where
  "unsat_stateful A  $\equiv$  ( $\forall \mathcal{I}. interpretation_{subst} \mathcal{I} \rightarrow \neg (\mathcal{I} \models_s unlabel\ A)$ )"

private lemma wflssst_eqs_wflssst': "wflssst S = wflssst' S []"
<proof> lemma par_comp_prot_impl_par_comp_stateful:
  assumes "par_comp_prot_stateful P Sec" "A  $\in$  P"
  shows "par_complssst A Sec"
<proof> lemma typing_cond_prot_impl_typing_cond_stateful:
  assumes "typing_cond_prot_stateful P" "A  $\in$  P"
  shows "typing_condsst (unlabel A)"
<proof> theorem par_comp_constr_prot_stateful:
  assumes P: "P = composed_prot Pi" "par_comp_prot_stateful P Sec" " $\forall n. component\_prot\ n (Pi\ n)$ "
  and left_secure: "component_secure_prot_stateful n (Pi n) Sec attack"
  shows " $\forall A \in P. suffix [(ln\ n, Send [Fun\ attack\ []])] A \rightarrow$ 
    unsat_stateful A  $\vee$  ( $\exists m. n \neq m \wedge component\_leaks\_stateful\ m\ A\ Sec$ )"
<proof>

end

end

```

6.2.8 Automated Compositionality Conditions

definition comp_GSMP_disjoint where

```

"comp_GSMP_disjoint public arity Ana  $\Gamma$  A' B' A B C  $\equiv$ 
  let B $\delta$  = B .set var_rename (max_var_set (fvset A))

```

```

in has_all_wt_instances_of  $\Gamma$  A' A  $\wedge$ 
   has_all_wt_instances_of  $\Gamma$  B' B $\delta$   $\wedge$ 
   finite_SMP_representation arity Ana  $\Gamma$  A  $\wedge$ 
   finite_SMP_representation arity Ana  $\Gamma$  B $\delta$   $\wedge$ 
   ( $\forall t \in A. \forall s \in B\delta. \Gamma t = \Gamma s \wedge \text{mgu } t s \neq \text{None} \rightarrow$ 
    (intruder_synth' public arity {} t)  $\vee$  ( $\exists u \in C. \text{is\_wt\_instance\_of\_cond } \Gamma t u$ ))"

```

definition `comp_par_comp'lsst` where

```

"comp_par_comp'lsst public arity Ana  $\Gamma$  pair_fun A M C  $\equiv$ 
 wftrms' arity C  $\wedge$ 
 ( $\forall (i,p) \in \text{setops}_{l_{sst}} A. \forall (j,q) \in \text{setops}_{l_{sst}} A. \text{if } i = j \text{ then True else}$ 
  (let s = pair' pair_fun p; t = pair' pair_fun q
   in mgu s (t · var_rename (max_var s)) = None))"

```

definition `comp_par_complsst` where

```

"comp_par_complsst public arity Ana  $\Gamma$  pair_fun A M C  $\equiv$ 
 let L = remdups (map (the_LabelN  $\circ$  fst) (filter (Not  $\circ$  has_Labels) A));
     MPO =  $\lambda B. \text{trms}_{sst} B \cup (\text{pair}' \text{pair\_fun}) \setminus \text{setops}_{sst} B$ ;
     pr =  $\lambda l. \text{MPO} (\text{proj\_unl } l A)$ 
 in length L > 1  $\wedge$ 
     comp_par_comp'lsst public arity Ana  $\Gamma$  pair_fun A M C  $\wedge$ 
     wftrms' arity (MPO (unlabel A))  $\wedge$ 
     ( $\forall i \in \text{set } L. \forall j \in \text{set } L. \text{if } i = j \text{ then True else}$ 
      comp_GSMP_disjoint public arity Ana  $\Gamma$  (pr i) (pr j) (M i) (M j) C)"

```

lemma `comp_par_complsstI:`

