Automated Stateful Protocol Verification

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

May 20, 2020

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

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

Abstract

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. In this AFP entry, we present a fully-automated approach for verifying stateful security protocols, i.e., protocols with mutable state that may span several sessions. The approach supports reachability goals like secrecy and authentication. We also include a simple user-friendly transaction-based protocol specification language that is embedded into Isabelle.

Keywords: Fully automated verification, stateful security protocols

Contents

1 Introduction

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. The latter provide overwhelmingly high assurance of the correctness, which automated methods often cannot: due to their complexity, bugs in such automated verification tools are likely and thus the risk of erroneously verifying a flawed protocol is non-negligible. There are a few works that try to combine advantages from both ends of the spectrum: a high degree of automation and assurance.

Inspired by [1], we present here a first step towards achieving this for a more challenging class of protocols, namely those that work with a mutable long-term state. To our knowledge this is the first approach that achieves fully automated verification of stateful protocols in an LCF-style theorem prover. The approach also includes a simple user-friendly transaction-based protocol specification language embedded into Isabelle, and can also leverage a number of existing results such as soundness of a typed model (see, e.g., [2–4]) and compositionality (see, e.g., [2, 5]). The Isabelle formalization extends the AFP entry on stateful protocol composition and typing [6].

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. Overall, the structure of this document follows the theory dependencies (see Figure 1.1): We start with the formal framework for verifying stateful security protocols (chapter 2). We continue with the setup for supporting the high-level protocol specifications language for security protocols (the Trac format) and the implementation of the fully automated proof tactics (chapter 3). Finally, we present examples (chapter 4).

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

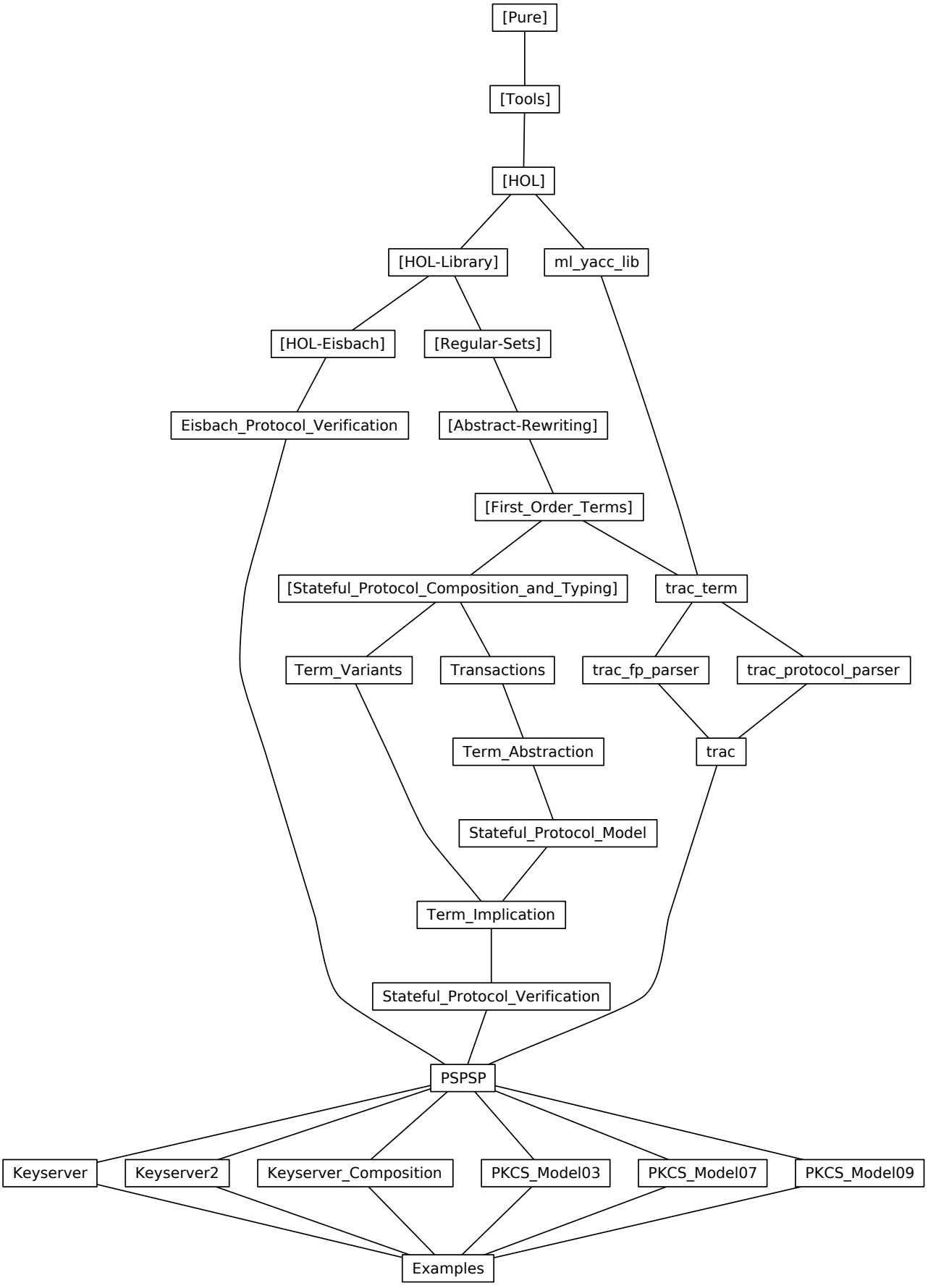

Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 Stateful Protocol Verification

2.1 Protocol Transactions (Transactions)

theory *Transactions*
  imports
    Stateful_Protocol_Composition_and_Typing.Typed_Model
    Stateful_Protocol_Composition_and_Typing.Labeled_Stateful_Strands
begin

2.1.1 Definitions

```
datatype 'b prot_atom =
  is_Atom: Atom 'b
| Value
| SetType
| AttackType
| Bottom
| OccursSecType

datatype ('a,'b,'c) prot_fun =
  Fu (the_Fu: 'a)
| Set (the_Set: 'c)
| Val (the_Val: "nat × bool")
| Abs (the_Abs: "'c set")
| Pair
| Attack nat
| PubConstAtom 'b nat
| PubConstSetType nat
| PubConstAttackType nat
| PubConstBottom nat
| PubConstOccursSecType nat
| OccursFact
| OccursSec

definition "is_Fun_Set t ≡ is_Fun t ∧ args t = [] ∧ is_Set (the_Fun t)"

abbreviation occurs where
  "occurs t ≡ Fun OccursFact [Fun OccursSec [], t]"

type_synonym ('a,'b,'c) prot_term_type = "(('a,'b,'c) prot_fun,'b prot_atom) term_type"

type_synonym ('a,'b,'c) prot_var = "('a,'b,'c) prot_term_type × nat"

type_synonym ('a,'b,'c) prot_term = "(('a,'b,'c) prot_fun,('a,'b,'c) prot_var) term"
type_synonym ('a,'b,'c) prot_terms = "('a,'b,'c) prot_term set"

type_synonym ('a,'b,'c) prot_subst = "(('a,'b,'c) prot_fun, ('a,'b,'c) prot_var) subst"

type_synonym ('a,'b,'c,'d) prot_strand_step =
  "(('a,'b,'c) prot_fun, ('a,'b,'c) prot_var, 'd) labeled_stateful_strand_step"
type_synonym ('a,'b,'c,'d) prot_strand = "('a,'b,'c,'d) prot_strand_step list"
type_synonym ('a,'b,'c,'d) prot_constr = "('a,'b,'c,'d) prot_strand_step list"

datatype ('a,'b,'c,'d) prot_transaction =
  Transaction
    (transaction_fresh:   "('a,'b,'c) prot_var list")
```

```
    (transaction_receive: "('a,'b,'c,'d) prot_strand")
    (transaction_selects: "('a,'b,'c,'d) prot_strand")
    (transaction_checks:  "('a,'b,'c,'d) prot_strand")
    (transaction_updates: "('a,'b,'c,'d) prot_strand")
    (transaction_send:    "('a,'b,'c,'d) prot_strand")

definition transaction_strand where
  "transaction_strand T ≡
    transaction_receive T@transaction_selects T@transaction_checks T@
    transaction_updates T@transaction_send T"

fun transaction_proj where
  "transaction_proj l (Transaction A B C D E F) = (
  let f = proj l
  in Transaction A (f B) (f C) (f D) (f E) (f F))"

fun transaction_star_proj where
  "transaction_star_proj (Transaction A B C D E F) = (
  let f = filter is_LabelS
  in Transaction A (f B) (f C) (f D) (f E) (f F))"

abbreviation fv_transaction where
  "fv_transaction T ≡ fv_lsst (transaction_strand T)"

abbreviation bvars_transaction where
  "bvars_transaction T ≡ bvars_lsst (transaction_strand T)"

abbreviation vars_transaction where
  "vars_transaction T ≡ vars_lsst (transaction_strand T)"

abbreviation trms_transaction where
  "trms_transaction T ≡ trms_lsst (transaction_strand T)"

abbreviation setops_transaction where
  "setops_transaction T ≡ setops_sst (unlabel (transaction_strand T))"

definition wellformed_transaction where
  "wellformed_transaction T ≡
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Assignment (unlabel (transaction_selects T)) ∧
    list_all is_Check (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T)) ∧
    set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) ∧
    set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {} ∧
    set (transaction_fresh T) ∩ fv_lsst (transaction_selects T) = {} ∧
    fv_transaction T ∩ bvars_transaction T = {} ∧
    fv_lsst (transaction_checks T) ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∧
    fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) - set (transaction_fresh T)
      ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∧
    (∀x ∈ set (unlabel (transaction_selects T)).
      is_Equality x ⟶ fv (the_rhs x) ⊆ fv_lsst (transaction_receive T))"

type_synonym ('a,'b,'c,'d) prot = "('a,'b,'c,'d) prot_transaction list"

abbreviation Var_Value_term ("⟨_⟩v") where
  "⟨n⟩v ≡ Var (Var Value, n)::('a,'b,'c) prot_term"

abbreviation Fun_Fu_term ("⟨_ _⟩t") where
  "⟨f T⟩t ≡ Fun (Fu f) T::('a,'b,'c) prot_term"

abbreviation Fun_Fu_const_term ("⟨_⟩c") where
  "⟨c⟩c ≡ Fun (Fu c) []::('a,'b,'c) prot_term"
```

```
abbreviation Fun_Set_const_term ("⟨_⟩s") where
  "⟨f⟩s ≡ Fun (Set f) []::('a,'b,'c) prot_term"

abbreviation Fun_Abs_const_term ("⟨_⟩a") where
  "⟨a⟩a ≡ Fun (Abs a) []::('a,'b,'c) prot_term"

abbreviation Fun_Attack_const_term ("attack⟨_⟩") where
  "attack⟨n⟩ ≡ Fun (Attack n) []::('a,'b,'c) prot_term"

abbreviation prot_transaction1 ("transaction1 _ _ new _ _ _") where
  "transaction1 (S1::('a,'b,'c,'d) prot_strand) S2 new (B::('a,'b,'c) prot_term list) S3 S4
  ≡ Transaction (map the_Var B) S1 [] S2 S3 S4"

abbreviation prot_transaction2 ("transaction2 _ _  _ _") where
  "transaction2 (S1::('a,'b,'c,'d) prot_strand) S2 S3 S4
  ≡ Transaction [] S1 [] S2 S3 S4"
```

2.1.2 Lemmata

```
lemma prot_atom_UNIV:
  "(UNIV::'b prot_atom set) = range Atom ∪ {Value, SetType, AttackType, Bottom, OccursSecType}"
proof -
  have "a ∈ range Atom ∨ a = Value ∨ a = SetType ∨ a = AttackType ∨ a = Bottom ∨ a = OccursSecType"
    for a::"'b prot_atom"
    by (cases a) auto
  thus ?thesis by auto
qed

instance prot_atom::(finite) finite
by intro_classes (simp add: prot_atom_UNIV)

instantiation prot_atom::(enum) enum
begin
definition "enum_prot_atom == map Atom enum_class.enum@[Value, SetType, AttackType, Bottom, OccursSecType]"
definition "enum_all_prot_atom P == list_all P (map Atom enum_class.enum@[Value, SetType, AttackType, Bottom,
OccursSecType])"
definition "enum_ex_prot_atom P == list_ex P (map Atom enum_class.enum@[Value, SetType, AttackType, Bottom,
OccursSecType])"

instance
proof intro_classes
  have *: "set (map Atom (enum_class.enum::'a list)) = range Atom"
          "distinct (enum_class.enum::'a list)"
    using UNIV_enum enum_distinct by auto

  show "(UNIV::'a prot_atom set) = set enum_class.enum"
    using *(1) by (simp add: prot_atom_UNIV enum_prot_atom_def)

  have "set (map Atom enum_class.enum) ∩ set [Value, SetType, AttackType, Bottom, OccursSecType] = {}"
by auto
  moreover have "inj_on Atom (set (enum_class.enum::'a list))" unfolding inj_on_def by auto
  hence "distinct (map Atom (enum_class.enum::'a list))" by (metis *(2) distinct_map)
  ultimately show "distinct (enum_class.enum::'a prot_atom list)" by (simp add: enum_prot_atom_def)

  have "Ball UNIV P ⟷ Ball (range Atom) P ∧ Ball {Value, SetType, AttackType, Bottom, OccursSecType}
P"
    for P::"'a prot_atom ⇒ bool"
    by (metis prot_atom_UNIV UNIV_I UnE)
  thus "enum_class.enum_all P = Ball (UNIV::'a prot_atom set) P" for P
    using *(1) Ball_set[of "map Atom enum_class.enum" P]
    by (auto simp add: enum_all_prot_atom_def)
```

```
  have "Bex UNIV P ⟷ Bex (range Atom) P ∨ Bex {Value, SetType, AttackType, Bottom, OccursSecType} P"
    for P::"'a prot_atom ⇒ bool"
    by (metis prot_atom_UNIV UNIV_I UnE)
  thus "enum_class.enum_ex P = Bex (UNIV::'a prot_atom set) P" for P
    using *(1) Bex_set[of "map Atom enum_class.enum" P]
    by (auto simp add: enum_ex_prot_atom_def)
qed
end

lemma wellformed_transaction_cases:
  assumes "wellformed_transaction T"
  shows
      "(l,x) ∈ set (transaction_receive T) ⟹ ∃t. x = receive⟨t⟩" (is "?A ⟹ ?A'")
      "(l,x) ∈ set (transaction_selects T) ⟹
            (∃t s. x = ⟨t := s⟩) ∨ (∃t s. x = select⟨t,s⟩)" (is "?B ⟹ ?B'")
      "(l,x) ∈ set (transaction_checks T) ⟹
             (∃t s. x = ⟨t == s⟩) ∨ (∃t s. x = ⟨t in s⟩) ∨ (∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)" (is "?C
⟹ ?C'")
      "(l,x) ∈ set (transaction_updates T) ⟹
             (∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" (is "?D ⟹ ?D'")
      "(l,x) ∈ set (transaction_send T) ⟹ ∃t. x = send⟨t⟩" (is "?E ⟹ ?E'")
proof -
  have a:
      "list_all is_Receive (unlabel (transaction_receive T))"
      "list_all is_Assignment (unlabel (transaction_selects T))"
      "list_all is_Check (unlabel (transaction_checks T))"
      "list_all is_Update (unlabel (transaction_updates T))"
      "list_all is_Send (unlabel (transaction_send T))"
    using assms unfolding wellformed_transaction_def by metis+

  note b = Ball_set unlabel_in
  note c = stateful_strand_step.collapse

  show "?A ⟹ ?A'" by (metis (mono_tags, lifting) a(1) b c(2))
  show "?B ⟹ ?B'" by (metis (mono_tags, lifting) a(2) b c(3,6))
  show "?C ⟹ ?C'" by (metis (mono_tags, lifting) a(3) b c(3,6,7))
  show "?D ⟹ ?D'" by (metis (mono_tags, lifting) a(4) b c(4,5))
  show "?E ⟹ ?E'" by (metis (mono_tags, lifting) a(5) b c(1))
qed

lemma wellformed_transaction_unlabel_cases:
  assumes "wellformed_transaction T"
  shows
      "x ∈ set (unlabel (transaction_receive T)) ⟹ ∃t. x = receive⟨t⟩" (is "?A ⟹ ?A'")
      "x ∈ set (unlabel (transaction_selects T)) ⟹
            (∃t s. x = ⟨t := s⟩) ∨ (∃t s. x = select⟨t,s⟩)" (is "?B ⟹ ?B'")
      "x ∈ set (unlabel (transaction_checks T)) ⟹
             (∃t s. x = ⟨t == s⟩) ∨ (∃t s. x = ⟨t in s⟩) ∨ (∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)"
        (is "?C ⟹ ?C'")
      "x ∈ set (unlabel (transaction_updates T)) ⟹
             (∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" (is "?D ⟹ ?D'")
      "x ∈ set (unlabel (transaction_send T)) ⟹ ∃t. x = send⟨t⟩" (is "?E ⟹ ?E'")
proof -
  have a:
      "list_all is_Receive (unlabel (transaction_receive T))"
      "list_all is_Assignment (unlabel (transaction_selects T))"
      "list_all is_Check (unlabel (transaction_checks T))"
      "list_all is_Update (unlabel (transaction_updates T))"
      "list_all is_Send (unlabel (transaction_send T))"
    using assms unfolding wellformed_transaction_def by metis+

  note b = Ball_set
  note c = stateful_strand_step.collapse
```

```
  show "?A ⟹ ?A'" by (metis (mono_tags, lifting) a(1) b c(2))
  show "?B ⟹ ?B'" by (metis (mono_tags, lifting) a(2) b c(3,6))
  show "?C ⟹ ?C'" by (metis (mono_tags, lifting) a(3) b c(3,6,7))
  show "?D ⟹ ?D'" by (metis (mono_tags, lifting) a(4) b c(4,5))
  show "?E ⟹ ?E'" by (metis (mono_tags, lifting) a(5) b c(1))
qed

lemma transaction_strand_subsets[simp]:
  "set (transaction_receive T) ⊆ set (transaction_strand T)"
  "set (transaction_selects T) ⊆ set (transaction_strand T)"
  "set (transaction_checks T) ⊆ set (transaction_strand T)"
  "set (transaction_updates T) ⊆ set (transaction_strand T)"
  "set (transaction_send T) ⊆ set (transaction_strand T)"
  "set (unlabel (transaction_receive T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_selects T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_checks T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_updates T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_send T)) ⊆ set (unlabel (transaction_strand T))"
unfolding transaction_strand_def unlabel_def by force+

lemma transaction_strand_subst_subsets[simp]:
  "set (transaction_receive T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_selects T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_checks T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_updates T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_send T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (unlabel (transaction_receive T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_selects T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_checks T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_updates T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_send T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
unfolding transaction_strand_def unlabel_def subst_apply_labeled_stateful_strand_def by force+

lemma transaction_dual_subst_unfold:
  "unlabel (duallsst (transaction_strand T ·lsst ϑ)) =
    unlabel (duallsst (transaction_receive T ·lsst ϑ))@
    unlabel (duallsst (transaction_selects T ·lsst ϑ))@
    unlabel (duallsst (transaction_checks T ·lsst ϑ))@
    unlabel (duallsst (transaction_updates T ·lsst ϑ))@
    unlabel (duallsst (transaction_send T ·lsst ϑ))"
by (simp add: transaction_strand_def unlabel_append duallsst_append subst_lsst_append)

lemma trms_transaction_unfold:
  "trms_transaction T =
      trmslsst (transaction_receive T) ∪ trmslsst (transaction_selects T) ∪
      trmslsst (transaction_checks T) ∪ trmslsst (transaction_updates T) ∪
      trmslsst (transaction_send T)"
by (metis trmssst_append unlabel_append append_assoc transaction_strand_def)

lemma trms_transaction_subst_unfold:
  "trmslsst (transaction_strand T ·lsst ϑ) =
      trmslsst (transaction_receive T ·lsst ϑ) ∪ trmslsst (transaction_selects T ·lsst ϑ) ∪
      trmslsst (transaction_checks T ·lsst ϑ) ∪ trmslsst (transaction_updates T ·lsst ϑ) ∪
      trmslsst (transaction_send T ·lsst ϑ)"
by (metis trmssst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma vars_transaction_unfold:
  "vars_transaction T =
      varslsst (transaction_receive T) ∪ varslsst (transaction_selects T) ∪
      varslsst (transaction_checks T) ∪ varslsst (transaction_updates T) ∪
      varslsst (transaction_send T)"
by (metis varssst_append unlabel_append append_assoc transaction_strand_def)
```

```
lemma vars_transaction_subst_unfold:
  "vars_lsst (transaction_strand T ·_lsst ϑ) =
      vars_lsst (transaction_receive T ·_lsst ϑ) ∪ vars_lsst (transaction_selects T ·_lsst ϑ) ∪
      vars_lsst (transaction_checks T ·_lsst ϑ) ∪ vars_lsst (transaction_updates T ·_lsst ϑ) ∪
      vars_lsst (transaction_send T ·_lsst ϑ)"
by (metis vars_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma fv_transaction_unfold:
  "fv_transaction T =
      fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∪
      fv_lsst (transaction_checks T) ∪ fv_lsst (transaction_updates T) ∪
      fv_lsst (transaction_send T)"
by (metis fv_sst_append unlabel_append append_assoc transaction_strand_def)

lemma fv_transaction_subst_unfold:
  "fv_lsst (transaction_strand T ·_lsst ϑ) =
      fv_lsst (transaction_receive T ·_lsst ϑ) ∪ fv_lsst (transaction_selects T ·_lsst ϑ) ∪
      fv_lsst (transaction_checks T ·_lsst ϑ) ∪ fv_lsst (transaction_updates T ·_lsst ϑ) ∪
      fv_lsst (transaction_send T ·_lsst ϑ)"
by (metis fv_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma fv_wellformed_transaction_unfold:
  assumes "wellformed_transaction T"
  shows "fv_transaction T =
    fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∪ set (transaction_fresh T)"
proof -
  let ?A = "set (transaction_fresh T)"
  let ?B = "fv_lsst (transaction_updates T)"
  let ?C = "fv_lsst (transaction_send T)"
  let ?D = "fv_lsst (transaction_receive T)"
  let ?E = "fv_lsst (transaction_selects T)"
  let ?F = "fv_lsst (transaction_checks T)"

  have "?A ⊆ ?B ∪ ?C" "?A ∩ ?D = {}" "?A ∩ ?E = {}" "?F ⊆ ?D ∪ ?E" "?B ∪ ?C - ?A ⊆ ?D ∪ ?E"
    using assms unfolding wellformed_transaction_def by fast+
  thus ?thesis using fv_transaction_unfold by blast
qed

lemma bvars_transaction_unfold:
  "bvars_transaction T =
      bvars_lsst (transaction_receive T) ∪ bvars_lsst (transaction_selects T) ∪
      bvars_lsst (transaction_checks T) ∪ bvars_lsst (transaction_updates T) ∪
      bvars_lsst (transaction_send T)"
by (metis bvars_sst_append unlabel_append append_assoc transaction_strand_def)

lemma bvars_transaction_subst_unfold:
  "bvars_lsst (transaction_strand T ·_lsst ϑ) =
      bvars_lsst (transaction_receive T ·_lsst ϑ) ∪ bvars_lsst (transaction_selects T ·_lsst ϑ) ∪
      bvars_lsst (transaction_checks T ·_lsst ϑ) ∪ bvars_lsst (transaction_updates T ·_lsst ϑ) ∪
      bvars_lsst (transaction_send T ·_lsst ϑ)"
by (metis bvars_sst_append unlabel_append append_assoc transaction_strand_def subst_lsst_append)

lemma bvars_wellformed_transaction_unfold:
  assumes "wellformed_transaction T"
  shows "bvars_transaction T = bvars_lsst (transaction_checks T)" (is ?A)
    and "bvars_lsst (transaction_receive T) = {}" (is ?B)
    and "bvars_lsst (transaction_selects T) = {}" (is ?C)
    and "bvars_lsst (transaction_updates T) = {}" (is ?D)
    and "bvars_lsst (transaction_send T) = {}" (is ?E)
proof -
  have 0: "list_all is_Receive (unlabel (transaction_receive T))"
          "list_all is_Assignment (unlabel (transaction_selects T))"
```

```
        "list_all is_Update (unlabel (transaction_updates T))"
        "list_all is_Send (unlabel (transaction_send T))"
    using assms unfolding wellformed_transaction_def by metis+

  have "filter is_NegChecks (unlabel (transaction_receive T)) = []"
       "filter is_NegChecks (unlabel (transaction_selects T)) = []"
       "filter is_NegChecks (unlabel (transaction_updates T)) = []"
       "filter is_NegChecks (unlabel (transaction_send T)) = []"
    using list_all_filter_nil[OF 0(1), of is_NegChecks]
          list_all_filter_nil[OF 0(2), of is_NegChecks]
          list_all_filter_nil[OF 0(3), of is_NegChecks]
          list_all_filter_nil[OF 0(4), of is_NegChecks]
          stateful_strand_step.distinct_disc(11,21,29,35,39,41)
    by blast+
  thus ?A ?B ?C ?D ?E
    using bvars_transaction_unfold[of T]
          bvars_sst_NegChecks[of "unlabel (transaction_receive T)"]
          bvars_sst_NegChecks[of "unlabel (transaction_selects T)"]
          bvars_sst_NegChecks[of "unlabel (transaction_updates T)"]
          bvars_sst_NegChecks[of "unlabel (transaction_send T)"]
    by (metis bvars_sst_def UnionE emptyE list.set(1) list.simps(8) subsetI subset_Un_eq sup_commute)+
qed

lemma transaction_strand_memberD[dest]:
  assumes "x ∈ set (transaction_strand T)"
  shows "x ∈ set (transaction_receive T) ∨ x ∈ set (transaction_selects T) ∨
         x ∈ set (transaction_checks T) ∨ x ∈ set (transaction_updates T) ∨
         x ∈ set (transaction_send T)"
using assms by (simp add: transaction_strand_def)

lemma transaction_strand_unlabel_memberD[dest]:
  assumes "x ∈ set (unlabel (transaction_strand T))"
  shows "x ∈ set (unlabel (transaction_receive T)) ∨ x ∈ set (unlabel (transaction_selects T)) ∨
         x ∈ set (unlabel (transaction_checks T)) ∨ x ∈ set (unlabel (transaction_updates T)) ∨
         x ∈ set (unlabel (transaction_send T))"
using assms by (simp add: unlabel_def transaction_strand_def)

lemma wellformed_transaction_strand_memberD[dest]:
  assumes "wellformed_transaction T" and "(l,x) ∈ set (transaction_strand T)"
  shows
    "x = receive⟨t⟩ ⟹ (l,x) ∈ set (transaction_receive T)" (is "?A ⟹ ?A'")
    "x = select⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_selects T)" (is "?B ⟹ ?B'")
    "x = ⟨t == s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?C ⟹ ?C'")
    "x = ⟨t in s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?D ⟹ ?D'")
    "x = ∀X⟨∨≠: F ∨∉: G⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (is "?E ⟹ ?E'")
    "x = insert⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_updates T)" (is "?F ⟹ ?F'")
    "x = delete⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_updates T)" (is "?G ⟹ ?G'")
    "x = send⟨t⟩ ⟹ (l,x) ∈ set (transaction_send T)" (is "?H ⟹ ?H'")
proof -
  have "(l,x) ∈ set (transaction_receive T) ∨ (l,x) ∈ set (transaction_selects T) ∨
        (l,x) ∈ set (transaction_checks T) ∨ (l,x) ∈ set (transaction_updates T) ∨
        (l,x) ∈ set (transaction_send T)"
    using assms(2) by auto
  thus "?A ⟹ ?A'" "?B ⟹ ?B'" "?C ⟹ ?C'" "?D ⟹ ?D'"
       "?E ⟹ ?E'" "?F ⟹ ?F'" "?G ⟹ ?G'" "?H ⟹ ?H'"
    using wellformed_transaction_cases[OF assms(1)] by fast+
qed

lemma wellformed_transaction_strand_unlabel_memberD[dest]:
  assumes "wellformed_transaction T" and "x ∈ set (unlabel (transaction_strand T))"
  shows
    "x = receive⟨t⟩ ⟹ x ∈ set (unlabel (transaction_receive T))" (is "?A ⟹ ?A'")
    "x = select⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_selects T))" (is "?B ⟹ ?B'")
```

```
    "x = ⟨t == s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?C ⟹ ?C'")
    "x = ⟨t in s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?D ⟹ ?D'")
    "x = ∀X⟨∨≠: F ∨∉: G⟩  ⟹ x ∈ set (unlabel (transaction_checks T))" (is "?E ⟹ ?E'")
    "x = insert⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_updates T))" (is "?F ⟹ ?F'")
    "x = delete⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_updates T))" (is "?G ⟹ ?G'")
    "x = send⟨t⟩ ⟹ x ∈ set (unlabel (transaction_send T))" (is "?H ⟹ ?H'")
proof -
  have "x ∈ set (unlabel (transaction_receive T)) ∨ x ∈ set (unlabel (transaction_selects T)) ∨
        x ∈ set (unlabel (transaction_checks T)) ∨ x ∈ set (unlabel (transaction_updates T)) ∨
        x ∈ set (unlabel (transaction_send T))"
    using assms(2) by auto
  thus "?A ⟹ ?A'" "?B ⟹ ?B'" "?C ⟹ ?C'" "?D ⟹ ?D'"
       "?E ⟹ ?E'" "?F ⟹ ?F'" "?G ⟹ ?G'" "?H ⟹ ?H'"
    using wellformed_transaction_unlabel_cases[OF assms(1)] by fast+
qed

lemma wellformed_transaction_send_receive_trm_cases:
  assumes T: "wellformed_transaction T"
  shows "t ∈ trms_lsst (transaction_receive T) ⟹ receive⟨t⟩ ∈ set (unlabel (transaction_receive T))"
    and "t ∈ trms_lsst (transaction_send T) ⟹ send⟨t⟩ ∈ set (unlabel (transaction_send T))"
using wellformed_transaction_unlabel_cases(1,5)[OF T]
      trms_sst_in[of t "unlabel (transaction_receive T)"]
      trms_sst_in[of t "unlabel (transaction_send T)"]
by fastforce+

lemma wellformed_transaction_send_receive_subst_trm_cases:
  assumes T: "wellformed_transaction T"
  shows "t ∈ trms_lsst (transaction_receive T) ·_set ϑ ⟹ receive⟨t⟩ ∈ set (unlabel (transaction_receive
T ·_lsst ϑ))"
    and "t ∈ trms_lsst (transaction_send T) ·_set ϑ ⟹ send⟨t⟩ ∈ set (unlabel (transaction_send T ·_lsst ϑ))"
proof -
  assume "t ∈ trms_lsst (transaction_receive T) ·_set ϑ"
  then obtain s where s: "s ∈ trms_lsst (transaction_receive T)" "t = s · ϑ"
    by blast
  hence "receive⟨s⟩ ∈ set (unlabel (transaction_receive T))"
    using wellformed_transaction_send_receive_trm_cases(1)[OF T] by simp
  thus "receive⟨t⟩ ∈ set (unlabel (transaction_receive T ·_lsst ϑ))"
    by (metis s(2) unlabel_subst[of _ ϑ] stateful_strand_step_subst_inI(2))
next
  assume "t ∈ trms_lsst (transaction_send T) ·_set ϑ"
  then obtain s where s: "s ∈ trms_lsst (transaction_send T)" "t = s · ϑ"
    by blast
  hence "send⟨s⟩ ∈ set (unlabel (transaction_send T))"
    using wellformed_transaction_send_receive_trm_cases(2)[OF T] by simp
  thus "send⟨t⟩ ∈ set (unlabel (transaction_send T ·_lsst ϑ))"
    by (metis s(2) unlabel_subst[of _ ϑ] stateful_strand_step_subst_inI(1))
qed

lemma wellformed_transaction_send_receive_fv_subset:
  assumes T: "wellformed_transaction T"
  shows "t ∈ trms_lsst (transaction_receive T) ⟹ fv t ⊆ fv_transaction T" (is "?A ⟹ ?A'")
    and "t ∈ trms_lsst (transaction_send T) ⟹ fv t ⊆ fv_transaction T" (is "?B ⟹ ?B'")
proof -
  have "t ∈ trms_lsst (transaction_receive T) ⟹ receive⟨t⟩ ∈ set (unlabel (transaction_strand T))"
       "t ∈ trms_lsst (transaction_send T) ⟹ send⟨t⟩ ∈ set (unlabel (transaction_strand T))"
    using wellformed_transaction_send_receive_trm_cases[OF T, of t]
    unfolding transaction_strand_def by force+
  thus "?A ⟹ ?A'" "?B ⟹ ?B'" by (induct "transaction_strand T") auto
qed

lemma dual_wellformed_transaction_ident_cases[dest]:
  "list_all is_Assignment (unlabel S) ⟹ dual_lsst S = S"
  "list_all is_Check (unlabel S) ⟹ dual_lsst S = S"
```

```
  "list_all is_Update (unlabel S) ⟹ dual_lsst S = S"
proof (induction S)
  case (Cons s S)
  obtain l x where s: "s = (l,x)" by moura
  { case 1 thus ?case using Cons s unfolding unlabel_def dual_lsst_def by (cases x) auto }
  { case 2 thus ?case using Cons s unfolding unlabel_def dual_lsst_def by (cases x) auto }
  { case 3 thus ?case using Cons s unfolding unlabel_def dual_lsst_def by (cases x) auto }
qed simp_all

lemma wellformed_transaction_wf_sst:
  fixes T::"('a, 'b, 'c, 'd) prot_transaction"
  assumes T: "wellformed_transaction T"
  shows "wf'_sst (set (transaction_fresh T)) (unlabel (dual_lsst (transaction_strand T)))" (is ?A)
    and "fv_transaction T ∩ bvars_transaction T = {}" (is ?B)
    and "set (transaction_fresh T) ∩ bvars_transaction T = {}" (is ?C)
proof -
  define T1 where "T1 ≡ unlabel (dual_lsst (transaction_receive T))"
  define T2 where "T2 ≡ unlabel (dual_lsst (transaction_selects T))"
  define T3 where "T3 ≡ unlabel (dual_lsst (transaction_checks T))"
  define T4 where "T4 ≡ unlabel (dual_lsst (transaction_updates T))"
  define T5 where "T5 ≡ unlabel (dual_lsst (transaction_send T))"

  define X where "X ≡ set (transaction_fresh T)"
  define Y where "Y ≡ X ∪ wfvarsoccs_sst T1"
  define Z where "Z ≡ Y ∪ wfvarsoccs_sst T2"

  define f where "f ≡ λS::('a,'b,'c) prot_fun, ('a,'b,'c) prot_var) stateful_strand.
        ⋃((λx. case x of
          Receive t ⇒ fv t
        | Equality Assign _ t' ⇒ fv t'
        | Insert t t' ⇒ fv t ∪ fv t'
        | _ ⇒ {}) ` set S)"

  note defs1 = T1_def T2_def T3_def T4_def T5_def
  note defs2 = X_def Y_def Z_def
  note defs3 = f_def

  have 0: "wf'_sst V (S @ S')"
    when "wf'_sst V S" "f S' ⊆ wfvarsoccs_sst S ∪ V" for V S S'
    by (metis that wf_sst_append_suffix' f_def)

  have 1: "unlabel (dual_lsst (transaction_strand T)) = T1@T2@T3@T4@T5"
    using dual_lsst_append unlabel_append unfolding transaction_strand_def defs1 by simp

  have 2:
      "∀x ∈ set T1. is_Send x" "∀x ∈ set T2. is_Assignment x" "∀x ∈ set T3. is_Check x"
      "∀x ∈ set T4. is_Update x" "∀x ∈ set T5. is_Receive x"
      "fv_sst T3 ⊆ fv_sst T1 ∪ fv_sst T2" "fv_sst T4 ∪ fv_sst T5 ⊆ X ∪ fv_sst T1 ∪ fv_sst T2"
      "X ∩ fv_sst T1 = {}" "X ∩ fv_sst T2 = {}"
      "∀x ∈ set T2. is_Equality x ⟶ fv (the_rhs x) ⊆ fv_sst T1"
    using T unfolding defs1 defs2 wellformed_transaction_def
    by (auto simp add: Ball_set dual_lsst_list_all fv_sst_unlabel_dual_lsst_eq simp del: fv_sst_def)

  have 3: "wf'_sst X T1" using 2(1)
  proof (induction T1 arbitrary: X)
    case (Cons s T)
    obtain t where "s = send⟨t⟩" using Cons.prems by (cases s) moura+
    thus ?case using Cons by auto
  qed simp

  have 4: "f T1 = {}" "fv_sst T1 = wfvarsoccs_sst T1" using 2(1)
  proof (induction T1)
    case (Cons s T)
```

```
  { case 1 thus ?case using Cons unfolding defs3 by (cases s) auto }
  { case 2 thus ?case using Cons unfolding defs3 wfvarsoccs_sst_def fv_sst_def by (cases s) auto }
qed (simp_all add: defs3 wfvarsoccs_sst_def fv_sst_def)

have 5: "f T2 ⊆ wfvarsoccs_sst T1" "fv_sst T2 = f T2 ∪ wfvarsoccs_sst T2" using 2(2,10)
proof (induction T2)
  case (Cons s T)
  { case 1 thus ?case using Cons
    proof (cases s)
      case (Equality ac t t') thus ?thesis using 1 Cons 4(2) unfolding defs3 by (cases ac) auto
    qed (simp_all add: defs3)
  }
  { case 2 thus ?case using Cons
    proof (cases s)
      case (Equality ac t t')
      hence "ac = Assign" "fv_sstp s = fv t' ∪ wfvarsoccs_sstp s" "f (s#T) = fv t' ∪ f T"
        using 2 unfolding defs3 by auto
      moreover have "fv_sst T = f T ∪ wfvarsoccs_sst T" using Cons.IH(2) 2 by auto
      ultimately show ?thesis unfolding wfvarsoccs_sst_def fv_sst_def by auto
    next
      case (InSet ac t t')
      hence "ac = Assign" "fv_sstp s = wfvarsoccs_sstp s" "f (s#T) = f T"
        using 2 unfolding defs3 by auto
      moreover have "fv_sst T = f T ∪ wfvarsoccs_sst T" using Cons.IH(2) 2 by auto
      ultimately show ?thesis unfolding wfvarsoccs_sst_def fv_sst_def by auto
    qed (simp_all add: defs3)
  }
qed (simp_all add: defs3 wfvarsoccs_sst_def fv_sst_def)

have "f T ⊆ fv_sst T" for T
proof
  fix x show "x ∈ f T ⟹ x ∈ fv_sst T"
  proof (induction T)
    case (Cons s T) thus ?case
    proof (cases "x ∈ f T")
      case False thus ?thesis
        using Cons.prems unfolding defs3 fv_sst_def
        by (auto split: stateful_strand_step.splits poscheckvariant.splits)
    qed auto
  qed (simp add: defs3 fv_sst_def)
qed
hence 6:
    "f T3 ⊆ X ∪ wfvarsoccs_sst T1 ∪ wfvarsoccs_sst T2"
    "f T4 ⊆ X ∪ wfvarsoccs_sst T1 ∪ wfvarsoccs_sst T2"
    "f T5 ⊆ X ∪ wfvarsoccs_sst T1 ∪ wfvarsoccs_sst T2"
  using 2(6,7) 4 5 by blast+

have 7:
    "wfvarsoccs_sst T3 = {}"
    "wfvarsoccs_sst T4 = {}"
    "wfvarsoccs_sst T5 = {}"
  using 2(3,4,5) unfolding wfvarsoccs_sst_def
  by (auto split: stateful_strand_step.splits)

have 8:
    "f T2 ⊆ wfvarsoccs_sst T1 ∪ X"
    "f T3 ⊆ wfvarsoccs_sst (T1@T2) ∪ X"
    "f T4 ⊆ wfvarsoccs_sst ((T1@T2)@T3) ∪ X"
    "f T5 ⊆ wfvarsoccs_sst (((T1@T2)@T3)@T4) ∪ X"
  using 4(1) 5(1) 6 7 wfvarsoccs_sst_append[of T1 T2]
        wfvarsoccs_sst_append[of "T1@T2" T3]
        wfvarsoccs_sst_append[of "(T1@T2)@T3" T4]
  by blast+
```

```
  have "wf'_sst X (T1@T2@T3@T4@T5)"
    using 0[OF 0[OF 0[OF 0[OF 3 8(1)] 8(2)] 8(3)] 8(4)]
    unfolding Y_def Z_def by simp
  thus ?A using 1 unfolding defs1 defs2 by simp

  have "set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
       "fv_transaction T ∩ bvars_transaction T = {}"
    using T unfolding wellformed_transaction_def by fast+
  thus ?B ?C using fv_transaction_unfold[of T] bvars_transaction_unfold[of T] by blast+
qed

lemma dual_wellformed_transaction_ident_cases'[dest]:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_selects T) = transaction_selects T"
        "dual_lsst (transaction_checks T) = transaction_checks T"
        "dual_lsst (transaction_updates T) = transaction_updates T"
using assms unfolding wellformed_transaction_def by auto

lemma dual_transaction_strand:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_strand T) =
         dual_lsst (transaction_receive T)@transaction_selects T@transaction_checks T@
         transaction_updates T@dual_lsst (transaction_send T)"
using dual_wellformed_transaction_ident_cases'[OF assms] dual_lsst_append
unfolding transaction_strand_def by metis

lemma dual_unlabel_transaction_strand:
  assumes "wellformed_transaction T"
  shows "unlabel (dual_lsst (transaction_strand T)) =
         (unlabel (dual_lsst (transaction_receive T)))@(unlabel (transaction_selects T))@
         (unlabel (transaction_checks T))@(unlabel (transaction_updates T))@
         (unlabel (dual_lsst (transaction_send T)))"
using dual_transaction_strand[OF assms] by (simp add: unlabel_def)

lemma dual_transaction_strand_subst:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_strand T ·_lsst δ) =
         (dual_lsst (transaction_receive T)@transaction_selects T@transaction_checks T@
          transaction_updates T@dual_lsst (transaction_send T)) ·_lsst δ"
proof -
  have "dual_lsst (transaction_strand T ·_lsst δ) = dual_lsst (transaction_strand T) ·_lsst δ"
    using dual_lsst_subst by metis
  thus ?thesis using dual_transaction_strand[OF assms] by argo
qed

lemma dual_transaction_ik_is_transaction_send:
  assumes "wellformed_transaction T"
  shows "ik_sst (unlabel (dual_lsst (transaction_strand T))) = trms_sst (unlabel (transaction_send T))"
    (is "?A = ?B")
proof -
  { fix t assume "t ∈ ?A"
    hence "receive⟨t⟩ ∈ set (unlabel (dual_lsst (transaction_strand T)))" by (simp add: ik_sst_def)
    hence "send⟨t⟩ ∈ set (unlabel (transaction_strand T))"
      using dual_lsst_unlabel_steps_iff(1) by metis
    hence "t ∈ ?B" using wellformed_transaction_strand_unlabel_memberD(8)[OF assms] by force
  } moreover {
    fix t assume "t ∈ ?B"
    hence "send⟨t⟩ ∈ set (unlabel (transaction_send T))"
      using wellformed_transaction_unlabel_cases(5)[OF assms] by fastforce
    hence "receive⟨t⟩ ∈ set (unlabel (dual_lsst (transaction_send T)))"
      using dual_lsst_unlabel_steps_iff(1) by metis
    hence "receive⟨t⟩ ∈ set (unlabel (dual_lsst (transaction_strand T)))"
```

```
    using dual_unlabel_transaction_strand[OF assms] by simp
  hence "t ∈ ?A" by (simp add: ik_sst_def)
 } ultimately show "?A = ?B" by auto
qed

lemma dual_transaction_ik_is_transaction_send':
 fixes δ::"('a,'b,'c) prot_subst"
 assumes "wellformed_transaction T"
 shows "ik_sst (unlabel (dual_lsst (transaction_strand T ·_lsst δ))) =
        trms_sst (unlabel (transaction_send T)) ·_set δ" (is "?A = ?B")
using dual_transaction_ik_is_transaction_send[OF assms]
      subst_lsst_unlabel[of "dual_lsst (transaction_strand T)" δ]
      ik_sst_subst[of "unlabel (dual_lsst (transaction_strand T))" δ]
      dual_lsst_subst[of "transaction_strand T" δ]
by auto

lemma db_sst_transaction_prefix_eq:
 assumes T: "wellformed_transaction T"
   and S: "prefix S (transaction_receive T@transaction_selects T@transaction_checks T)"
 shows "db_lsst A = db_lsst (A@dual_lsst (S ·_lsst δ))"
proof -
 let ?T1 = "transaction_receive T"
 let ?T2 = "transaction_selects T"
 let ?T3 = "transaction_checks T"

 have *: "prefix (unlabel S) (unlabel (?T1@?T2@?T3))" using S prefix_proj(1) by blast

 have "list_all is_Receive (unlabel ?T1)"
      "list_all is_Assignment (unlabel ?T2)"
      "list_all is_Check (unlabel ?T3)"
   using T by (simp_all add: wellformed_transaction_def)
 hence "∀b ∈ set (unlabel ?T1). ¬is_Insert b ∧ ¬is_Delete b"
       "∀b ∈ set (unlabel ?T2). ¬is_Insert b ∧ ¬is_Delete b"
       "∀b ∈ set (unlabel ?T3). ¬is_Insert b ∧ ¬is_Delete b"
   by (metis (mono_tags, lifting) Ball_set stateful_strand_step.distinct_disc(16,18),
       metis (mono_tags, lifting) Ball_set stateful_strand_step.distinct_disc(24,26,33,37),
       metis (mono_tags, lifting) Ball_set stateful_strand_step.distinct_disc(24,26,33,35,37,39))
 hence "∀b ∈ set (unlabel (?T1@?T2@?T3)). ¬is_Insert b ∧ ¬is_Delete b"
   by (auto simp add: unlabel_def)
 hence "∀b ∈ set (unlabel S). ¬is_Insert b ∧ ¬is_Delete b"
   using * unfolding prefix_def by fastforce
 hence "∀b ∈ set (unlabel (dual_lsst S) ·_sst δ). ¬is_Insert b ∧ ¬is_Delete b"
 proof (induction S)
   case (Cons a S)
   then obtain l b where "a = (l,b)" by (metis surj_pair)
   thus ?case
     using Cons unfolding dual_lsst_def unlabel_def subst_apply_stateful_strand_def
     by (cases b) auto
 qed simp
 hence **: "∀b ∈ set (unlabel (dual_lsst (S ·_lsst δ))). ¬is_Insert b ∧ ¬is_Delete b"
   by (metis dual_lsst_subst_unlabel)

 show ?thesis
   using db_sst_no_upd_append[OF **] unlabel_append
   unfolding db_sst_def by metis
qed

lemma db_lsst_dual_lsst_set_ex:
  assumes "d ∈ set (db'_lsst (dual_lsst A ·_lsst ϑ) I D)"
   "∀t u. insert⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃s. u = Fun (Set s) [])"
   "∀t u. delete⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃s. u = Fun (Set s) [])"
   "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
 shows "∃s. snd d = Fun (Set s) []"
```

```
  using assms
proof (induction A arbitrary: D)
  case (Cons a A)
  obtain l b where a: "a = (l,b)" by (metis surj_pair)

  have 1: "unlabel (dual_lsst (a#A) ·_lsst ϑ) = receive⟨t · ϑ⟩#unlabel (dual_lsst A ·_lsst ϑ)"
    when "b = send⟨t⟩" for t
    by (simp add: a that subst_lsst_unlabel_cons)

  have 2: "unlabel (dual_lsst (a#A) ·_lsst ϑ) = send⟨t · ϑ⟩#unlabel (dual_lsst A ·_lsst ϑ)"
    when "b = receive⟨t⟩" for t
    by (simp add: a that subst_lsst_unlabel_cons)

  have 3: "unlabel (dual_lsst (a#A) ·_lsst ϑ) = (b ·_sstp ϑ)#unlabel (dual_lsst A ·_lsst ϑ)"
    when "∄t. b = send⟨t⟩ ∨ b = receive⟨t⟩"
    using a that dual_lsst_Cons subst_lsst_unlabel_cons[of l b]
    by (cases b) auto

  show ?case using 1 2 3 a Cons by (cases b) fastforce+
qed simp

lemma is_Fun_SetE[elim]:
  assumes t: "is_Fun_Set t"
  obtains s where "t = Fun (Set s) []"
proof (cases t)
  case (Fun f T)
  then obtain s where "f = Set s" using t unfolding is_Fun_Set_def by (cases f) moura+
  moreover have "T = []" using Fun t unfolding is_Fun_Set_def by (cases T) auto
  ultimately show ?thesis using Fun that by fast
qed (use t is_Fun_Set_def in fast)

lemma Fun_Set_InSet_iff:
  "(u = ⟨a: Var x ∈ Fun (Set s) []⟩) ⟷
   (is_InSet u ∧ is_Var (the_elem_term u) ∧ is_Fun_Set (the_set_term u) ∧
    the_Set (the_Fun (the_set_term u)) = s ∧ the_Var (the_elem_term u) = x ∧ the_check u = a)"
  (is "?A ⟷ ?B")
proof
  show "?A ⟹ ?B" unfolding is_Fun_Set_def by auto

  assume B: ?B
  thus ?A
  proof (cases u)
    case (InSet b t t')
    hence "b = a" "t = Var x" "t' = Fun (Set s) []"
      using B by (simp, fastforce, fastforce)
    thus ?thesis using InSet by fast
  qed auto
qed

lemma Fun_Set_NotInSet_iff:
  "(u = ⟨Var x not in Fun (Set s) []⟩) ⟷
   (is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1 ∧
    is_Var (fst (hd (the_ins u))) ∧ is_Fun_Set (snd (hd (the_ins u)))) ∧
    the_Set (the_Fun (snd (hd (the_ins u)))) = s ∧ the_Var (fst (hd (the_ins u))) = x"
  (is "?A ⟷ ?B")
proof
  show "?A ⟹ ?B" unfolding is_Fun_Set_def by auto

  assume B: ?B
  show ?A
  proof (cases u)
    case (NegChecks X F F')
    hence "X = []" "F = []"
```

```
    using B by auto
  moreover have "fst (hd (the_ins u)) = Var x" "snd (hd (the_ins u)) = Fun (Set s) []"
    using B is_Fun_SetE[of "snd (hd (the_ins u))"]
    by (force, fastforce)
  hence "F' = [(Var x, Fun (Set s) [])]"
    using NegChecks B by (cases "the_ins u") auto
  ultimately show ?thesis using NegChecks by fast
 qed (use B in auto)
qed

lemma is_Fun_Set_exi: "is_Fun_Set x ⟷ (∃s. x = Fun (Set s) [])"
by (metis prot_fun.collapse(2) term.collapse(2) prot_fun.disc(15) term.disc(2)
          term.sel(2,4) is_Fun_Set_def un_Fun1_def)

lemma is_Fun_Set_subst:
 assumes "is_Fun_Set S'"
 shows "is_Fun_Set (S' · σ)"
using assms by (fastforce simp add: is_Fun_Set_def)

lemma is_Update_in_transaction_updates:
 assumes tu: "is_Update t"
 assumes t: "t ∈ set (unlabel (transaction_strand TT))"
 assumes vt: "wellformed_transaction TT"
 shows "t ∈ set (unlabel (transaction_updates TT))"
using t tu vt unfolding transaction_strand_def wellformed_transaction_def list_all_iff
by (auto simp add: unlabel_append)

lemma transaction_fresh_vars_subset:
 assumes "wellformed_transaction T"
 shows "set (transaction_fresh T) ⊆ fv_transaction T"
using assms fv_transaction_unfold[of T]
unfolding wellformed_transaction_def
by auto

lemma transaction_fresh_vars_notin:
 assumes T: "wellformed_transaction T"
   and x: "x ∈ set (transaction_fresh T)"
 shows "x ∉ fv_lsst (transaction_receive T)" (is ?A)
   and "x ∉ fv_lsst (transaction_selects T)" (is ?B)
   and "x ∉ fv_lsst (transaction_checks T)" (is ?C)
   and "x ∉ vars_lsst (transaction_receive T)" (is ?D)
   and "x ∉ vars_lsst (transaction_selects T)" (is ?E)
   and "x ∉ vars_lsst (transaction_checks T)" (is ?F)
   and "x ∉ bvars_lsst (transaction_receive T)" (is ?G)
   and "x ∉ bvars_lsst (transaction_selects T)" (is ?H)
   and "x ∉ bvars_lsst (transaction_checks T)" (is ?I)
proof -
 have 0:
    "set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
    "set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {}"
    "set (transaction_fresh T) ∩ fv_lsst (transaction_selects T) = {}"
    "fv_transaction T ∩ bvars_transaction T = {}"
    "fv_lsst (transaction_checks T) ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T)"
   using T unfolding wellformed_transaction_def
   by fast+

 have 1: "set (transaction_fresh T) ∩ bvars_lsst (transaction_checks T) = {}"
   using 0(1,4) fv_transaction_unfold[of T] bvars_transaction_unfold[of T] by blast

 have 2:
    "vars_lsst (transaction_receive T) = fv_lsst (transaction_receive T)"
    "vars_lsst (transaction_selects T) = fv_lsst (transaction_selects T)"
    "bvars_lsst (transaction_receive T) = {}"
```

```
    "bvars_lsst (transaction_selects T) = {}"
  using bvars_wellformed_transaction_unfold[OF T] bvars_transaction_unfold[of T]
        vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_receive T)"]
        vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_selects T)"]
  by blast+

 show ?A ?B ?C ?D ?E ?G ?H ?I using 0 1 2 x by fast+

 show ?F using 0(2,3,5) 1 x vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_checks T)"] by fast
qed


lemma transaction_proj_member:
 assumes "T ∈ set P"
 shows "transaction_proj n T ∈ set (map (transaction_proj n) P)"
using assms by simp

lemma transaction_strand_proj:
 "transaction_strand (transaction_proj n T) = proj n (transaction_strand T)"
proof -
 obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
 thus ?thesis
  using transaction_proj.simps[of n A B C D E F]
  unfolding transaction_strand_def proj_def Let_def by auto
qed

lemma transaction_proj_fresh_eq:
 "transaction_fresh (transaction_proj n T) = transaction_fresh T"
proof -
 obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
 thus ?thesis
  using transaction_proj.simps[of n A B C D E F]
  unfolding transaction_fresh_def proj_def Let_def by auto
qed

lemma transaction_proj_trms_subset:
 "trms_transaction (transaction_proj n T) ⊆ trms_transaction T"
proof -
 obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
 thus ?thesis
  using transaction_proj.simps[of n A B C D E F] trms_sst_proj_subset(1)[of n]
  unfolding transaction_fresh_def Let_def transaction_strand_def by auto
qed

lemma transaction_proj_vars_subset:
 "vars_transaction (transaction_proj n T) ⊆ vars_transaction T"
proof -
 obtain A B C D E F where "T = Transaction A B C D E F" by (cases T) simp
 thus ?thesis
  using transaction_proj.simps[of n A B C D E F]
        sst_vars_proj_subset(3)[of n "transaction_strand T"]
  unfolding transaction_fresh_def Let_def transaction_strand_def by simp
qed

end
```