```

fixes pair_fun A MPO pr
defines "MPO  $\equiv \lambda B. \text{trms}_{sst} B \cup (\text{pair}' \text{pair\_fun}) \setminus \text{setops}_{sst} B$ "
       and "pr  $\equiv \lambda l. \text{MPO} (\text{proj\_unl } l A)$ "
assumes L_def: "L = remdups (map (the_LabelN  $\circ$  fst) (filter (Not  $\circ$  has_Labels) A))"
       and L_gt: "length L > 1"
       and cpc': "comp_par_comp'lsst public arity Ana  $\Gamma$  pair_fun A M C"
       and MPO_wf: "wftrms' arity (MPO (unlabel A))"
       and GSMP_disj: " $\forall i \in \text{set } L. \forall j \in \text{set } L. \text{if } i = j \text{ then True else}$ 
                      comp_GSMP_disjoint public arity Ana  $\Gamma$  (pr i) (pr j) (M i) (M j) C"
shows "comp_par_complsst public arity Ana  $\Gamma$  pair_fun A M C"
<proof>

```

lemma `comp_par_complsstI':`

```

fixes pair_fun A MPO pr Ms
defines "MPO  $\equiv \lambda B. \text{trms}_{sst} B \cup (\text{pair}' \text{pair\_fun}) \setminus \text{setops}_{sst} B$ "
       and "pr  $\equiv \lambda l. \text{MPO} (\text{proj\_unl } l A)$ "
       and "M  $\equiv \lambda l. \text{case find } (=(\text{fst } l \circ \text{fst})) \text{ Ms of Some } M \Rightarrow \text{set } (\text{snd } M) \mid \text{None} \Rightarrow \{\}$ "
assumes L_def: "map fst Ms = remdups (map (the_LabelN  $\circ$  fst) (filter (Not  $\circ$  has_Labels) A))"
       and L_gt: "length (map fst Ms) > 1"
       and cpc': "comp_par_comp'lsst public arity Ana  $\Gamma$  pair_fun A M C"
       and MPO_wf: "wftrms' arity (MPO (unlabel A))"
       and GSMP_disj: " $\forall i \in \text{set } (\text{map fst Ms}). \forall j \in \text{set } (\text{map fst Ms}). \text{if } i = j \text{ then True else}$ 
                      comp_GSMP_disjoint public arity Ana  $\Gamma$  (pr i) (pr j) (M i) (M j) C"
shows "comp_par_complsst public arity Ana  $\Gamma$  pair_fun A M C"
<proof>

```

locale `labeled_stateful_typed_model'` =

```

  labeled_typed_model' arity public Ana  $\Gamma$  label_witness1 label_witness2
+ stateful_typed_model' arity public Ana  $\Gamma$  Pair
  for arity::"'fun  $\Rightarrow$  nat"
  and public::"'fun  $\Rightarrow$  bool"
  and Ana::"('fun, ('fun, 'atom::finite) term_type  $\times$  nat)) term
             $\Rightarrow$  (('fun, ('fun, 'atom) term_type  $\times$  nat)) term list
             $\times$  ('fun, ('fun, 'atom) term_type  $\times$  nat)) term list)"
  and  $\Gamma$ ::"('fun, ('fun, 'atom) term_type  $\times$  nat)) term  $\Rightarrow$  ('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: "wftrms' arity A'" "wftrms' arity B'"
    and C_wf: "wftrms' arity C'"
    and AB'_disj: "comp_GSMP_disjoint public arity Ana Γ A' B' A B C'"
  shows "GSMP_disjoint A' B' (f 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_complsst public arity Ana Γ Pair A M C'"
  shows "par_complsst A (f C - {m. {} ⊢c m})"
⟨proof⟩

end

locale labeled_stateful_typing' =
  labeled_stateful_typed_model' arity public Ana Γ Pair label_witness1 label_witness2
+ stateful_typing_result' 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"
  and label_witness1::"lbl"
  and label_witness2::"lbl"
begin

sublocale labeled_stateful_typing
⟨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_complsst public arity Ana Γ Pair A M C'"
    and B: "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·lsst δ ∧ wtsubst δ ∧ wftrms (subst_range δ)"
    (is "∀b ∈ set B. ∃a ∈ set A. ∃δ. b = a ·lsst δ ∧ ?D δ")
  shows "par_complsst B (f 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

```
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"
```