2.2 Term Abstraction (Term_Abstraction)

```
theory Term_Abstraction
 imports Transactions
begin
```

2.2.1 Definitions

```
fun to_abs ("α₀") where
  "α₀ [] _ = {}"
| "α₀ ((Fun (Val m) [],Fun (Set s) S)#D) n =
    (if m = n then insert s (α₀ D n) else α₀ D n)"
| "α₀ (_#D) n = α₀ D n"

fun abs_apply_term (infixl "·α" 67) where
  "Var x ·α α = Var x"
| "Fun (Val n) T ·α α = Fun (Abs (α n)) (map (λt. t ·α α) T)"
| "Fun f T ·α α = Fun f (map (λt. t ·α α) T)"

definition abs_apply_list (infixl "·αlist" 67) where
  "M ·αlist α ≡ map (λt. t ·α α) M"

definition abs_apply_terms (infixl "·αset" 67) where
  "M ·αset α ≡ (λt. t ·α α) ‘ M"

definition abs_apply_pairs (infixl "·αpairs" 67) where
  "F ·αpairs α ≡ map (λ(s,t). (s ·α α, t ·α α)) F"

definition abs_apply_strand_step (infixl "·αstp" 67) where
  "s ·αstp α ≡ (case s of
    (l,send⟨t⟩) ⇒ (l,send⟨t ·α α⟩)
  | (l,receive⟨t⟩) ⇒ (l,receive⟨t ·α α⟩)
  | (l,⟨ac: t �doteq t'⟩) ⇒ (l,⟨ac: (t ·α α) �doteq (t' ·α α)⟩)
  | (l,insert⟨t,t'⟩) ⇒ (l,insert⟨t ·α α,t' ·α α⟩)
  | (l,delete⟨t,t'⟩) ⇒ (l,delete⟨t ·α α,t' ·α α⟩)
  | (l,⟨ac: t ∈ t'⟩) ⇒ (l,⟨ac: (t ·α α) ∈ (t' ·α α)⟩)
  | (l,∀X⟨∨≠: F ∨∉: F'⟩) ⇒ (l,∀X⟨∨≠: (F ·αpairs α) ∨∉: (F' ·αpairs α)⟩))"

definition abs_apply_strand (infixl "·αst" 67) where
  "S ·αst α ≡ map (λx. x ·αstp α) S"
```

2.2.2 Lemmata

```
lemma to_abs_alt_def:
  "α₀ D n = {s. ∃S. (Fun (Val n) [], Fun (Set s) S) ∈ set D}"
by (induct D n rule: to_abs.induct) auto

lemma abs_term_apply_const[simp]:
  "is_Val f ⟹ Fun f [] ·α a = Fun (Abs (a (the_Val f))) []"
  "¬is_Val f ⟹ Fun f [] ·α a = Fun f []"
by (cases f; auto)+

lemma abs_fv: "fv (t ·α a) = fv t"
by (induct t a rule: abs_apply_term.induct) auto

lemma abs_eq_if_no_Val:
  assumes "∀f ∈ funs_term t. ¬is_Val f"
  shows "t ·α a = t ·α b"
using assms
proof (induction t)
  case (Fun f T) thus ?case by (cases f) simp_all
qed simp

lemma abs_list_set_is_set_abs_set: "set (M ·αlist α) = (set M) ·αset α"
unfolding abs_apply_list_def abs_apply_terms_def by simp

lemma abs_set_empty[simp]: "{} ·αset α = {}"
unfolding abs_apply_terms_def by simp
```

```
lemma abs_in:
  assumes "t ∈ M"
  shows "t ·α α ∈ M ·αset α"
using assms unfolding abs_apply_terms_def
by (induct t α rule: abs_apply_term.induct) blast+

lemma abs_set_union: "(A ∪ B) ·αset a = (A ·αset a) ∪ (B ·αset a)"
unfolding abs_apply_terms_def
by auto

lemma abs_subterms: "subterms (t ·α α) = subterms t ·αset α"
proof (induction t)
  case (Fun f T) thus ?case by (cases f) (auto simp add: abs_apply_terms_def)
qed (simp add: abs_apply_terms_def)

lemma abs_subterms_in: "s ∈ subterms t ⟹ s ·α a ∈ subterms (t ·α a)"
proof (induction t)
  case (Fun f T) thus ?case by (cases f) auto
qed simp

lemma abs_ik_append: "(ikₛₛₜ (A@B) ·set I) ·αset a = (ikₛₛₜ A ·set I) ·αset a ∪ (ikₛₛₜ B ·set I) ·αset a"
unfolding abs_apply_terms_def ikₛₛₜ_def
by auto

lemma to_abs_in:
  assumes "(Fun (Val n) [], Fun (Set s) []) ∈ set D"
  shows "s ∈ α₀ D n"
using assms by (induct rule: to_abs.induct) auto

lemma to_abs_empty_iff_notin_db:
  "Fun (Val n) [] ·α α₀ D = Fun (Abs {}) [] ⟷ (∄s S. (Fun (Val n) [], Fun (Set s) S) ∈ set D)"
by (simp add: to_abs_alt_def)

lemma to_abs_list_insert:
  assumes "Fun (Val n) [] ≠ t"
  shows "α₀ D n = α₀ (List.insert (t,s) D) n"
using assms to_abs_alt_def[of D n] to_abs_alt_def[of "List.insert (t,s) D" n]
by auto

lemma to_abs_list_insert':
  "insert s (α₀ D n) = α₀ (List.insert (Fun (Val n) [], Fun (Set s) S) D) n"
using to_abs_alt_def[of D n]
      to_abs_alt_def[of "List.insert (Fun (Val n) [], Fun (Set s) S) D" n]
by auto

lemma to_abs_list_remove_all:
  assumes "Fun (Val n) [] ≠ t"
  shows "α₀ D n = α₀ (List.removeAll (t,s) D) n"
using assms to_abs_alt_def[of D n] to_abs_alt_def[of "List.removeAll (t,s) D" n]
by auto

lemma to_abs_list_remove_all':
  "α₀ D n - {s} = α₀ (filter (λd. ∄S. d = (Fun (Val n) [], Fun (Set s) S)) D) n"
using to_abs_alt_def[of D n]
      to_abs_alt_def[of "filter (λd. ∄S. d = (Fun (Val n) [], Fun (Set s) S)) D" n]
by auto

lemma to_abs_dbₛₛₜ_append:
  assumes "∀u s. insert⟨u, s⟩ ∈ set B ⟶ Fun (Val n) [] ≠ u · 𝓘"
    and "∀u s. delete⟨u, s⟩ ∈ set B ⟶ Fun (Val n) [] ≠ u · 𝓘"
  shows "α₀ (db'ₛₛₜ A 𝓘 D) n = α₀ (db'ₛₛₜ (A@B) 𝓘 D) n"
using assms
proof (induction B rule: List.rev_induct)
```

```
  case (snoc b B)
  hence IH: "α₀ (db'_sst A I D) n = α₀ (db'_sst (A@B) I D) n" by auto
  have *: "∀u s. b = insert⟨u,s⟩ ⟶ Fun (Val n) [] ≠ u · I"
          "∀u s. b = delete⟨u,s⟩ ⟶ Fun (Val n) [] ≠ u · I"
    using snoc.prems by simp_all
  show ?case
  proof (cases b)
    case (Insert u s)
    hence **: "db'_sst (A@B@[b]) I D = List.insert (u · I,s · I) (db'_sst (A@B) I D)"
      using db_sst_append[of "A@B" "[b]"] by simp
    have "Fun (Val n) [] ≠ u · I" using *(1) Insert by auto
    thus ?thesis using IH ** to_abs_list_insert by metis
  next
    case (Delete u s)
    hence **: "db'_sst (A@B@[b]) I D = List.removeAll (u · I,s · I) (db'_sst (A@B) I D)"
      using db_sst_append[of "A@B" "[b]"] by simp
    have "Fun (Val n) [] ≠ u · I" using *(2) Delete by auto
    thus ?thesis using IH ** to_abs_list_remove_all by metis
  qed (simp_all add: db_sst_no_upd_append[of "[b]" "A@B"] IH)
qed simp

lemma to_abs_neq_imp_db_update:
  assumes "α₀ (db_sst A I) n ≠ α₀ (db_sst (A@B) I) n"
  shows "∃u s. u · I = Fun (Val n) [] ∧ (insert⟨u,s⟩ ∈ set B ∨ delete⟨u,s⟩ ∈ set B)"
proof -
  { fix D have ?thesis when "α₀ D n ≠ α₀ (db'_sst B I D) n" using that
    proof (induction B I D rule: db'_sst.induct)
      case 2 thus ?case
        by (metis db'_sst.simps(2) list.set_intros(1,2) subst_apply_pair_pair to_abs_list_insert)
    next
      case 3 thus ?case
        by (metis db'_sst.simps(3) list.set_intros(1,2) subst_apply_pair_pair to_abs_list_remove_all)
    qed simp_all
  } thus ?thesis using assms by (metis db_sst_append db_sst_def)
qed

lemma abs_term_subst_eq:
  fixes δ ϑ::"(('a,'b,'c) prot_fun, ('d,'e prot_atom) term × nat) subst"
  assumes "∀x ∈ fv t. δ x ·α a = ϑ x ·α b"
    and "∄n T. Fun (Val n) T ∈ subterms t"
  shows "t · δ ·α a = t · ϑ ·α b"
using assms
proof (induction t)
  case (Fun f T) thus ?case
  proof (cases f)
    case (Val n)
    hence False using Fun.prems(2) by blast
    thus ?thesis by metis
  qed auto
qed simp

lemma abs_term_subst_eq':
  fixes δ ϑ::"(('a,'b,'c) prot_fun, ('d,'e prot_atom) term × nat) subst"
  assumes "∀x ∈ fv t. δ x ·α a = ϑ x"
    and "∄n T. Fun (Val n) T ∈ subterms t"
  shows "t · δ ·α a = t · ϑ"
using assms
proof (induction t)
  case (Fun f T) thus ?case
  proof (cases f)
    case (Val n)
    hence False using Fun.prems(2) by blast
    thus ?thesis by metis
```

```
  qed auto
qed simp

lemma abs_val_in_funs_term:
  assumes "f ∈ funs_term t" "is_Val f"
  shows "Abs (α (the_Val f)) ∈ funs_term (t ·α α)"
using assms by (induct t α rule: abs_apply_term.induct) auto

end
```

2.3 Stateful Protocol Model (Stateful_Protocol_Model)

```
theory Stateful_Protocol_Model
  imports Stateful_Protocol_Composition_and_Typing.Stateful_Compositionality
          Transactions Term_Abstraction
begin
```

2.3.1 Locale Setup

```
locale stateful_protocol_model =
  fixes arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ (('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  assumes Ana_f_assm1: "∀f. let (K, M) = Ana_f f in (∀k ∈ subterms_set (set K).
      is_Fun k ⟶ (is_Fu (the_Fun k)) ∧ length (args k) = arity_f (the_Fu (the_Fun k)))"
    and Ana_f_assm2: "∀f. let (K, M) = Ana_f f in ∀i ∈ fv_set (set K) ∪ set M. i < arity_f f"
    and public_f_assm: "∀f. arity_f f > (0::nat) ⟶ public_f f"
    and Γ_f_assm: "∀f. arity_f f = (0::nat) ⟶ Γ_f f ≠ None"
    and label_witness_assm: "label_witness1 ≠ label_witness2"
begin

lemma Ana_f_assm1_alt:
  assumes "Ana_f f = (K,M)" "k ∈ subterms_set (set K)"
  shows "(∃x. k = Var x) ∨ (∃h T. k = Fun (Fu h) T ∧ length T = arity_f h)"
proof (cases k)
  case (Fun g T)
  let ?P = "λk. is_Fun k ⟶ is_Fu (the_Fun k) ∧ length (args k) = arity_f (the_Fu (the_Fun k))"
  let ?Q = "λK M. ∀k ∈ subterms_set (set K). ?P k"

  have "?Q (fst (Ana_f f)) (snd (Ana_f f))" using Ana_f_assm1 split_beta[of ?Q "Ana_f f"] by meson
  hence "?Q K M" using assms(1) by simp
  hence "?P k" using assms(2) by blast
  thus ?thesis using Fun by (cases g) auto
qed simp

lemma Ana_f_assm2_alt:
  assumes "Ana_f f = (K,M)" "i ∈ fv_set (set K) ∪ set M"
  shows "i < arity_f f"
using Ana_f_assm2 assms by fastforce
```

2.3.2 Definitions

```
fun arity where
  "arity (Fu f) = arity_f f"
| "arity (Set s) = arity_s s"
| "arity (Val _) = 0"
| "arity (Abs _) = 0"
| "arity Pair = 2"
```

```
| "arity (Attack _) = 0"
| "arity OccursFact = 2"
| "arity OccursSec = 0"
| "arity (PubConstAtom _ _) = 0"
| "arity (PubConstSetType _) = 0"
| "arity (PubConstAttackType _) = 0"
| "arity (PubConstBottom _) = 0"
| "arity (PubConstOccursSecType _) = 0"

fun public where
  "public (Fu f) = public_f f"
| "public (Set s) = (arity_s s > 0)"
| "public (Val n) = snd n"
| "public (Abs _) = False"
| "public Pair = True"
| "public (Attack _) = False"
| "public OccursFact = True"
| "public OccursSec = False"
| "public (PubConstAtom _ _) = True"
| "public (PubConstSetType _) = True"
| "public (PubConstAttackType _) = True"
| "public (PubConstBottom _) = True"
| "public (PubConstOccursSecType _) = True"

fun Ana where
  "Ana (Fun (Fu f) T) = (
    if arity_f f = length T ∧ arity_f f > 0
    then let (K,M) = Ana_f f in (K ·_list (!) T, map ((!) T) M)
    else ([], []))"
| "Ana _ = ([], [])"

definition Γ_v where
  "Γ_v v ≡ (
    if (∀t ∈ subterms (fst v).
          case t of (TComp f T) ⇒ arity f > 0 ∧ arity f = length T | _ ⇒ True)
    then fst v
    else TAtom Bottom)"

fun Γ where
  "Γ (Var v) = Γ_v v"
| "Γ (Fun f T) = (
    if arity f = 0
    then case f of
      (Fu g) ⇒ TAtom (case Γ_f g of Some a ⇒ Atom a | None ⇒ Bottom)
    | (Val _) ⇒ TAtom Value
    | (Abs _) ⇒ TAtom Value
    | (Set _) ⇒ TAtom SetType
    | (Attack _) ⇒ TAtom AttackType
    | OccursSec ⇒ TAtom OccursSecType
    | (PubConstAtom a _) ⇒ TAtom (Atom a)
    | (PubConstSetType _) ⇒ TAtom SetType
    | (PubConstAttackType _) ⇒ TAtom AttackType
    | (PubConstBottom _) ⇒ TAtom Bottom
    | (PubConstOccursSecType _) ⇒ TAtom OccursSecType
    | _ ⇒ TAtom Bottom
    else TComp f (map Γ T))"

lemma Γ_consts_simps[simp]:
  "arity_f g = 0 ⟹ Γ (Fun (Fu g) []) = TAtom (case Γ_f g of Some a ⇒ Atom a | None ⇒ Bottom)"
  "Γ (Fun (Val n) []) = TAtom Value"
  "Γ (Fun (Abs b) []) = TAtom Value"
  "arity_s s = 0 ⟹ Γ (Fun (Set s) []) = TAtom SetType"
  "Γ (Fun (Attack x) []) = TAtom AttackType"
```

```
  "Γ (Fun OccursSec []) = TAtom OccursSecType"
  "Γ (Fun (PubConstAtom a t) []) = TAtom (Atom a)"
  "Γ (Fun (PubConstSetType t) []) = TAtom SetType"
  "Γ (Fun (PubConstAttackType t) []) = TAtom AttackType"
  "Γ (Fun (PubConstBottom t) []) = TAtom Bottom"
  "Γ (Fun (PubConstOccursSecType t) []) = TAtom OccursSecType"
by simp+

lemma Γ_Set_simps[simp]:
  "arity_s s ≠ 0 ⟹ Γ (Fun (Set s) T) = TComp (Set s) (map Γ T)"
  "Γ (Fun (Set s) T) = TAtom SetType ∨ Γ (Fun (Set s) T) = TComp (Set s) (map Γ T)"
  "Γ (Fun (Set s) T) ≠ TAtom Value"
  "Γ (Fun (Set s) T) ≠ TAtom (Atom a)"
  "Γ (Fun (Set s) T) ≠ TAtom AttackType"
  "Γ (Fun (Set s) T) ≠ TAtom OccursSecType"
  "Γ (Fun (Set s) T) ≠ TAtom Bottom"
by auto
```

2.3.3 Locale Interpretations

```
lemma Ana_Fu_cases:
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Ana_f g = (K',M')"
  shows "(K,M) = (if arity_f g = length T ∧ arity_f g > 0
                then (K' ·_list (!) T, map ((!) T) M')
                else ([],[]))" (is ?A)
    and "(K,M) = (K' ·_list (!) T, map ((!) T) M') ∨ (K,M) = ([],[])" (is ?B)
proof -
  show ?A using assms by (cases "arity_f g = length T ∧ arity_f g > 0") auto
  thus ?B by metis
qed

lemma Ana_Fu_intro:
  assumes "arity_f f = length T" "arity_f f > 0"
    and "Ana_f f = (K',M')"
  shows "Ana (Fun (Fu f) T) = (K' ·_list (!) T, map ((!) T) M')"
using assms by simp

lemma Ana_Fu_elim:
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Ana_f g = (K',M')"
    and "(K,M) ≠ ([],[])"
  shows "arity_f g = length T" (is ?A)
    and "(K,M) = (K' ·_list (!) T, map ((!) T) M')" (is ?B)
proof -
  show ?A using assms by force
  moreover have "arity_f g > 0" using assms by force
  ultimately show ?B using assms by auto
qed

lemma Ana_nonempty_inv:
  assumes "Ana t ≠ ([],[])"
  shows "∃f T. t = Fun (Fu f) T ∧ arity_f f = length T ∧ arity_f f > 0 ∧
              (∃K M. Ana_f f = (K, M) ∧ Ana t = (K ·_list (!) T, map ((!) T) M))"
using assms
proof (induction t rule: Ana.induct)
  case (1 f T)
  hence *: "arity_f f = length T" "0 < arity_f f"
          "Ana (Fun (Fu f) T) = (case Ana_f f of (K, M) ⇒ (K ·_list (!) T, map ((!) T) M))"
    using Ana.simps(1)[of f T] unfolding Let_def by metis+
```

```
  obtain K M where **: "Ana_f f = (K, M)" by (metis surj_pair)
  hence "Ana (Fun (Fu f) T) = (K ·_list (!) T, map ((!) T) M)" using *(3) by simp
  thus ?case using ** *(1,2) by blast
qed simp_all

lemma assm1:
  assumes "Ana t = (K,M)"
  shows "fv_set (set K) ⊆ fv t"
using assms
proof (induction t rule: term.induct)
  case (Fun f T)
  have aux: "fv_set (set K ·_set (!) T) ⊆ fv_set (set T)"
    when K: "∀i ∈ fv_set (set K). i < length T"
    for K::"(('fun,'atom,'sets) prot_fun, nat) term list"
  proof
    fix x assume "x ∈ fv_set (set K ·_set (!) T)"
    then obtain k where k: "k ∈ set K" "x ∈ fv (k · (!) T)" by moura
    have "∀i ∈ fv k. i < length T" using K k(1) by simp
    thus "x ∈ fv_set (set T)"
      by (metis (no_types, lifting) k(2) contra_subsetD fv_set_mono image_subsetI nth_mem
                                    subst_apply_fv_unfold)
  qed

  { fix g assume f: "f = Fu g" and K: "K ≠ []"
    obtain K' M' where *: "Ana_f g = (K',M')" by moura
    have "(K, M) ≠ ([], [])" using K by simp
    hence "(K, M) = (K' ·_list (!) T, map ((!) T) M')" "arity_f g = length T"
      using Ana_Fu_cases(1)[OF Fun.prems f *]
      by presburger+
    hence ?case using aux[of K'] Ana_f_assm2_alt[OF *] by auto
  } thus ?case using Fun by (cases f) fastforce+
qed simp

lemma assm2:
  assumes "Ana t = (K,M)"
  and "⋀g S'. Fun g S' ⊑ t ⟹ length S' = arity g"
  and "k ∈ set K"
  and "Fun f T' ⊑ k"
  shows "length T' = arity f"
using assms
proof (induction t rule: term.induct)
  case (Fun g T)
  obtain h where 2: "g = Fu h"
    using Fun.prems(1,3) by (cases g) auto
  obtain K' M' where 1: "Ana_f h = (K',M')" by moura
  have "(K,M) ≠ ([],[])" using Fun.prems(3) by auto
  hence "(K,M) = (K' ·_list (!) T, map ((!) T) M')"
        "⋀i. i ∈ fv_set (set K') ∪ set M' ⟹ i < length T"
    using Ana_Fu_cases(1)[OF Fun.prems(1) 2 1] Ana_f_assm2_alt[OF 1]
    by presburger+
  hence "K = K' ·_list (!) T" and 3: "∀i∈fv_set (set K'). i < length T" by simp_all
  then obtain k' where k': "k' ∈ set K'" "k = k' · (!) T" using Fun.prems(3) by moura
  hence 4: "Fun f T' ∈ subterms (k' · (!) T)" "fv k' ⊆ fv_set (set K')"
    using Fun.prems(4) by auto
  show ?case
  proof (cases "∃i ∈ fv k'. Fun f T' ∈ subterms (T ! i)")
    case True
    hence "Fun f T' ∈ subterms_set (set T)" using k' Fun.prems(4) 3 by auto
    thus ?thesis using Fun.prems(2) by auto
  next
    case False
    then obtain S where "Fun f S ∈ subterms k'" "Fun f T' = Fun f S · (!) T"
      using k'(2) Fun.prems(4) subterm_subst_not_img_subterm by force
```

```
    thus ?thesis using Ana_f_assm1_alt[OF 1, of "Fun f S"] k'(1) by (cases f) auto
  qed
qed simp

lemma assm4:
  assumes "Ana (Fun f T) = (K, M)"
  shows "set M ⊆ set T"
using assms
proof (cases f)
  case (Fu g)
  obtain K' M' where *: "Ana_f g = (K',M')" by moura
  have "M = [] ∨ (arity_f g = length T ∧ M = map ((!) T) M')"
    using Ana_Fu_cases(1)[OF assms Fu *]
    by (meson prod.inject)
  thus ?thesis using Ana_f_assm2_alt[OF *] by auto
qed auto

lemma assm5: "Ana t = (K,M) ⟹ K ≠ [] ∨ M ≠ [] ⟹ Ana (t · δ) = (K ·_list δ, M ·_list δ)"
proof (induction t rule: term.induct)
  case (Fun f T) thus ?case
  proof (cases f)
    case (Fu g)
    obtain K' M' where *: "Ana_f g = (K',M')" by moura
    have **: "K = K' ·_list (!) T" "M = map ((!) T) M'"
             "arity_f g = length T" "∀i ∈ fv_set (set K') ∪ set M'. i < arity_f g" "0 < arity_f g"
      using Fun.prems(2) Ana_Fu_cases(1)[OF Fun.prems(1) Fu *] Ana_f_assm2_alt[OF *]
      by (meson prod.inject)+

    have ***: "∀i ∈ fv_set (set K'). i < length T" "∀i ∈ set M'. i < length T" using **(3,4) by auto

    have "K ·_list δ = K' ·_list (!) (map (λt. t · δ) T)"
         "M ·_list δ = map ((!) (map (λt. t · δ) T)) M'"
      using subst_idx_map[OF ***(2), of δ]
            subst_idx_map'[OF ***(1), of δ]
            **(1,2)
      by fast+
    thus ?thesis using Fu * **(3,5) by auto
  qed auto
qed simp

sublocale intruder_model arity public Ana
apply unfold_locales
by (metis assm1, metis assm2, rule Ana.simps, metis assm4, metis assm5)

adhoc_overloading INTRUDER_SYNTH intruder_synth
adhoc_overloading INTRUDER_DEDUCT intruder_deduct

lemma assm6: "arity c = 0 ⟹ ∃a. ∀X. Γ (Fun c X) = TAtom a" by (cases c) auto

lemma assm7: "0 < arity f ⟹ Γ (Fun f T) = TComp f (map Γ T)" by auto

lemma assm8: "infinite {c. Γ (Fun c []::('fun,'atom,'sets) prot_term) = TAtom a ∧ public c}"
  (is "?P a")
proof -
  let ?T = "λf. (range f)::('fun,'atom,'sets) prot_fun set"
  let ?A = "λf. ∀x::nat ∈ UNIV. ∀y::nat ∈ UNIV. (f x = f y) = (x = y)"
  let ?B = "λf. ∀x::nat ∈ UNIV. f x ∈ ?T f"
  let ?C = "λf. ∀y::('fun,'atom,'sets) prot_fun ∈ ?T f. ∃x ∈ UNIV. y = f x"
  let ?D = "λf b. ?T f ⊆ {c. Γ (Fun c []::('fun,'atom,'sets) prot_term) = TAtom b ∧ public c}"

  have sub_lmm: "?P b" when "?A f" "?C f" "?C f" "?D f b" for b f
  proof -
    have "∃g::nat ⇒ ('fun,'atom,'sets) prot_fun. bij_betw g UNIV (?T f)"
```

```
      using bij_betwI'[of UNIV f "?T f"] that(1,2,3) by blast
    hence "infinite (?T f)" by (metis nat_not_finite bij_betw_finite)
    thus ?thesis using infinite_super[OF that(4)] by blast
  qed

  show ?thesis
  proof (cases a)
    case (Atom b) thus ?thesis using sub_lmm[of "PubConstAtom b" a] by force
  next
    case Value thus ?thesis using sub_lmm[of "λn. Val (n,True)" a] by force
  next
    case SetType thus ?thesis using sub_lmm[of PubConstSetType a] by fastforce
  next
    case AttackType thus ?thesis using sub_lmm[of PubConstAttackType a] by fastforce
  next
    case Bottom thus ?thesis using sub_lmm[of PubConstBottom a] by fastforce
  next
    case OccursSecType thus ?thesis using sub_lmm[of PubConstOccursSecType a] by fastforce
  qed
qed

lemma assm9: "TComp f T ⊑ Γ t ⟹ arity f > 0"
proof (induction t rule: term.induct)
  case (Var x)
  hence "Γ (Var x) ≠ TAtom Bottom" by force
  hence "∀t ∈ subterms (fst x). case t of
           TComp f T ⇒ arity f > 0 ∧ arity f = length T
         | _ ⇒ True"
    using Var Γ.simps(1)[of x] unfolding Γ_v_def by meson
  thus ?case using Var by (fastforce simp add: Γ_v_def)
next
  case (Fun g S)
  have "arity g ≠ 0" using Fun.prems Var_subtermeq assm6 by force
  thus ?case using Fun by (cases "TComp f T = TComp g (map Γ S)") auto
qed

lemma assm10: "wf_trm (Γ (Var x))"
unfolding wf_trm_def by (auto simp add: Γ_v_def)

lemma assm11: "arity f > 0 ⟹ public f" using public_f_assm by (cases f) auto

lemma assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))" by (simp add: Γ_v_def)

lemma assm13: "arity c = 0 ⟹ Ana (Fun c T) = ([],[])" by (cases c) simp_all

lemma assm14:
  assumes "Ana (Fun f T) = (K,M)"
  shows "Ana (Fun f T · δ) = (K ·_list δ, M ·_list δ)"
proof -
  show ?thesis
  proof (cases "(K, M) = ([],[])")
    case True
    { fix g assume f: "f = Fu g"
      obtain K' M' where "Ana_f g = (K',M')" by moura
      hence ?thesis using assms f True by auto
    } thus ?thesis using True assms by (cases f) auto
  next
    case False
    then obtain g where **: "f = Fu g" using assms by (cases f) auto
    obtain K' M' where *: "Ana_f g = (K',M')" by moura
    have ***: "K = K' ·_list (!) T" "M = map ((!) T) M'" "arity_f g = length T"
              "∀i ∈ fv_set (set K') ∪ set M'. i < arity_f g"
      using Ana_Fu_cases(1)[OF assms ** *] False Ana_f_assm2_alt[OF *]
```

```
    by (meson prod.inject)+
  have ****: "∀i∈fv_set (set K'). i < length T" "∀i∈set M'. i < length T" using ***(3,4) by auto
  have "K ·list δ = K' ·list (!) (map (λt. t · δ) T)"
       "M ·list δ = map ((!) (map (λt. t · δ) T)) M'"
    using subst_idx_map[OF ****(2), of δ]
          subst_idx_map'[OF ****(1), of δ]
          ***(1,2)
    by auto
  thus ?thesis using assms * ** ***(3) by auto
 qed
qed

sublocale labeled_stateful_typed_model' arity public Ana Γ Pair label_witness1 label_witness2
by unfold_locales
   (metis assm6, metis assm7, metis assm8, metis assm9,
    rule assm10, metis assm11, rule arity.simps(5), metis assm14,
    metis assm12, metis assm13, metis assm14, rule label_witness_assm)
```