7 Examples

```

fun public::"ex_fun  $\Rightarrow$  bool" where
  "public (privkey _) = False"
| "public _ = True"

fun Anacrypt::"ex_term list  $\Rightarrow$  (ex_term list  $\times$  ex_term list)" where
  "Anacrypt [Fun pub [k],m] = ([k], [m])"
| "Anacrypt _ = ([], [])"

fun Anasign::"ex_term list  $\Rightarrow$  (ex_term list  $\times$  ex_term list)" where
  "Anasign [k,m] = ([], [m])"
| "Anasign _ = ([], [])"

fun Ana::"ex_term  $\Rightarrow$  (ex_term list  $\times$  ex_term list)" where
  "Ana (Fun crypt T) = Anacrypt 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) = Anasign T"
| "Ana (Fun x509 T) = ([], T)"
| "Ana _ = ([], [])"

```

7.1.2 TLS example: Locale interpretation

```

lemma assm1:
  "Ana t = (K,M)  $\Longrightarrow$  fvset (set K)  $\subseteq$  fv t"
  "Ana t = (K,M)  $\Longrightarrow$  ( $\bigwedge$ g S'. Fun g S'  $\sqsubseteq$  t  $\Longrightarrow$  length S' = arity g)
   $\Longrightarrow$  k  $\in$  set K  $\Longrightarrow$  Fun f T'  $\sqsubseteq$  k  $\Longrightarrow$  length T' = arity f"
  "Ana t = (K,M)  $\Longrightarrow$  K  $\neq$  []  $\vee$  M  $\neq$  []  $\Longrightarrow$  Ana (t  $\cdot$   $\delta$ ) = (K  $\cdot$ list  $\delta$ , M  $\cdot$ list  $\delta$ )"
<proof>

```

```

lemma assm2: "Ana (Fun f T) = (K, M)  $\Longrightarrow$  set M  $\subseteq$  set T"
<proof>

```

```

lemma assm6: "0 < arity f  $\Longrightarrow$  public f" <proof>

```

```

global_interpretation im: intruder_model arity public Ana

```

```

  defines wftrm = "im.wftrm"
  and wftrms = "im.wftrms"
<proof>

```

7.1.3 TLS Example: Typing function

```

definition  $\Gamma_v$ ::"ex_var  $\Rightarrow$  ex_type" where
  " $\Gamma_v$  v = (if ( $\forall$ t  $\in$  subterms (fst v). case t of
    (TComp f T)  $\Rightarrow$  arity f > 0  $\wedge$  arity f = length T
    | _  $\Rightarrow$  True)
    then fst v else TAtom Bot)"

```

```

fun  $\Gamma$ ::"ex_term  $\Rightarrow$  ex_type" where
  " $\Gamma$  (Var v) =  $\Gamma_v$  v"
| " $\Gamma$  (Fun (privkey _) _) = TAtom PrivKey"
| " $\Gamma$  (Fun changeCipher _) = TAtom PubConst"
| " $\Gamma$  (Fun serverHelloDone _) = TAtom PubConst"
| " $\Gamma$  (Fun (pubconst  $\tau$ ) _) = TAtom  $\tau$ "
| " $\Gamma$  (Fun f T) = TComp f (map  $\Gamma$  T)"

```

7.1.4 TLS Example: Locale interpretation (typed model)

```

lemma assm7: "arity c = 0  $\Longrightarrow$   $\exists$ a.  $\forall$ X.  $\Gamma$  (Fun c X) = TAtom a" <proof>

```

```

lemma assm8: "0 < arity f  $\Longrightarrow$   $\Gamma$  (Fun f X) = TComp f (map  $\Gamma$  X)" <proof>

```

lemma *assm9*: "infinite {c. Γ (Fun c []) = TAtom a \wedge public c}"

<proof>

lemma *assm10*:

assumes "TComp f T \sqsubseteq Γ (Var x)"

shows "arity f > 0"

<proof>

lemma *assm11*: "im.wf_{term} (Γ (Var x))"

<proof>

lemma *assm12*: " Γ (Var (τ , n)) = Γ (Var (τ , m))"

<proof>

lemma *Ana_const*: "arity c = 0 \implies Ana (Fun c T) = ([], [])"

<proof>

lemma *Ana_keys_subterm*: "Ana t = (K,T) \implies k \in set K \implies k \sqsubseteq t"

<proof>

global interpretation *tm*: typed_model' arity public Ana Γ

<proof>

7.1.5 TLS example: Proving type-flaw resistance

abbreviation $\Gamma_v_clientHello$ where

" $\Gamma_v_clientHello \equiv$

TComp clientHello [TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce]"

abbreviation $\Gamma_v_serverHello$ where

" $\Gamma_v_serverHello \equiv$

TComp serverHello [TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce, TAtom Nonce]"

abbreviation Γ_v_pub where

" $\Gamma_v_pub \equiv$ TComp pub [TAtom PrivKey]"

abbreviation Γ_v_x509 where

" $\Gamma_v_x509 \equiv$ TComp x509 [TAtom Agent, Γ_v_pub]"

abbreviation Γ_v_sign where

" $\Gamma_v_sign \equiv$ TComp sign [TAtom PrivKey, Γ_v_x509]"

abbreviation $\Gamma_v_serverCert$ where

" $\Gamma_v_serverCert \equiv$ TComp serverCert [Γ_v_sign]"

abbreviation $\Gamma_v_pmsForm$ where

" $\Gamma_v_pmsForm \equiv$ TComp pmsForm [TAtom SymKey]"

abbreviation Γ_v_crypt where

" $\Gamma_v_crypt \equiv$ TComp crypt [Γ_v_pub , $\Gamma_v_pmsForm$]"

abbreviation $\Gamma_v_clientKeyExchange$ where

" $\Gamma_v_clientKeyExchange \equiv$

TComp clientKeyExchange [Γ_v_crypt]"

abbreviation $\Gamma_v_HSMsigs$ where

" $\Gamma_v_HSMsigs \equiv$ TComp concat [

$\Gamma_v_clientHello$,

$\Gamma_v_serverHello$,

$\Gamma_v_serverCert$,

TAtom PubConst,

$\Gamma_v_clientKeyExchange]$ "

```

abbreviation "T1 n ≡ Var (TAtom Nonce,n)"
abbreviation "T2 n ≡ Var (TAtom Nonce,n)"
abbreviation "RA n ≡ Var (TAtom Nonce,n)"
abbreviation "RB 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 "Prca n ≡ Var (TAtom PrivKey,n)"
abbreviation "PMS n ≡ Var (TAtom SymKey,n)"
abbreviation "PB n ≡ Var (TComp pub [TAtom PrivKey],n)"
abbreviation "HSMsigs n ≡ Var (Γv_HSMsigs,n)"

```

Defining the over-approximation set

```

abbreviation clientHellotrm where
  "clientHellotrm ≡ Fun clientHello [T1 0, RA 1, S 2, Cipher 3, Comp 4]"

abbreviation serverHellotrm where
  "serverHellotrm ≡ Fun serverHello [T2 0, RB 1, S 2, Cipher 3, Comp 4]"

abbreviation serverCerttrm where
  "serverCerttrm ≡ Fun serverCert [Fun sign [Prca 0, Fun x509 [B 1, PB 2]]]"

abbreviation serverHelloDonetrm where
  "serverHelloDonetrm ≡ Fun serverHelloDone []"

abbreviation clientKeyExchangetrm where
  "clientKeyExchangetrm ≡ Fun clientKeyExchange [Fun crypt [PB 0, Fun pmsForm [PMS 1]]]"

abbreviation changeCiphertrm where
  "changeCiphertrm ≡ Fun changeCipher []"

abbreviation finishedtrm where
  "finishedtrm ≡ Fun finished [Fun prfun [
    Fun clientFinished [
      Fun prfun [Fun master [PMS 0, RA 1, RB 2]],
      RA 3, RB 4, Fun hash [HSMsigs 5]
    ]
  ]]"

definition MTLS::"ex_term list" where
  "MTLS ≡ [
    clientHellotrm,
    serverHellotrm,
    serverCerttrm,
    serverHelloDonetrm,
    clientKeyExchangetrm,
    changeCiphertrm,
    finishedtrm
  ]"

```

7.1.6 Theorem: The TLS handshake protocol is type-flaw resistant

```

theorem "tm.tfrset (set MTLS)"
⟨proof⟩

end

```

7.2 The Keyserver Example

```
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
```

7 Examples

```

"public (pubkey _) = False"
| "public encodingsecret = False"
| "public _ = True"

fun Anacrypt::"ex_term list ⇒ (ex_term list × ex_term list)" where
  "Anacrypt [k,m] = ([Fun invkey [Fun encodingsecret [], k]], [m])"
| "Anacrypt _ = ([], [])"

fun Anasign::"ex_term list ⇒ (ex_term list × ex_term list)" where
  "Anasign [k,m] = ([], [m])"
| "Anasign _ = ([], [])"

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) = Anasign T"
| "Ana (Fun crypt T) = Anacrypt T"
| "Ana _ = ([], [])"