2.3.4 Minor Lemmata

```
lemma Γv_TAtom[simp]: "Γv (TAtom a, n) = TAtom a"
unfolding Γv_def by simp

lemma Γv_TAtom':
 assumes "a ≠ Bottom"
 shows "Γv (τ, n) = TAtom a ⟷ τ = TAtom a"
proof
 assume "Γv (τ, n) = TAtom a"
 thus "τ = TAtom a" by (metis (no_types, lifting) assms Γv_def fst_conv term.inject(1))
qed simp

lemma Γv_TAtom_inv:
 "Γv x = TAtom (Atom a) ⟹ ∃m. x = (TAtom (Atom a), m)"
 "Γv x = TAtom Value ⟹ ∃m. x = (TAtom Value, m)"
 "Γv x = TAtom SetType ⟹ ∃m. x = (TAtom SetType, m)"
 "Γv x = TAtom AttackType ⟹ ∃m. x = (TAtom AttackType, m)"
 "Γv x = TAtom OccursSecType ⟹ ∃m. x = (TAtom OccursSecType, m)"
by (metis Γv_TAtom' surj_pair prot_atom.distinct(7),
    metis Γv_TAtom' surj_pair prot_atom.distinct(15),
    metis Γv_TAtom' surj_pair prot_atom.distinct(21),
    metis Γv_TAtom' surj_pair prot_atom.distinct(25),
    metis Γv_TAtom' surj_pair prot_atom.distinct(30))

lemma Γv_TAtom'':
 "(fst x = TAtom (Atom a)) = (Γv x = TAtom (Atom a))" (is "?A = ?A'")
 "(fst x = TAtom Value) = (Γv x = TAtom Value)" (is "?B = ?B'")
 "(fst x = TAtom SetType) = (Γv x = TAtom SetType)" (is "?C = ?C'")
 "(fst x = TAtom AttackType) = (Γv x = TAtom AttackType)" (is "?D = ?D'")
 "(fst x = TAtom OccursSecType) = (Γv x = TAtom OccursSecType)" (is "?E = ?E'")
proof -
 have 1: "?A ⟹ ?A'" "?B ⟹ ?B'" "?C ⟹ ?C'" "?D ⟹ ?D'" "?E ⟹ ?E'"
   by (metis Γv_TAtom prod.collapse)+

 have 2: "?A' ⟹ ?A" "?B' ⟹ ?B" "?C' ⟹ ?C" "?D' ⟹ ?D" "?E' ⟹ ?E"
   using Γv_TAtom Γv_TAtom_inv(1) apply fastforce
   using Γv_TAtom Γv_TAtom_inv(2) apply fastforce
   using Γv_TAtom Γv_TAtom_inv(3) apply fastforce
   using Γv_TAtom Γv_TAtom_inv(4) apply fastforce
   using Γv_TAtom Γv_TAtom_inv(5) by fastforce

 show "?A = ?A'" "?B = ?B'" "?C = ?C'" "?D = ?D'" "?E = ?E'"
   using 1 2 by metis+
qed
```

```
lemma Γv_Var_image:
  "Γv ‘ X = Γ ‘ Var ‘ X"
by force

lemma Γ_Fu_const:
  assumes "arityf g = 0"
  shows "∃a. Γ (Fun (Fu g) T) = TAtom (Atom a)"
proof -
  have "Γf g ≠ None" using assms Γf_assm by blast
  thus ?thesis using assms by force
qed

lemma Fun_Value_type_inv:
  fixes T::"('fun,'atom,'sets) prot_term list"
  assumes "Γ (Fun f T) = TAtom Value"
  shows "(∃n. f = Val n) ∨ (∃bs. f = Abs bs)"
proof -
  have *: "arity f = 0" by (metis const_type_inv assms)
  show ?thesis  using assms
  proof (cases f)
    case (Fu g)
    hence "arityf g = 0" using * by simp
    hence False using Fu Γ_Fu_const[of g T] assms by auto
    thus ?thesis by metis
  next
    case (Set s)
    hence "arity_s s = 0" using * by simp
    hence False using Set assms by auto
    thus ?thesis by metis
  qed simp_all
qed

lemma abs_Γ: "Γ t = Γ (t ·α α)"
by (induct t α rule: abs_apply_term.induct) auto

lemma Anaf_keys_not_pubval_terms:
  assumes "Anaf f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "¬is_Val g"
proof
  assume "is_Val g"
  then obtain n S where *: "Fun (Val n) S ∈ subterms_set (set K)"
    using assms(2) funs_term_Fun_subterm[OF assms(3)]
    by (cases g) auto
  show False using Anaf_assm1_alt[OF assms(1) *] by simp
qed

lemma Anaf_keys_not_abs_terms:
  assumes "Anaf f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "¬is_Abs g"
proof
  assume "is_Abs g"
  then obtain a S where *: "Fun (Abs a) S ∈ subterms_set (set K)"
    using assms(2) funs_term_Fun_subterm[OF assms(3)]
    by (cases g) auto
  show False using Anaf_assm1_alt[OF assms(1) *] by simp
qed

lemma Anaf_keys_not_pairs:
```

```
  assumes "Ana_f f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "g ≠ Pair"
proof
  assume "g = Pair"
  then obtain S where *: "Fun Pair S ∈ subterms_set (set K)"
    using assms(2) funs_term_Fun_subterm[OF assms(3)]
    by (cases g) auto
  show False using Ana_f_assm1_alt[OF assms(1) *] by simp
qed

lemma Ana_Fu_keys_funs_term_subset:
  fixes K::"('fun,'atom,'sets) prot_term list"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
  shows "⋃(funs_term ` set K) ⊆ ⋃(funs_term ` set K') ∪ funs_term (Fun (Fu f) S)"
proof -
  { fix k assume k: "k ∈ set K"
    then obtain k' where k':
        "k' ∈ set K'" "k = k' · (!) S" "arity_f f = length S"
        "subterms k' ⊆ subterms_set (set K')"
      using assms Ana_Fu_elim[OF assms(1) _ assms(2)] by fastforce

    have 1: "funs_term k' ⊆ ⋃(funs_term ` set K')" using k'(1) by auto

    have "i < length S" when "i ∈ fv k'" for i
      using that Ana_f_assm2_alt[OF assms(2), of i] k'(1,3)
      by auto
    hence 2: "funs_term (S ! i) ⊆ funs_term (Fun (Fu f) S)" when "i ∈ fv k'" for i
      using that by force

    have "funs_term k ⊆ ⋃(funs_term ` set K') ∪ funs_term (Fun (Fu f) S)"
      using funs_term_subst[of k' "(!) S"] k'(2) 1 2 by fast
  } thus ?thesis by blast
qed

lemma Ana_Fu_keys_not_pubval_terms:
  fixes k::"('fun,'atom,'sets) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). is_Val g ⟶ ¬public g"
  shows "∀g ∈ funs_term k. is_Val g ⟶ ¬public g"
using assms(3,4) Ana_f_keys_not_pubval_terms[OF assms(2)]
      Ana_Fu_keys_funs_term_subset[OF assms(1,2)]
by blast

lemma Ana_Fu_keys_not_abs_terms:
  fixes k::"('fun,'atom,'sets) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). ¬is_Abs g"
  shows "∀g ∈ funs_term k. ¬is_Abs g"
using assms(3,4) Ana_f_keys_not_abs_terms[OF assms(2)]
      Ana_Fu_keys_funs_term_subset[OF assms(1,2)]
by blast

lemma Ana_Fu_keys_not_pairs:
  fixes k::"('fun,'atom,'sets) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
```

```
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). g ≠ Pair"
  shows "∀g ∈ funs_term k. g ≠ Pair"
using assms(3,4) Ana_f_keys_not_pairs[OF assms(2)]
      Ana_Fu_keys_funs_term_subset[OF assms(1,2)]
by blast

lemma deduct_occurs_in_ik:
  fixes t::"('fun,'atom,'sets) prot_term"
  assumes t: "M ⊢ occurs t"
    and M: "∀s ∈ subterms_set M. OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
           "∀s ∈ subterms_set M. OccursSec ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
           "Fun OccursSec [] ∉ M"
  shows "occurs t ∈ M"
using private_fun_deduct_in_ik''[of M OccursFact "[Fun OccursSec [], t]" OccursSec] t M
by fastforce

lemma wellformed_transaction_sem_receives:
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_valid: "wellformed_transaction T"
    and I: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) I"
    and s: "receive⟨t⟩ ∈ set (unlabel (transaction_receive T ·_lsst ϑ))"
  shows "IK ⊢ t · I"
proof -
  let ?R = "unlabel (dual_lsst (transaction_receive T ·_lsst ϑ))"
  let ?S = "λA. unlabel (dual_lsst (A ·_lsst ϑ))"
  let ?S' = "?S (transaction_receive T)"

  obtain l B s where B:
      "(l,send⟨t⟩) = dual_lsstp ((l,s) ·_lsstp ϑ)"
      "prefix ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ]) (transaction_receive T ·_lsst ϑ)"
    using s dual_lsst_unlabel_steps_iff(2)[of t "transaction_receive T ·_lsst ϑ"]
          dual_lsst_in_set_prefix_obtain_subst[of "send⟨t⟩" "transaction_receive T" ϑ]
    by blast

  have 1: "unlabel (dual_lsst ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ])) = unlabel (dual_lsst (B ·_lsst ϑ))@[send⟨t⟩]"
    using B(1) unlabel_append dual_lsstp_subst dual_lsst_subst singleton_lst_proj(4)
          dual_lsst_subst_snoc subst_lsst_append subst_lsst_singleton
    by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps )

  have "strand_sem_stateful IK DB ?S' I"
    using I strand_sem_append_stateful[of IK DB _ _ I] transaction_dual_subst_unfold[of T ϑ]
    by fastforce
  hence "strand_sem_stateful IK DB (unlabel (dual_lsst (B ·_lsst ϑ))@[send⟨t⟩]) I"
    using B 1 unfolding prefix_def unlabel_def
    by (metis dual_lsst_def map_append strand_sem_append_stateful)
  hence t_deduct: "IK ∪ (ik_lsst (dual_lsst (B ·_lsst ϑ)) ·_set I) ⊢ t · I"
    using strand_sem_append_stateful[of IK DB "unlabel (dual_lsst (B ·_lsst ϑ))" "[send⟨t⟩]" I]
    by simp

  have "∀s ∈ set (unlabel (transaction_receive T)). ∃t. s = receive⟨t⟩"
    using T_valid wellformed_transaction_unlabel_cases(1)[OF T_valid] by auto
  moreover { fix A::"('fun,'atom,'sets,'lbl) prot_strand" and ϑ
    assume "∀s ∈ set (unlabel A). ∃t. s = receive⟨t⟩"
    hence "∀s ∈ set (unlabel (A ·_lsst ϑ)). ∃t. s = receive⟨t⟩"
    proof (induction A)
      case (Cons a A) thus ?case using subst_lsst_cons[of a A ϑ] by (cases a) auto
    qed simp
    hence "∀s ∈ set (unlabel (A ·_lsst ϑ)). ∃t. s = receive⟨t⟩"
      by (simp add: list.pred_set is_Receive_def)
    hence "∀s ∈ set (unlabel (dual_lsst (A ·_lsst ϑ))). ∃t. s = send⟨t⟩"
      by (metis dual_lsst_memberD dual_lsstp_inv(2) unlabel_in unlabel_mem_has_label)
  }
```

```
  ultimately have "∀s ∈ set ?R. ∃t. s = send⟨t⟩" by simp
  hence "ik_sst ?R = {}" unfolding unlabel_def ik_sst_def by fast
  hence "ik_lsst (dual_lsst (B ·_lsst ϑ)) = {}"
    using B(2) 1 ik_sst_append dual_lsst_append
    by (metis (no_types, lifting) Un_empty map_append prefix_def unlabel_def)
  thus ?thesis using t_deduct by simp
qed

lemma wellformed_transaction_sem_selects:
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and "select⟨t,u⟩ ∈ set (unlabel (transaction_selects T ·_lsst ϑ))"
  shows "(t · ℐ, u · ℐ) ∈ DB"
proof -
  let ?s = "select⟨t,u⟩"
  let ?R = "transaction_receive T@transaction_selects T"
  let ?R' = "unlabel (dual_lsst (?R ·_lsst ϑ))"
  let ?S = "λA. unlabel (dual_lsst (A ·_lsst ϑ))"
  let ?S' = "?S (transaction_receive T)@?S (transaction_selects T)"
  let ?P = "λa. is_Receive a ∨ is_Assignment a"
  let ?Q = "λa. is_Send a ∨ is_Assignment a"

  have s: "?s ∈ set (unlabel (?R ·_lsst ϑ))"
    using assms(3) subst_lsst_append[of "transaction_receive T"]
          unlabel_append[of "transaction_receive T ·_lsst ϑ"]
    by auto

  obtain l B s where B:
      "(l,?s) = dual_lsstp ((l,s) ·_lsstp ϑ)"
      "prefix ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ]) (?R ·_lsst ϑ)"
    using s dual_lsst_unlabel_steps_iff(6)[of assign t u]
          dual_lsst_in_set_prefix_obtain_subst[of ?s ?R ϑ]
    by blast

  have 1: "unlabel (dual_lsst ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ])) = unlabel (dual_lsst (B ·_lsst ϑ))@[?s]"
    using B(1) unlabel_append dual_lsstp_subst dual_lsst_subst singleton_lst_proj(4)
          dual_lsst_subst_snoc subst_lsst_append subst_lsst_singleton
    by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps)

  have "strand_sem_stateful IK DB ?S' ℐ"
    using ℐ strand_sem_append_stateful[of IK DB _ _ ℐ] transaction_dual_subst_unfold[of T ϑ]
    by fastforce
  hence "strand_sem_stateful IK DB (unlabel (dual_lsst (B ·_lsst ϑ))@[?s]) ℐ"
    using B 1 strand_sem_append_stateful subst_lsst_append
    unfolding prefix_def unlabel_def dual_lsst_def
    by (metis (no_types) map_append)
  hence in_db: "(t · ℐ, u · ℐ) ∈ dbupd_sst (unlabel (dual_lsst (B ·_lsst ϑ))) ℐ DB"
    using strand_sem_append_stateful[of IK DB "unlabel (dual_lsst (B ·_lsst ϑ))" "[?s]" ℐ]
    by simp

  have "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ?Q a"
  proof
    fix a assume a: "a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ)))"

    have "∀a ∈ set (unlabel ?R). ?P a"
      using wellformed_transaction_unlabel_cases(1)[OF T_valid]
            wellformed_transaction_unlabel_cases(2)[OF T_valid]
      unfolding unlabel_def
      by fastforce
    hence "∀a ∈ set (unlabel (?R ·_lsst ϑ)). ?P a"
      using stateful_strand_step_cases_subst(2,8)[of _ ϑ] subst_lsst_unlabel[of ?R ϑ]
      by (simp add: subst_apply_stateful_strand_def del: unlabel_append)
    hence B_P: "∀a ∈ set (unlabel (B ·_lsst ϑ)). ?P a"
```

```
    using unlabel_mono[OF set_mono_prefix[OF append_prefixD[OF B(2)]]]
    by blast

  obtain l where "(l,a) ∈ set (dual_lsst (B ·_lsst ϑ))"
    using a by (meson unlabel_mem_has_label)
  then obtain b where b: "(l,b) ∈ set (B ·_lsst ϑ)" "dual_lsstp (l,b) = (l,a)"
    using dual_lsst_memberD by blast
  hence "?P b" using B_P unfolding unlabel_def by fastforce
  thus "?Q a" using dual_lsstp_inv[OF b(2)] by (cases b) auto
 qed
 hence "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ¬is_Insert a ∧ ¬is_Delete a" by fastforce
 thus ?thesis using dbupd_sst_no_upd[of "unlabel (dual_lsst (B ·_lsst ϑ))" I DB] in_db by simp
qed

lemma wellformed_transaction_sem_pos_checks:
 assumes T_valid: "wellformed_transaction T"
   and I: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) I"
   and "⟨t in u⟩ ∈ set (unlabel (transaction_checks T ·_lsst ϑ))"
 shows "(t · I, u · I) ∈ DB"
proof -
 let ?s = "⟨t in u⟩"
 let ?R = "transaction_receive T@transaction_selects T@transaction_checks T"
 let ?R' = "unlabel (dual_lsst (?R ·_lsst ϑ))"
 let ?S = "λA. unlabel (dual_lsst (A ·_lsst ϑ))"
 let ?S' = "?S (transaction_receive T)@?S (transaction_selects T)@?S (transaction_checks T)"
 let ?P = "λa. is_Receive a ∨ is_Assignment a ∨ is_Check a"
 let ?Q = "λa. is_Send a ∨ is_Assignment a ∨ is_Check a"

 have s: "?s ∈ set (unlabel (?R ·_lsst ϑ))"
   using assms(3) subst_lsst_append[of "transaction_receive T@transaction_selects T"]
         unlabel_append[of "transaction_receive T@transaction_selects T ·_lsst ϑ"]
   by auto

 obtain l B s where B:
     "(l,?s) = dual_lsstp ((l,s) ·_lsstp ϑ)"
     "prefix ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ]) (?R ·_lsst ϑ)"
   using s dual_lsst_unlabel_steps_iff(6)[of check t u]
         dual_lsst_in_set_prefix_obtain_subst[of ?s ?R ϑ]
   by blast

 have 1: "unlabel (dual_lsst ((B ·_lsst ϑ)@[(l,s) ·_lsstp ϑ])) = unlabel (dual_lsst (B ·_lsst ϑ))@[?s]"
   using B(1) unlabel_append dual_lsstp_subst dual_lsst_subst singleton_lst_proj(4)
         dual_lsst_subst_snoc subst_lsst_append subst_lsst_singleton
   by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps )

 have "strand_sem_stateful IK DB ?S' I"
   using I strand_sem_append_stateful[of IK DB _ _ I] transaction_dual_subst_unfold[of T ϑ]
   by fastforce
 hence "strand_sem_stateful IK DB (unlabel (dual_lsst (B ·_lsst ϑ))@[?s]) I"
   using B 1 strand_sem_append_stateful subst_lsst_append
   unfolding prefix_def unlabel_def dual_lsst_def
   by (metis (no_types) map_append)
 hence in_db: "(t · I, u · I) ∈ dbupd_sst (unlabel (dual_lsst (B ·_lsst ϑ))) I DB"
   using strand_sem_append_stateful[of IK DB "unlabel (dual_lsst (B ·_lsst ϑ))" "[?s]" I]
   by simp

 have "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ?Q a"
 proof
   fix a assume a: "a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ)))"

   have "∀a ∈ set (unlabel ?R). ?P a"
     using wellformed_transaction_unlabel_cases(1,2,3)[OF T_valid]
     unfolding unlabel_def
```

```
    by fastforce
  hence "∀a ∈ set (unlabel (?R ·lsst ϑ)). ?P a"
    using stateful_strand_step_cases_subst(2,8,9)[of _ ϑ] subst_lsst_unlabel[of ?R ϑ]
    by (simp add: subst_apply_stateful_strand_def del: unlabel_append)
  hence B_P: "∀a ∈ set (unlabel (B ·lsst ϑ)). ?P a"
    using unlabel_mono[OF set_mono_prefix[OF append_prefixD[OF B(2)]]]
    by blast

  obtain l where "(l,a) ∈ set (duallsst (B ·lsst ϑ))"
    using a by (meson unlabel_mem_has_label)
  then obtain b where b: "(l,b) ∈ set (B ·lsst ϑ)" "duallsstp (l,b) = (l,a)"
    using duallsst_memberD by blast
  hence "?P b" using B_P unfolding unlabel_def by fastforce
  thus "?Q a" using duallsstp_inv[OF b(2)] by (cases b) auto
 qed
 hence "∀a ∈ set (unlabel (duallsst (B ·lsst ϑ))). ¬is_Insert a ∧ ¬is_Delete a" by fastforce
 thus ?thesis using dbupdsst_no_upd[of "unlabel (duallsst (B ·lsst ϑ))" I DB] in_db by simp
qed

lemma wellformed_transaction_sem_neg_checks:
 assumes T_valid: "wellformed_transaction T"
   and I: "strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ϑ))) I"
   and "NegChecks X [] [(t,u)] ∈ set (unlabel (transaction_checks T ·lsst ϑ))"
 shows "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ (t · δ · I, u · δ · I) ∉ DB" (is ?A)
   and "X = [] ⟹ (t · I, u · I) ∉ DB" (is "?B ⟹ ?B'")
proof -
 let ?s = "NegChecks X [] [(t,u)]"
 let ?R = "transaction_receive T@transaction_selects T@transaction_checks T"
 let ?R' = "unlabel (duallsst (?R ·lsst ϑ))"
 let ?S = "λA. unlabel (duallsst (A ·lsst ϑ))"
 let ?S' = "?S (transaction_receive T)@?S (transaction_selects T)@?S (transaction_checks T)"
 let ?P = "λa. is_Receive a ∨ is_Assignment a ∨ is_Check a"
 let ?Q = "λa. is_Send a ∨ is_Assignment a ∨ is_Check a"
 let ?U = "λδ. subst_domain δ = set X ∧ ground (subst_range δ)"

 have s: "?s ∈ set (unlabel (?R ·lsst ϑ))"
   using assms(3) subst_lsst_append[of "transaction_receive T@transaction_selects T"]
         unlabel_append[of "transaction_receive T@transaction_selects T ·lsst ϑ"]
   by auto

 obtain l B s where B:
     "(l,?s) = duallsstp ((l,s) ·lsstp ϑ)"
     "prefix ((B ·lsst ϑ)@[(l,s) ·lsstp ϑ]) (?R ·lsst ϑ)"
   using s duallsst_unlabel_steps_iff(7)[of X "[]" "[(t,u)]"]
         duallsst_in_set_prefix_obtain_subst[of ?s ?R ϑ]
   by blast

 have 1: "unlabel (duallsst ((B ·lsst ϑ)@[(l,s) ·lsstp ϑ])) = unlabel (duallsst (B ·lsst ϑ))@[?s]"
   using B(1) unlabel_append duallsstp_subst duallsst_subst singleton_lst_proj(4)
         duallsst_subst_snoc subst_lsst_append subst_lsst_singleton
   by (metis (no_types, lifting) subst_apply_labeled_stateful_strand_step.simps)

 have "strand_sem_stateful IK DB ?S' I"
   using I strand_sem_append_stateful[of IK DB _ _ I] transaction_dual_subst_unfold[of T ϑ]
   by fastforce
 hence "strand_sem_stateful IK DB (unlabel (duallsst (B ·lsst ϑ))@[?s]) I"
   using B 1 strand_sem_append_stateful subst_lsst_append
   unfolding prefix_def unlabel_def duallsst_def
   by (metis (no_types) map_append)
 hence "negchecks_model I (dbupdsst (unlabel (duallsst (B ·lsst ϑ))) I DB) X [] [(t,u)]"
   using strand_sem_append_stateful[of IK DB "unlabel (duallsst (B ·lsst ϑ))" "[?s]" I]
   by fastforce
 hence in_db: "∀δ. ?U δ ⟶ (t · δ · I, u · δ · I) ∉ dbupdsst (unlabel (duallsst (B ·lsst ϑ))) I DB"
```

```
    unfolding negchecks_model_def
    by simp

  have "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ?Q a"
  proof
    fix a assume a: "a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ)))"

    have "∀a ∈ set (unlabel ?R). ?P a"
      using wellformed_transaction_unlabel_cases(1,2,3)[OF T_valid]
      unfolding unlabel_def
      by fastforce
    hence "∀a ∈ set (unlabel (?R ·_lsst ϑ)). ?P a"
      using stateful_strand_step_cases_subst(2,8,9)[of _ ϑ] subst_lsst_unlabel[of ?R ϑ]
      by (simp add: subst_apply_stateful_strand_def del: unlabel_append)
    hence B_P: "∀a ∈ set (unlabel (B ·_lsst ϑ)). ?P a"
      using unlabel_mono[OF set_mono_prefix[OF append_prefixD[OF B(2)]]]
      by blast

    obtain l where "(l,a) ∈ set (dual_lsst (B ·_lsst ϑ))"
      using a by (meson unlabel_mem_has_label)
    then obtain b where b: "(l,b) ∈ set (B ·_lsst ϑ)" "dual_lsstp (l,b) = (l,a)"
      using dual_lsst_memberD by blast
    hence "?P b" using B_P unfolding unlabel_def by fastforce
    thus "?Q a" using dual_lsstp_inv[OF b(2)] by (cases b) auto
  qed
  hence "∀a ∈ set (unlabel (dual_lsst (B ·_lsst ϑ))). ¬is_Insert a ∧ ¬is_Delete a" by fastforce
  thus ?A using dbupd_sst_no_upd[of "unlabel (dual_lsst (B ·_lsst ϑ))" I DB] in_db by simp
  moreover have "δ = Var" "t · δ = t"
    when "subst_domain δ = set []" for t and δ::"('fun, 'atom, 'sets) prot_subst"
    using that by auto
  moreover have "subst_domain Var = set []" "range_vars Var = {}"
    by simp_all
  ultimately show "?B ⟹ ?B'" unfolding range_vars_alt_def by metis
qed

lemma wellformed_transaction_fv_in_receives_or_selects:
  assumes T: "wellformed_transaction T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "x ∈ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T)"
proof -
  have "x ∈ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∪
            fv_lsst (transaction_checks T) ∪ fv_lsst (transaction_updates T) ∪
            fv_lsst (transaction_send T)"
    using x(1) fv_sst_append unlabel_append
    by (metis transaction_strand_def append_assoc)
  thus ?thesis using T x(2) unfolding wellformed_transaction_def by blast
qed

lemma dual_transaction_ik_is_transaction_send'':
  fixes δ I::"('a,'b,'c) prot_subst"
  assumes "wellformed_transaction T"
  shows "(ik_sst (unlabel (dual_lsst (transaction_strand T ·_lsst δ))) ·_set I) ·_αset a =
         (trms_sst (unlabel (transaction_send T)) ·_set δ ·_set I) ·_αset a" (is "?A = ?B")
using dual_transaction_ik_is_transaction_send[OF assms]
      subst_lsst_unlabel[of "dual_lsst (transaction_strand T)" δ]
      ik_sst_subst[of "unlabel (dual_lsst (transaction_strand T))" δ]
      dual_lsst_subst[of "transaction_strand T" δ]
by (auto simp add: abs_apply_terms_def)

lemma while_prot_terms_fun_mono:
  "mono (λM'. M ∪ ⋃(subterms ` M') ∪ ⋃((set ∘ fst ∘ Ana) ` M'))"
unfolding mono_def by fast
```

```
lemma while_prot_terms_SMP_overapprox:
  fixes M::"('fun,'atom,'sets) prot_terms"
  assumes N_supset: "M ∪ ⋃(subterms ` N) ∪ ⋃((set ∘ fst ∘ Ana) ` N) ⊆ N"
    and Value_vars_only: "∀x ∈ fv_set N. Γ_v x = TAtom Value"
  shows "SMP M ⊆ {a · δ | a δ. a ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ)}"
proof -
  define f where "f ≡ λM'. M ∪ ⋃(subterms ` M') ∪ ⋃((set ∘ fst ∘ Ana) ` M')"
  define S where "S ≡ {a · δ | a δ. a ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ)}"

  note 0 = Value_vars_only

  have "t ∈ S" when "t ∈ SMP M" for t
  using that
  proof (induction t rule: SMP.induct)
    case (MP t)
    hence "t ∈ N" "wt_subst Var" "wf_trms (subst_range Var)" using N_supset by auto
    hence "t · Var ∈ S" unfolding S_def by blast
    thus ?case by simp
  next
    case (Subterm t t')
    then obtain δ a where a: "a · δ = t" "a ∈ N" "wt_subst δ" "wf_trms (subst_range δ)"
      by (auto simp add: S_def)
    hence "∀x ∈ fv a. ∃τ. Γ (Var x) = TAtom τ" using 0 by auto
    hence *: "∀x ∈ fv a. (∃f. δ x = Fun f []) ∨ (∃y. δ x = Var y)"
      using a(3) TAtom_term_cases[OF wf_trm_subst_rangeD[OF a(4)]]
      by (metis wt_subst_def)
    obtain b where b: "b · δ = t'" "b ∈ subterms a"
      using subterms_subst_subterm[OF *, of t'] Subterm.hyps(2) a(1)
      by fast
    hence "b ∈ N" using N_supset a(2) by blast
    thus ?case using a b(1) unfolding S_def by blast
  next
    case (Substitution t ϑ)
    then obtain δ a where a: "a · δ = t" "a ∈ N" "wt_subst δ" "wf_trms (subst_range δ)"
      by (auto simp add: S_def)
    have "wt_subst (δ ∘_s ϑ)" "wf_trms (subst_range (δ ∘_s ϑ))"
      by (fact wt_subst_compose[OF a(3) Substitution.hyps(2)],
          fact wf_trms_subst_compose[OF a(4) Substitution.hyps(3)])
    moreover have "t · ϑ = a · δ ∘_s ϑ" using a(1) subst_subst_compose[of a δ ϑ] by simp
    ultimately show ?case using a(2) unfolding S_def by blast
  next
    case (Ana t K T k)
    then obtain δ a where a: "a · δ = t" "a ∈ N" "wt_subst δ" "wf_trms (subst_range δ)"
      by (auto simp add: S_def)
    obtain Ka Ta where a': "Ana a = (Ka,Ta)" by moura
    have *: "K = Ka ·_list δ"
    proof (cases a)
      case (Var x)
      then obtain g U where gU: "t = Fun g U"
        using a(1) Ana.hyps(2,3) Ana_var
        by (cases t) simp_all
      have "Γ (Var x) = TAtom Value" using Var a(2) 0 by auto
      hence "Γ (Fun g U) = TAtom Value"
        using a(1,3) Var gU wt_subst_trm''[OF a(3), of a]
        by argo
      thus ?thesis using gU Fun_Value_type_inv Ana.hyps(2,3) by fastforce
    next
      case (Fun g U) thus ?thesis using a(1) a' Ana.hyps(2) Ana_subst'[of g U] by simp
    qed
    then obtain ka where ka: "k = ka · δ" "ka ∈ set Ka" using Ana.hyps(3) by auto
    have "ka ∈ set ((fst ∘ Ana) a)" using ka(2) a' by simp
    hence "ka ∈ N" using a(2) N_supset by auto
    thus ?case using ka a(3,4) unfolding S_def by blast
```

```
  qed
  thus ?thesis unfolding S_def by blast
qed
```