```

Keyserver example: Locale interpretation

```

lemma assm1:
  "Ana t = (K,M) ⇒ fvset (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 wftrm = "im.wftrm"
⟨proof⟩

```

```

type_synonym ex_strand_step = "(ex_fun,ex_var) strand_step"
type_synonym ex_strand = "(ex_fun,ex_var) strand"

```

Typing function

```

definition Γv::"ex_var ⇒ ex_type" where
  "Γv v = (if (∀t ∈ subterms (fst v). case t of
    (TComp f T) ⇒ arity f > 0 ∧ arity f = length T
    | _ ⇒ True)
  then fst v else TAtom Bot)"

```

```

fun Γ::"ex_term ⇒ ex_type" where
  "Γ (Var v) = Γv v"
| "Γ (Fun (attack _) _) = TAtom Attack"
| "Γ (Fun (pubkey _) _) = TAtom Value"
| "Γ (Fun encodingsecret _) = TAtom PrivFunSec"
| "Γ (Fun (pubconst τ _) _) = TAtom τ"
| "Γ (Fun f T) = TComp f (map Γ T)"

```

Locale interpretation: typed model

```

lemma assm7: "arity c = 0 ⇒ ∃a. ∀X. Γ (Fun c X) = TAtom a" ⟨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  $\sqsubseteq$   $\Gamma$  t  $\implies$  arity f > 0"
<proof>
```

```
lemma assm11: "im.wftrm ( $\Gamma$  (Var x))"
<proof>
```

```
lemma assm12: " $\Gamma$  (Var ( $\tau$ , n)) =  $\Gamma$  (Var ( $\tau$ , m))"
<proof>
```

```
lemma Ana_const: "arity c = 0  $\implies$  Ana (Fun c T) = ([], [])"
<proof>
```

```
lemma Ana_subst': "Ana (Fun f T) = (K,M)  $\implies$  Ana (Fun f T  $\cdot$   $\delta$ ) = (K  $\cdot$ list  $\delta$ , M  $\cdot$ list  $\delta$ )"
<proof>
```

```
global_interpretation tm: typing_result arity public Ana  $\Gamma$ 
<proof>
```

Locale interpretation: labeled stateful typed model

```
global_interpretation stm: labeled_stateful_typing' arity public Ana  $\Gamma$  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  $\equiv$  Var (TAtom Value,n)"
abbreviation "A n  $\equiv$  Var (TAtom Agent,n)"
abbreviation "X n  $\equiv$  (TAtom Agent,n)"
```

```
abbreviation "ringset t  $\equiv$  Fun ring [Fun encodingsecret [], t]"
abbreviation "validset t t'  $\equiv$  Fun valid [Fun encodingsecret [], t, t']"
abbreviation "revokedset t t'  $\equiv$  Fun revoked [Fun encodingsecret [], t, t']"
abbreviation "eventsset  $\equiv$  Fun events [Fun encodingsecret []]"
```

```
abbreviation Sks :: "(ex_fun,ex_var) stateful_strand_step list" where
```

```
"Sks  $\equiv$  [
  insert(Fun (attack 0) [], eventsset),
  delete(PK 0, validset (A 0) (A 0)),
   $\forall$  (TAtom Agent,0) (PK 0 not in revokedset (A 0) (A 0)),
   $\forall$  (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.tfrsst Sks"
```

<proof>

7.2.3 Theorem: Type-flaw resistance of the keyserver examples from the ESORICS18 paper

abbreviation "signmsg t t' \equiv Fun sign [t, t']"

abbreviation "cryptmsg t t' \equiv Fun crypt [t, t']"

abbreviation "invkeymsg t \equiv Fun invkey [Fun encodingsecret [], t]"

abbreviation "updatemsg a b c d \equiv Fun update [a,b,c,d]"

abbreviation "pwmsg t t' \equiv Fun pw [t, t']"

abbreviation "beginauthset n t t' \equiv Fun (beginauth n) [Fun encodingsecret [], t, t']"

abbreviation "endauthset n t t' \equiv Fun (endauth n) [Fun encodingsecret [], t, t']"

abbreviation "pubkeyset t \equiv Fun pubkeys [Fun encodingsecret [], t]"

abbreviation "seenset t \equiv Fun seen [Fun encodingsecret [], t]"

declare [[coercion "Var::ex_var \Rightarrow ex_term"]]

declare [[coercion_enabled]]

definition S'_{ks} :: "ex_labeled_stateful_strand_step list" where

"S'_{ks} \equiv [

~~constructive_steps_from_the_list_provided_in_the_compile_steps_are_ignored~~

~~###/R11/1~~

~~<1, send[invkeymsg (PK 0)]>,
<*, <PK 0 in validset (A 0) (A 1)>,>
<1, receive[Fun (attack 0) []]>,>~~

~~###/R12/1~~

~~<1, send[signmsg (invkeymsg (PK 0)) (Fun tuple' [A 0, PK 0])]>,>
<*, <PK 0 in validset (A 0) (A 1)>,>
<*, $\forall X 0, X 1$ <PK 0 not in validset (Var (X 0)) (Var (X 1))>,>
<1, $\forall X 0, X 1$ <PK 0 not in revokedset (Var (X 0)) (Var (X 1))>,>
<*, <PK 0 not in beginauthset 0 (A 0) (A 1)>,>~~

~~###/R13/1~~

~~<*, <PK 0 in beginauthset 0 (A 0) (A 1)>,>
<*, <PK 0 in endauthset 0 (A 0) (A 1)>,>~~

~~###/R14/1~~

~~<*, receive[PK 0]>,>
<*, receive[invkeymsg (PK 0)]>,>~~

~~###/R15/1~~

~~<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)>,>~~

~~###/R16/1~~

~~<1, select<PK 0, ringset (A 0)>,>
<1, delete<PK 0, ringset (A 0)>,>~~

~~###/R17/1~~

~~<*, <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)>,>~~

~~###/R18/1~~

~~#####/R18/1~~

~~###/R19/1~~

```

⟨1, send⟨[PK 0]⟩⟩,

#A16/R710/1
⟨1, send⟨[Fun (attack 0) []]⟩⟩,

concat (steps from the second protocol / applicable steps / the / eg / 6)
#A16/R72/1
⟨2, send⟨[invkeymsg (PK 0)]⟩⟩,
⟨*, ⟨PK 0 in validset (A 0) (A 1)⟩⟩,
⟨2, receive⟨[Fun (attack 1) []]⟩⟩,

#A16/R72/2
⟨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))⟩⟩,

#A16/R73/2
⟨*, ⟨PK 0 in beginauthset 1 (A 0) (A 1)⟩⟩,
⟨*, ⟨PK 0 in endauthset 1 (A 0) (A 1)⟩⟩,

#A16/R74/2
⟨*, receive⟨[PK 0]⟩⟩,
⟨*, receive⟨[invkeymsg (PK 0)]⟩⟩,

#A16/R75/2
⟨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)))]⟩⟩,

#A16/R76/2
⟨*, ⟨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)⟩⟩,

#A16/R77/2
⟨2, receive⟨[pwmsg (A 0) (A 1)]⟩⟩,

#A16/R78/2
not applicable

#A16/R79/2
⟨2, insert⟨PK 0, pubkeysset (A 0)⟩⟩,

#A16/R710/2
⟨2, send⟨[Fun (attack 1) []]⟩⟩
]"

```

```

theorem "stm.tfrsst (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.wtsubst δ ∧ im.wftrms (subst_range δ) ∧ fv (t · δ) = {}}"

```

7 Examples

```
    and "Sec  $\equiv$  (f (set S)) - {m. im.intruder_synth {} m}"  
    shows "stm.par_compll_sst S'_{ks} Sec"  
    <proof>  
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.
- [5] A. V. Hess, S. A. Mödersheim, and A. D. Brucker. Stateful protocol composition in isabelle/hol. *ACM Transactions on Privacy and Security*, 2023. URL <https://www.brucker.ch/bibliography/abstract/hess.ea-stateful-protocol-composition-2023>.