2.3.5 The Protocol Transition System, Defined in Terms of the Reachable Constraints

```
definition transaction_fresh_subst where
  "transaction_fresh_subst σ T A ≡
    subst_domain σ = set (transaction_fresh T) ∧
    (∀t ∈ subst_range σ. ∃n. t = Fun (Val (n,False)) []) ∧
    (∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst A)) ∧
    (∀t ∈ subst_range σ. t ∉ subterms_set (trms_transaction T)) ∧
    inj_on σ (subst_domain σ)"


definition transaction_renaming_subst where
  "transaction_renaming_subst α P A ≡
    ∃n ≥ max_var_set (⋃(vars_transaction ` set P) ∪ vars_lsst A). α = var_rename n"

definition constraint_model where
  "constraint_model I A ≡
    constr_sem_stateful I (unlabel A) ∧
    interpretation_subst I ∧
    wf_trms (subst_range I)"

definition welltyped_constraint_model where
  "welltyped_constraint_model I A ≡  wt_subst I ∧ constraint_model I A"

lemma constraint_model_prefix:
  assumes "constraint_model I (A@B)"
  shows "constraint_model I A"
by (metis assms strand_sem_append_stateful unlabel_append constraint_model_def)

lemma welltyped_constraint_model_prefix:
  assumes "welltyped_constraint_model I (A@B)"
  shows "welltyped_constraint_model I A"
by (metis assms constraint_model_prefix welltyped_constraint_model_def)

lemma constraint_model_Val_is_Value_term:
  assumes "welltyped_constraint_model I A"
    and "t · I = Fun (Val n) []"
  shows "t = Fun (Val n) [] ∨ (∃m. t = Var (TAtom Value, m))"
proof -
  have "wt_subst I" using assms(1) unfolding welltyped_constraint_model_def by simp
  moreover have "Γ (Fun (Val n) []) = TAtom Value" by auto
  ultimately have *: "Γ t = TAtom Value" by (metis (no_types) assms(2) wt_subst_trm'')

  show ?thesis
  proof (cases t)
    case (Var x)
    obtain τ m where x: "x = (τ, m)" by (metis surj_pair)
    have "Γv x = TAtom Value" using * Var by auto
    hence "τ = TAtom Value" using x Γv_TAtom'[of Value τ m] by simp
    thus ?thesis using x Var by metis
  next
    case (Fun f T) thus ?thesis using assms(2) by auto
  qed
qed
```

The set of symbolic constraints reachable in any symbolic run of the protocol `P`.

σ instantiates the fresh variables of transaction `T` with fresh terms. α is a variable-renaming whose range consists of fresh variables.

```
inductive_set reachable_constraints::
```

```
  "('fun,'atom,'sets,'lbl) prot ⇒ ('fun,'atom,'sets,'lbl) prot_constr set"
  for P::"('fun,'atom,'sets,'lbl) prot"
where
  init:
  "[] ∈ reachable_constraints P"
| step:
  "⟦A ∈ reachable_constraints P;
    T ∈ set P;
    transaction_fresh_subst σ T A;
    transaction_renaming_subst α P A
   ⟧ ⟹ A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α) ∈ reachable_constraints P"
```

2.3.6 Admissible Transactions

```
definition admissible_transaction_checks where
  "admissible_transaction_checks T ≡
    ∀x ∈ set (unlabel (transaction_checks T)).
      is_Check x ∧
      (is_InSet x ⟶
          is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
          fst (the_Var (the_elem_term x)) = TAtom Value) ∧
      (is_NegChecks x ⟶
          bvars_sstp x = [] ∧
          ((the_eqs x = [] ∧ length (the_ins x) = 1) ∨
           (the_ins x = [] ∧ length (the_eqs x) = 1))) ∧
      (is_NegChecks x ∧ the_eqs x = [] ⟶ (let h = hd (the_ins x) in
          is_Var (fst h) ∧ is_Fun_Set (snd h) ∧
          fst (the_Var (fst h)) = TAtom Value))"

definition admissible_transaction_selects where
  "admissible_transaction_selects T ≡
    ∀x ∈ set (unlabel (transaction_selects T)).
      is_InSet x ∧ the_check x = Assign ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
      fst (the_Var (the_elem_term x)) = TAtom Value"

definition admissible_transaction_updates where
  "admissible_transaction_updates T ≡
    ∀x ∈ set (unlabel (transaction_updates T)).
      is_Update x ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
      fst (the_Var (the_elem_term x)) = TAtom Value"

definition admissible_transaction_terms where
  "admissible_transaction_terms T ≡
    wf_trms' arity (trms_lsst (transaction_strand T)) ∧
    (∀f ∈ ⋃(funs_term ‘ trms_transaction T).
      ¬is_Val f ∧ ¬is_Abs f ∧ ¬is_PubConstSetType f ∧ f ≠ Pair ∧
      ¬is_PubConstAttackType f ∧ ¬is_PubConstBottom f ∧ ¬is_PubConstOccursSecType f) ∧
    (∀r ∈ set (unlabel (transaction_strand T)).
      (∃f ∈ ⋃(funs_term ‘ (trms_sstp r)). is_Attack f) ⟶
        (let t = the_msg r in is_Send r ∧ is_Fun t ∧ is_Attack (the_Fun t) ∧ args t = []))"

definition admissible_transaction_occurs_checks where
  "admissible_transaction_occurs_checks T ≡ (
    (∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶
      receive⟨occurs (Var x)⟩ ∈ set (unlabel (transaction_receive T))) ∧
    (∀x ∈ set (transaction_fresh T). fst x = TAtom Value ⟶
      send⟨occurs (Var x)⟩ ∈ set (unlabel (transaction_send T))) ∧
    (∀r ∈ set (unlabel (transaction_receive T)). is_Receive r ⟶
      (OccursFact ∈ funs_term (the_msg r) ∨ OccursSec ∈ funs_term (the_msg r)) ⟶
        (∃x ∈ fv_transaction T - set (transaction_fresh T).
          fst x = TAtom Value ∧ the_msg r = occurs (Var x))) ∧
    (∀r ∈ set (unlabel (transaction_send T)). is_Send r ⟶
      (OccursFact ∈ funs_term (the_msg r) ∨ OccursSec ∈ funs_term (the_msg r)) ⟶
```

```
        (∃x ∈ set (transaction_fresh T).
          fst x = TAtom Value ∧ the_msg r = occurs (Var x)))
  )"

definition admissible_transaction where
  "admissible_transaction T ≡ (
    wellformed_transaction T ∧
    distinct (transaction_fresh T) ∧
    list_all (λx. fst x = TAtom Value) (transaction_fresh T) ∧
    (∀x ∈ vars_lsst (transaction_strand T). is_Var (fst x) ∧ (the_Var (fst x) = Value)) ∧
    bvars_lsst (transaction_strand T) = {} ∧
    (∀x ∈ fv_transaction T - set (transaction_fresh T).
     ∀y ∈ fv_transaction T - set (transaction_fresh T).
      x ≠ y ⟶ ⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
               ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))) ∧
    admissible_transaction_selects T ∧
    admissible_transaction_checks T ∧
    admissible_transaction_updates T ∧
    admissible_transaction_terms T ∧
    admissible_transaction_occurs_checks T
)"

lemma transaction_no_bvars:
  assumes "admissible_transaction T"
  shows "fv_transaction T = vars_transaction T"
    and "bvars_transaction T = {}"
proof -
  have "wellformed_transaction T" "bvars_lsst (transaction_strand T) = {}"
    using assms unfolding admissible_transaction_def
    by blast+
  thus "bvars_transaction T = {}" "fv_transaction T = vars_transaction T"
    using bvars_wellformed_transaction_unfold vars_sst_is_fv_sst_bvars_sst
    by fast+
qed

lemma transactions_fv_bvars_disj:
  assumes "∀T ∈ set P. admissible_transaction T"
  shows "(⋃T ∈ set P. fv_transaction T) ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
using assms transaction_no_bvars(2) by fast

lemma transaction_bvars_no_Value_type:
  assumes "admissible_transaction T"
    and "x ∈ bvars_transaction T"
  shows "¬TAtom Value ⊑ Γ_v x"
using assms transaction_no_bvars(2) by blast

lemma transaction_receive_deduct:
  assumes T_adm: "admissible_transaction T"
    and 𝓘: "constraint_model 𝓘 (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and t: "receive⟨t⟩ ∈ set (unlabel (transaction_receive T ·_lsst σ ∘_s α))"
  shows "ik_lsst A ·_set 𝓘 ⊢ t · 𝓘"
proof -
  define ϑ where "ϑ ≡ σ ∘_s α"

  have t': "send⟨t⟩ ∈ set (unlabel (dual_lsst (transaction_receive T ·_lsst ϑ)))"
    using t dual_lsst_unlabel_steps_iff(2) unfolding ϑ_def by blast
  then obtain T1 T2 where T: "unlabel (dual_lsst (transaction_receive T ·_lsst ϑ)) = T1@send⟨t⟩#T2"
    using t' by (meson split_list)

  have "constr_sem_stateful 𝓘 (unlabel A@unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
    using 𝓘 unlabel_append[of A] unfolding constraint_model_def ϑ_def by simp
```

```
  hence "constr_sem_stateful 𝓘 (unlabel A@T1@[send⟨t⟩])"
    using strand_sem_append_stateful[of "{}" "{}" "unlabel A@T1@[send⟨t⟩]" _ 𝓘]
          transaction_dual_subst_unfold[of T ϑ] T
    by (metis append.assoc append_Cons append_Nil)
  hence "ik_sst (unlabel A@T1) ·_set 𝓘 ⊢ t · 𝓘"
    using strand_sem_append_stateful[of "{}" "{}" "unlabel A@T1" "[send⟨t⟩]" 𝓘] T
    by force
  moreover have "¬is_Receive x"
    when x: "x ∈ set (unlabel (dual_lsst (transaction_receive T ·_lsst ϑ)))" for x
  proof -
    have *: "is_Receive a" when "a ∈ set (unlabel (transaction_receive T))" for a
      using T_adm Ball_set[of "unlabel (transaction_receive T)" is_Receive] that
      unfolding admissible_transaction_def wellformed_transaction_def
      by blast

    obtain l where l: "(l,x) ∈ set (dual_lsst (transaction_receive T ·_lsst ϑ))"
      using x unfolding unlabel_def by fastforce
    then obtain ly where ly: "ly ∈ set (transaction_receive T ·_lsst ϑ)" "(l,x) = dual_lsstp ly"
      unfolding dual_lsst_def by auto

    obtain j y where j: "ly = (j,y)" by (metis surj_pair)
    hence "j = l" using ly(2) by (cases y) auto
    hence y: "(l,y) ∈ set (transaction_receive T ·_lsst ϑ)" "(l,x) = dual_lsstp (l,y)"
      by (metis j ly(1), metis j ly(2))

    obtain z where z:
        "z ∈ set (unlabel (transaction_receive T))"
        "(l,z) ∈ set (transaction_receive T)"
        "(l,y) = (l,z) ·_lsstp ϑ"
      using y(1) unfolding subst_apply_labeled_stateful_strand_def unlabel_def by force

    have "is_Receive y" using *[OF z(1)] z(3) by (cases z) auto
    thus "¬is_Receive x" using l y by (cases y) auto
  qed
  hence "¬is_Receive x" when "x ∈ set T1" for x using T that by simp
  hence "ik_sst T1 = {}" unfolding ik_sst_def is_Receive_def by fast
  hence "ik_sst (unlabel A@T1) = ik_lsst A" using ik_sst_append[of "unlabel A" T1] by simp
  ultimately show ?thesis by (simp add: ϑ_def)
qed

lemma transaction_checks_db:
  assumes T: "admissible_transaction T"
    and 𝓘: "constraint_model 𝓘 (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
  shows "⟨Var (TAtom Value, n) in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
         ⟹ (α (TAtom Value, n) · 𝓘, Fun (Set s) []) ∈ set (db_lsst A 𝓘)"
      (is "?A ⟹ ?B")
    and "⟨Var (TAtom Value, n) not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
         ⟹ (α (TAtom Value, n) · 𝓘, Fun (Set s) []) ∉ set (db_lsst A 𝓘)"
      (is "?C ⟹ ?D")
proof -
  let ?x = "λn. (TAtom Value, n)"
  let ?s = "Fun (Set s) []"
  let ?T = "transaction_receive T@transaction_selects T@transaction_checks T"
  let ?T' = "?T ·_lsst σ ∘_s α"
  let ?S = "λS. transaction_receive T@transaction_selects T@S"
  let ?S' = "λS. ?S S ·_lsst σ ∘_s α"

  have T_valid: "wellformed_transaction T" using T by (simp add: admissible_transaction_def)

  have "constr_sem_stateful 𝓘 (unlabel (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)))"
    using 𝓘 unfolding constraint_model_def by simp
```

```
  moreover have
      "dual_lsst (transaction_strand T ·_lsst δ) =
       dual_lsst (?S (T1@[c]) ·_lsst δ)@
       dual_lsst (T2@transaction_updates T@transaction_send T ·_lsst δ)"
    when "transaction_checks T = T1@c#T2" for T1 T2 c δ
    using that dual_lsst_append subst_lsst_append
    unfolding transaction_strand_def
    by (metis append.assoc append_Cons append_Nil)
  ultimately have T'_model: "constr_sem_stateful I (unlabel (A@dual_lsst (?S' (T1@[(l,c)]))))"
    when "transaction_checks T = T1@(l,c)#T2" for T1 T2 l c
    using strand_sem_append_stateful[of _ _ _ _ I]
    by (simp add: that transaction_strand_def)

  show "?A ⟹ ?B"
  proof -
    assume a: ?A
    hence *: "⟨Var (?x n) in ?s⟩ ∈ set (unlabel ?T)"
      unfolding transaction_strand_def unlabel_def by simp
    then obtain l T1 T2 where T1: "transaction_checks T = T1@(l,⟨Var (?x n) in ?s⟩)#T2"
      by (metis a split_list unlabel_mem_has_label)

    have "?x n ∈ fv_lsst (transaction_checks T)"
      using a by force
    hence "?x n ∉ set (transaction_fresh T)"
      using a transaction_fresh_vars_notin[OF T_valid] by fast
    hence "unlabel (A@dual_lsst (?S' (T1@[(l,⟨Var (?x n) in ?s⟩)]))) =
          unlabel (A@dual_lsst (?S' T1))@[⟨α (?x n) in ?s⟩]"
      using T a σ dual_lsst_append subst_lsst_append unlabel_append
      by (fastforce simp add: transaction_fresh_subst_def unlabel_def dual_lsst_def
                              subst_apply_labeled_stateful_strand_def)
    moreover have "db_sst (unlabel A) = db_sst (unlabel (A@dual_lsst (?S' T1)))"
      by (simp add: T1 db_sst_transaction_prefix_eq[OF T_valid] del: unlabel_append)
    ultimately have "∃M. strand_sem_stateful M (set (db_sst (unlabel A) I)) [⟨α (?x n) in ?s⟩] I"
      using T'_model[OF T1] db_sst_set_is_dbupd_sst[of _ I] strand_sem_append_stateful[of _ _ _ _ I]
      by (simp add: db_sst_def del: unlabel_append)
    thus ?B by simp
  qed

  show "?C ⟹ ?D"
  proof -
    assume a: ?C
    hence *: "⟨Var (?x n) not in ?s⟩ ∈ set (unlabel ?T)"
      unfolding transaction_strand_def unlabel_def by simp
    then obtain l T1 T2 where T1: "transaction_checks T = T1@(l,⟨Var (?x n) not in ?s⟩)#T2"
      by (metis a split_list unlabel_mem_has_label)

    have "?x n ∈ vars_sstp ⟨Var (?x n) not in ?s⟩"
      using vars_sstp_cases(9)[of "[]" "Var (?x n)" ?s] by auto
    hence "?x n ∈ vars_lsst (transaction_checks T)"
      using a unfolding vars_sst_def by force
    hence "?x n ∉ set (transaction_fresh T)"
      using a transaction_fresh_vars_notin[OF T_valid] by fast
    hence "unlabel (A@dual_lsst (?S' (T1@[(l,⟨Var (?x n) not in ?s⟩)]))) =
          unlabel (A@dual_lsst (?S' T1))@[⟨α (?x n) not in ?s⟩]"
      using T a σ dual_lsst_append subst_lsst_append unlabel_append
      by (fastforce simp add: transaction_fresh_subst_def unlabel_def dual_lsst_def
                              subst_apply_labeled_stateful_strand_def)
    moreover have "db_sst (unlabel A) = db_sst (unlabel (A@dual_lsst (?S' T1)))"
      by (simp add: T1 db_sst_transaction_prefix_eq[OF T_valid] del: unlabel_append)
    ultimately have "∃M. strand_sem_stateful M (set (db_sst (unlabel A) I)) [⟨α (?x n) not in ?s⟩] I"
      using T'_model[OF T1] db_sst_set_is_dbupd_sst[of _ I] strand_sem_append_stateful[of _ _ _ _ I]
      by (simp add: db_sst_def del: unlabel_append)
    thus ?D using stateful_strand_sem_NegChecks_no_bvars(1)[of _ _ _ ?s I] by simp
```

```
  qed
qed

lemma transaction_selects_db:
  assumes T: "admissible_transaction T"
    and 𝓘: "constraint_model 𝓘 (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
  shows "select⟨Var (TAtom Value, n), Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))
         ⟹ (α (TAtom Value, n) · 𝓘, Fun (Set s) []) ∈ set (db_lsst A 𝓘)"
      (is "?A ⟹ ?B")
proof -
  let ?x = "λn. (TAtom Value, n)"
  let ?s = "Fun (Set s) []"
  let ?T = "transaction_receive T@transaction_selects T@transaction_checks T"
  let ?T' = "?T ·_lsst σ ∘_s α"
  let ?S = "λS. transaction_receive T@S"
  let ?S' = "λS. ?S S ·_lsst σ ∘_s α"

  have T_valid: "wellformed_transaction T" using T by (simp add: admissible_transaction_def)

  have "constr_sem_stateful 𝓘 (unlabel (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)))"
    using 𝓘 unfolding constraint_model_def by simp
  moreover have
      "dual_lsst (transaction_strand T ·_lsst δ) =
       dual_lsst (?S (T1@[c]) ·_lsst δ)@
       dual_lsst (T2@transaction_checks T @ transaction_updates T@transaction_send T ·_lsst δ)"
    when "transaction_selects T = T1@c#T2" for T1 T2 c δ
    using that dual_lsst_append subst_lsst_append
    unfolding transaction_strand_def by (metis append.assoc append_Cons append_Nil)
  ultimately have T'_model: "constr_sem_stateful 𝓘 (unlabel (A@dual_lsst (?S' (T1@[(l,c)]))))"
    when "transaction_selects T = T1@(l,c)#T2" for T1 T2 l c
    using strand_sem_append_stateful[of _ _ _ _ 𝓘]
    by (simp add: that transaction_strand_def)

  show "?A ⟹ ?B"
  proof -
    assume a: ?A
    hence *: "select⟨Var (?x n), ?s⟩ ∈ set (unlabel ?T)"
      unfolding transaction_strand_def unlabel_def by simp
    then obtain l T1 T2 where T1: "transaction_selects T = T1@(l,select⟨Var (?x n), ?s⟩)#T2"
      by (metis a split_list unlabel_mem_has_label)

    have "?x n ∈ fv_lsst (transaction_selects T)"
      using a by force
    hence "?x n ∉ set (transaction_fresh T)"
      using a transaction_fresh_vars_notin[OF T_valid] by fast
    hence "unlabel (A@dual_lsst (?S' (T1@[(l,select⟨Var (?x n), ?s⟩)]))) =
           unlabel (A@dual_lsst (?S' T1))@[select⟨α (?x n), ?s⟩]"
      using T a σ dual_lsst_append subst_lsst_append unlabel_append
      by (fastforce simp add: transaction_fresh_subst_def unlabel_def dual_lsst_def
                              subst_apply_labeled_stateful_strand_def)
    moreover have "db_sst (unlabel A) = db_sst (unlabel (A@dual_lsst (?S' T1)))"
      by (simp add: T1 db_sst_transaction_prefix_eq[OF T_valid] del: unlabel_append)
    ultimately have "∃M. strand_sem_stateful M (set (db_sst (unlabel A) 𝓘)) [⟨α (?x n) in ?s⟩] 𝓘"
      using T'_model[OF T1] db_sst_set_is_dbupd_sst[of _ 𝓘] strand_sem_append_stateful[of _ _ _ _ 𝓘]
      by (simp add: db_sst_def del: unlabel_append)
    thus ?B by simp
  qed
qed

lemma transactions_have_no_Value_consts:
  assumes "admissible_transaction T"
```

```
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a T. t = Fun (Val a) T" (is ?A)
    and "∄a T. t = Fun (Abs a) T" (is ?B)
proof -
  have "admissible_transaction_terms T" using assms(1) unfolding admissible_transaction_def by blast
  hence "¬is_Val f" "¬is_Abs f"
    when "f ∈ ⋃(funs_term ‘ (trms_transaction T))" for f
    using that unfolding admissible_transaction_terms_def by blast+
  moreover have "f ∈ ⋃(funs_term ‘ (trms_transaction T))"
    when "f ∈ funs_term t" for f
    using that assms(2) funs_term_subterms_eq(2)[of "trms_transaction T"] by blast+
  ultimately have *: "¬is_Val f" "¬is_Abs f"
    when "f ∈ funs_term t" for f
    using that by presburger+

  show ?A using *(1) by force
  show ?B using *(2) by force
qed

lemma transactions_have_no_Value_consts':
  assumes "admissible_transaction T"
    and "t ∈ trms_lsst (transaction_strand T)"
  shows "∄a T. Fun (Val a) T ∈ subterms t"
    and "∄a T. Fun (Abs a) T ∈ subterms t"
using transactions_have_no_Value_consts[OF assms(1)] assms(2) by fast+

lemma transactions_have_no_PubConsts:
  assumes "admissible_transaction T"
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a T. t = Fun (PubConstSetType a) T" (is ?A)
    and "∄a T. t = Fun (PubConstAttackType a) T" (is ?B)
    and "∄a T. t = Fun (PubConstBottom a) T" (is ?C)
    and "∄a T. t = Fun (PubConstOccursSecType a) T" (is ?D)
proof -
  have "admissible_transaction_terms T" using assms(1) unfolding admissible_transaction_def by blast
  hence "¬is_PubConstSetType f" "¬is_PubConstAttackType f"
        "¬is_PubConstBottom f" "¬is_PubConstOccursSecType f"
    when "f ∈ ⋃(funs_term ‘ (trms_transaction T))" for f
    using that unfolding admissible_transaction_terms_def by blast+
  moreover have "f ∈ ⋃(funs_term ‘ (trms_transaction T))"
    when "f ∈ funs_term t" for f
    using that assms(2) funs_term_subterms_eq(2)[of "trms_transaction T"] by blast+
  ultimately have *:
      "¬is_PubConstSetType f" "¬is_PubConstAttackType f"
      "¬is_PubConstBottom f" "¬is_PubConstOccursSecType f"
    when "f ∈ funs_term t" for f
    using that by presburger+

  show ?A using *(1) by force
  show ?B using *(2) by force
  show ?C using *(3) by force
  show ?D using *(4) by force
qed

lemma transactions_have_no_PubConsts':
  assumes "admissible_transaction T"
    and "t ∈ trms_lsst (transaction_strand T)"
  shows "∄a T. Fun (PubConstSetType a) T ∈ subterms t"
    and "∄a T. Fun (PubConstAttackType a) T ∈ subterms t"
    and "∄a T. Fun (PubConstBottom a) T ∈ subterms t"
    and "∄a T. Fun (PubConstOccursSecType a) T ∈ subterms t"
using transactions_have_no_PubConsts[OF assms(1)] assms(2) by fast+
```

```
lemma transaction_inserts_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_updates T"
    and "insert⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n)"
    and "∃u. s = Fun (Set u) []"
proof -
  let ?x = "insert⟨t,s⟩"

  have "?x ∈ set (unlabel (transaction_updates T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence *: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
          "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
          "is_Set (the_Fun (the_set_term ?x))"
    using assms(2) unfolding admissible_transaction_updates_def is_Fun_Set_def by fastforce+

  show "∃n. t = Var (TAtom Value, n)" using *(1,2) by (cases t) auto
  show "∃u. s = Fun (Set u) []" using *(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_deletes_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_updates T"
    and "delete⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n)"
    and "∃u. s = Fun (Set u) []"
proof -
  let ?x = "delete⟨t,s⟩"

  have "?x ∈ set (unlabel (transaction_updates T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence *: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
          "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
          "is_Set (the_Fun (the_set_term ?x))"
    using assms(2) unfolding admissible_transaction_updates_def is_Fun_Set_def by fastforce+

  show "∃n. t = Var (TAtom Value, n)" using *(1,2) by (cases t) auto
  show "∃u. s = Fun (Set u) []" using *(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_selects_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_selects T"
    and "select⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
proof -
  let ?x = "select⟨t,s⟩"

  have *: "?x ∈ set (unlabel (transaction_selects T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)

  have **: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
          "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
          "is_Set (the_Fun (the_set_term ?x))"
    using * assms(2) unfolding admissible_transaction_selects_def is_Fun_Set_def by fastforce+

  have "fv_sstp ?x ⊆ fv_lsst (transaction_selects T)"
    using * by force
  hence ***: "fv_sstp ?x ∩ set (transaction_fresh T) = {}"
```

```
    using T_valid unfolding wellformed_transaction_def by fast

  show ?A using **(1,2) *** by (cases t) auto
  show ?B using **(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_inset_checks_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_checks T"
    and "⟨t in s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
proof -
  let ?x = "⟨t in s⟩"

  have *: "?x ∈ set (unlabel (transaction_checks T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)

  have **: "is_Var (the_elem_term ?x)" "fst (the_Var (the_elem_term ?x)) = TAtom Value"
           "is_Fun (the_set_term ?x)" "args (the_set_term ?x) = []"
           "is_Set (the_Fun (the_set_term ?x))"
    using * assms(2) unfolding admissible_transaction_checks_def is_Fun_Set_def by fastforce+

  have "fv_sstp ?x ⊆ fv_lsst (transaction_checks T)"
    using * by force
  hence ***: "fv_sstp ?x ∩ set (transaction_fresh T) = {}"
    using T_valid unfolding wellformed_transaction_def by fast

  show ?A using **(1,2) *** by (cases t) auto
  show ?B using **(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma transaction_notinset_checks_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_checks T"
    and "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"
    and "(t,s) ∈ set G"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
proof -
  let ?x = "∀X⟨∨≠: F ∨∉: G⟩"

  have 0: "?x ∈ set (unlabel (transaction_checks T))"
    using assms(3) wellformed_transaction_unlabel_cases[OF T_valid, of ?x]
    by (auto simp add: transaction_strand_def unlabel_def)
  hence 1: "F = [] ∧ length G = 1"
    using assms(2,4) unfolding admissible_transaction_checks_def by fastforce
  hence "hd G = (t,s)" using assms(4) by (cases "the_ins ?x") auto
  hence **: "is_Var t" "fst (the_Var t) = TAtom Value" "is_Fun s" "args s = []" "is_Set (the_Fun s)"
    using 0 1 assms(2) unfolding admissible_transaction_checks_def Let_def is_Fun_Set_def
    by fastforce+

  have "fv_sstp ?x ⊆ fv_lsst (transaction_checks T)"
       "set (bvars_sstp ?x) ⊆ bvars_lsst (transaction_checks T)"
    using 0 by force+
  moreover have
      "fv_lsst (transaction_checks T) ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T)"
      "set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {}"
      "set (transaction_fresh T) ∩ fv_lsst (transaction_selects T) = {}"
    using T_valid unfolding wellformed_transaction_def by fast+
  ultimately have
      "fv_sstp ?x ∩ set (transaction_fresh T) = {}"
```

```
    "set (bvars_sstp ?x) ∩ set (transaction_fresh T) = {}"
  using wellformed_transaction_wf_sst(2,3)[OF T_valid]
        fv_transaction_unfold[of T] bvars_transaction_unfold[of T]
  by blast+
hence ***: "fv t ∩ set (transaction_fresh T) = {}"
  using assms(4) by auto

show ?A using **(1,2) *** by (cases t) auto
show ?B using **(3,4,5) unfolding is_Set_def by (cases s) auto
qed

lemma admissible_transaction_strand_step_cases:
assumes T_adm: "admissible_transaction T"
shows "r ∈ set (unlabel (transaction_receive T)) ⟹ ∃t. r = receive⟨t⟩"
      (is "?A ⟹ ?A'")
  and "r ∈ set (unlabel (transaction_selects T)) ⟹
         ∃x s. r = select⟨Var x, Fun (Set s) []⟩ ∧
               fst x = TAtom Value ∧ x ∈ fv_transaction T - set (transaction_fresh T)"
      (is "?B ⟹ ?B'")
  and "r ∈ set (unlabel (transaction_checks T)) ⟹
         (∃x s. (r = ⟨Var x in Fun (Set s) []⟩ ∨ r = ⟨Var x not in Fun (Set s) []⟩) ∧
                fst x = TAtom Value ∧ x ∈ fv_transaction T - set (transaction_fresh T)) ∨
         (∃s t. r = ⟨s == t⟩ ∨ r = ⟨s != t⟩)"
      (is "?C ⟹ ?C'")
  and "r ∈ set (unlabel (transaction_updates T)) ⟹
         ∃x s. (r = insert⟨Var x, Fun (Set s) []⟩ ∨ r = delete⟨Var x, Fun (Set s) []⟩) ∧
               fst x = TAtom Value"
      (is "?D ⟹ ?D'")
  and "r ∈ set (unlabel (transaction_send T)) ⟹ ∃t. r = send⟨t⟩"
      (is "?E ⟹ ?E'")
proof -
  have T_valid: "wellformed_transaction T"
    using T_adm unfolding admissible_transaction_def by metis

  show "?A ⟹ ?A'"
    using T_valid Ball_set[of "unlabel (transaction_receive T)" is_Receive]
    unfolding wellformed_transaction_def is_Receive_def
    by blast

  show "?E ⟹ ?E'"
    using T_valid Ball_set[of "unlabel (transaction_send T)" is_Send]
    unfolding wellformed_transaction_def is_Send_def
    by blast

  show "?B ⟹ ?B'"
  proof -
    assume r: ?B
    have "admissible_transaction_selects T"
      using T_adm unfolding admissible_transaction_def by simp
    hence *: "is_InSet r" "the_check r = Assign" "is_Var (the_elem_term r)"
             "is_Fun (the_set_term r)" "is_Set (the_Fun (the_set_term r))"
             "args (the_set_term r) = []" "fst (the_Var (the_elem_term r)) = TAtom Value"
      using r unfolding admissible_transaction_selects_def is_Fun_Set_def
      by fast+

    obtain rt rs where r': "r = select⟨rt,rs⟩" using *(1,2) by (cases r) auto
    obtain x where x: "rt = Var x" "fst x = TAtom Value" using *(3,7) r' by auto
    obtain f S where fS: "rs = Fun f S" using *(4) r' by auto
    obtain s where s: "f = Set s" using *(5) fS r' by (cases f) auto
    hence S: "S = []" using *(6) fS r' by (cases S) auto

    have fv_r1: "fv_sstp r ⊆ fv_transaction T"
      using r fv_transaction_unfold[of T] by auto
```

```
  have fv_r2: "fv_sstp r ∩ set (transaction_fresh T) = {}"
    using r T_valid unfolding wellformed_transaction_def by fastforce

  show ?B' using r' x fS s S fv_r1 fv_r2 by simp
qed

show "?C ⟹ ?C'"
proof -
  assume r: ?C
  have adm_checks: "admissible_transaction_checks T"
    using assms unfolding admissible_transaction_def by simp

  have fv_r1: "fv_sstp r ⊆ fv_transaction T"
    using r fv_transaction_unfold[of T] by auto

  have fv_r2: "fv_sstp r ∩ set (transaction_fresh T) = {}"
    using r T_valid unfolding wellformed_transaction_def by fastforce

  have "(is_InSet r ∧ the_check r = Check) ∨
        (is_Equality r ∧ the_check r = Check) ∨
        is_NegChecks r"
    using r adm_checks unfolding admissible_transaction_checks_def by fast
  thus ?C'
  proof (elim disjE conjE)
    assume *: "is_InSet r" "the_check r = Check"
    hence **: "is_Var (the_elem_term r)" "is_Fun (the_set_term r)"
              "is_Set (the_Fun (the_set_term r))" "args (the_set_term r) = []"
              "fst (the_Var (the_elem_term r)) = TAtom Value"
      using r adm_checks unfolding admissible_transaction_checks_def is_Fun_Set_def
      by fast+

    obtain rt rs where r': "r = ⟨rt in rs⟩" using * by (cases r) auto
    obtain x where x: "rt = Var x" "fst x = TAtom Value" using **(1,5) r' by auto
    obtain f S where fS: "rs = Fun f S" using **(2) r' by auto
    obtain s where s: "f = Set s" using **(3) fS r' by (cases f) auto
    hence S: "S = []" using **(4) fS r' by auto

    show ?C' using r' x fS s S fv_r1 fv_r2 by simp
  next
    assume *: "is_NegChecks r"
    hence **: "bvars_sstp r = []"
              "(the_eqs r = [] ∧ length (the_ins r) = 1) ∨
               (the_ins r = [] ∧ length (the_eqs r) = 1)"
      using r adm_checks unfolding admissible_transaction_checks_def by fast+
    show ?C' using **(2)
    proof (elim disjE conjE)
      assume ***: "the_eqs r = []" "length (the_ins r) = 1"
      then obtain t s where ts: "the_ins r = [(t,s)]" by (cases "the_ins r") auto
      hence "hd (the_ins r) = (t,s)" by simp
      hence ****: "is_Var (fst (t,s))" "is_Fun (snd (t,s))"
                  "is_Set (the_Fun (snd (t,s)))" "args (snd (t,s)) = []"
                  "fst (the_Var (fst (t,s))) = TAtom Value"
        using r adm_checks * ***(1) unfolding admissible_transaction_checks_def is_Fun_Set_def
        by metis+
      obtain x where x: "t = Var x" "fst x = TAtom Value" using ts ****(1,5) by (cases t) simp_all
      obtain f S where fS: "s = Fun f S" using ts ****(2) by (cases s) simp_all
      obtain ss where ss: "f = Set ss" using fS ****(3) by (cases f) simp_all
      have S: "S = []" using ts fS ss ****(4) by simp

      show ?C' using ts x fS ss S *** **(1) * fv_r1 fv_r2 by (cases r) auto
    next
      assume ***: "the_ins r = []" "length (the_eqs r) = 1"
```

```
      then obtain t s where "the_eqs r = [(t,s)]" by (cases "the_eqs r") auto
      thus ?C' using *** **(1) * by (cases r) auto
    qed
  qed (auto simp add: is_Equality_def the_check_def)
 qed

 show "?D ⟹ ?D'"
 proof -
  assume r: ?D
  have adm_upds: "admissible_transaction_updates T"
   using assms unfolding admissible_transaction_def by simp

  have *: "is_Update r" "is_Var (the_elem_term r)" "is_Fun (the_set_term r)"
          "is_Set (the_Fun (the_set_term r))" "args (the_set_term r) = []"
          "fst (the_Var (the_elem_term r)) = TAtom Value"
   using r adm_upds unfolding admissible_transaction_updates_def is_Fun_Set_def by fast+

  obtain t s where ts: "r = insert⟨t,s⟩ ∨ r = delete⟨t,s⟩" using *(1) by (cases r) auto
  obtain x where x: "t = Var x" "fst x = TAtom Value" using ts *(2,6) by (cases t) auto
  obtain f T where fT: "s = Fun f T" using ts *(3) by (cases s) auto
  obtain ss where ss: "f = Set ss" using ts fT *(4) by (cases f) fastforce+
  have T: "T = []" using ts fT *(5) ss by (cases T) auto

  show ?D'
   using ts x fT ss T by blast
 qed
qed

lemma transaction_Value_vars_are_fv:
 assumes "admissible_transaction T"
  and "x ∈ vars_transaction T"
  and "Γv x = TAtom Value"
 shows "x ∈ fv_transaction T"
using assms Γv_TAtom''(2)[of x] vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
unfolding admissible_transaction_def by fast

lemma protocol_transaction_vars_TAtom_typed:
 assumes P: "admissible_transaction T"
 shows "∀x ∈ vars_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
  and "∀x ∈ fv_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
  and "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
proof -
 have P': "wellformed_transaction T"
  using P unfolding admissible_transaction_def by fast

 show "∀x ∈ vars_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
  using P Γv_TAtom''
  unfolding admissible_transaction_def is_Var_def prot_atom.is_Atom_def the_Var_def
  by fastforce
 thus "∀x ∈ fv_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
  using vars_sst_is_fv_sst_bvars_sst by fast

 have "list_all (λx. fst x = Var Value) (transaction_fresh T)"
  using P Γv_TAtom'' unfolding admissible_transaction_def by fast
 thus "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  using Γv_TAtom''(2) unfolding list_all_iff by fast
qed

lemma protocol_transactions_no_pubconsts:
 assumes "admissible_transaction T"
 shows "Fun (Val (n,True)) S ∉ subterms_set (trms_transaction T)"
using assms transactions_have_no_Value_consts(1)
by fast
```

```
lemma protocol_transactions_no_abss:
  assumes "admissible_transaction T"
  shows "Fun (Abs n) S ∉ subterms_set (trms_transaction T)"
using assms transactions_have_no_Value_consts(2)
by fast

lemma admissible_transaction_strand_sem_fv_ineq:
  assumes T_adm: "admissible_transaction T"
    and 𝓘: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) 𝓘"
    and x: "x ∈ fv_transaction T - set (transaction_fresh T)"
    and y: "y ∈ fv_transaction T - set (transaction_fresh T)"
    and x_not_y: "x ≠ y"
  shows "ϑ x · 𝓘 ≠ ϑ y · 𝓘"
proof -
  have "⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
        ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))"
    using x y x_not_y T_adm unfolding admissible_transaction_def by auto
  hence "⟨Var x != Var y⟩ ∈ set (unlabel (transaction_strand T)) ∨
         ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_strand T))"
    unfolding transaction_strand_def unlabel_def by auto
  hence "⟨ϑ x != ϑ y⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ∨
         ⟨ϑ y != ϑ x⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
    using stateful_strand_step_subst_inI(8)[of _ _ "unlabel (transaction_strand T)" ϑ]
          subst_lsst_unlabel[of "transaction_strand T" ϑ]
          dual_lsst_unlabel_steps_iff(7)[of "[]" _ "[]"]
    by force
  then obtain B where B:
      "prefix (B@[⟨ϑ x != ϑ y⟩]) (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ∨
       prefix (B@[⟨ϑ y != ϑ x⟩]) (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
    unfolding prefix_def
    by (metis (no_types, hide_lams) append.assoc append_Cons append_Nil split_list)
  thus ?thesis
    using 𝓘 strand_sem_append_stateful[of IK DB _ _ 𝓘]
          stateful_strand_sem_NegChecks_no_bvars(2)
    unfolding prefix_def
    by metis
qed

lemma admissible_transactions_wf_trms:
  assumes "admissible_transaction T"
  shows "wf_trms (trms_transaction T)"
by (metis wf_trms_code assms admissible_transaction_def admissible_transaction_terms_def)

lemma admissible_transaction_no_Ana_Attack:
  assumes "admissible_transaction_terms T"
    and "t ∈ subterms_set (trms_transaction T)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
proof -
  obtain r where r: "r ∈ set (unlabel (transaction_strand T))" "t ∈ subterms_set (trms_sstp r)"
    using assms(2) by force

  obtain K M where t: "Ana t = (K, M)"
    by (metis surj_pair)

  show ?thesis
  proof
    assume n: "attack⟨n⟩ ∈ set (snd (Ana t))"
    hence "attack⟨n⟩ ∈ set M" using t by simp
    hence n': "attack⟨n⟩ ∈ subterms_set (trms_sstp r)"
      using Ana_subterm[OF t] r(2) subterms_subset by fast
    hence "∃f ∈ ⋃(funs_term ` trms_sstp r). is_Attack f"
      using funs_term_Fun_subterm' unfolding is_Attack_def by fast
```

```
    hence "is_Send r" "is_Fun (the_msg r)" "is_Attack (the_Fun (the_msg r))" "args (the_msg r) = []"
      using assms(1) r(1) unfolding admissible_transaction_terms_def by metis+
    hence "t = attack⟨n⟩"
      using n' r(2) unfolding is_Send_def is_Attack_def by auto
    thus False using n by fastforce
  qed
qed

lemma admissible_transaction_occurs_fv_types:
  assumes "admissible_transaction T"
    and "x ∈ vars_transaction T"
  shows "∃a. Γ (Var x) = TAtom a ∧ Γ (Var x) ≠ TAtom OccursSecType"
proof -
  have "is_Var (fst x)" "the_Var (fst x) = Value"
    using assms unfolding admissible_transaction_def by blast+
  thus ?thesis using Γv_TAtom''(2)[of x] by force
qed

lemma admissible_transaction_Value_vars:
  assumes T: "admissible_transaction T"
    and x: "x ∈ fv_transaction T"
  shows "Γv x = TAtom Value"
proof -
  have "x ∈ vars_transaction T"
    using x vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
    by blast
  hence "is_Var (fst x)" "the_Var (fst x) = Value"
    using T assms unfolding admissible_transaction_def list_all_iff by fast+
  thus "Γv x = TAtom Value" using Γv_TAtom''(2)[of x] by force
qed
```

2.3.7 Lemmata: Renaming and Fresh Substitutions

```
lemma transaction_renaming_subst_is_renaming:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "∃m. α (τ,n) = Var (τ,n+Suc m)"
using assms by (auto simp add: transaction_renaming_subst_def var_rename_def)

lemma transaction_renaming_subst_is_renaming':
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "∃y. α x = Var y"
using assms by (auto simp add: transaction_renaming_subst_def var_rename_def)

lemma transaction_renaming_subst_vars_disj:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "fvset (α ‘ (⋃(vars_transaction ‘ set P))) ∩ (⋃(vars_transaction ‘ set P)) = {}" (is ?A)
    and "fvset (α ‘ varslsst A) ∩ varslsst A = {}" (is ?B)
    and "T ∈ set P ⟹ vars_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C1")
    and "T ∈ set P ⟹ bvars_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C2")
    and "T ∈ set P ⟹ fv_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C3")
    and "varslsst A ∩ range_vars α = {}" (is ?D1)
    and "bvarslsst A ∩ range_vars α = {}" (is ?D2)
    and "fvlsst A ∩ range_vars α = {}" (is ?D3)
proof -
  define X where "X ≡ ⋃(vars_transaction ‘ set P) ∪ varslsst A"

  have 1: "finite X" by (simp add: X_def)

  obtain n where n: "n ≥ max_var_set X" "α = var_rename n"
    using assms unfolding transaction_renaming_subst_def X_def by moura
```

```
  hence 2: "∀x ∈ X. snd x < Suc n"
    using less_Suc_max_var_set[OF _ 1] unfolding var_rename_def by fastforce

  have 3: "x ∉ fv_set (α ‘ X)" "fv (α x) ∩ X = {}" "x ∉ range_vars α" when x: "x ∈ X" for x
    using 2 x n unfolding var_rename_def by force+

  show ?A ?B using 3(1,2) unfolding X_def by auto

  show ?C1 when T: "T ∈ set P" using T 3(3) unfolding X_def by blast
  thus ?C2 ?C3 when T: "T ∈ set P"
    using T by (simp_all add: disjoint_iff_not_equal vars_sst_is_fv_sst_bvars_sst)

  show ?D1 using 3(3) unfolding X_def by auto
  thus ?D2 ?D3 by (simp_all add: disjoint_iff_not_equal vars_sst_is_fv_sst_bvars_sst)
qed

lemma transaction_renaming_subst_wt:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "wt_subst α"
proof -
  { fix x::"('fun,'atom,'sets) prot_var"
    obtain τ n where x: "x = (τ,n)" by moura
    then obtain m where m: "α x = Var (τ,m)"
      using assms transaction_renaming_subst_is_renaming by moura
    hence "Γ (α x) = Γ_v x" using x by (simp add: Γ_v_def)
  } thus ?thesis by (simp add: wt_subst_def)
qed

lemma transaction_renaming_subst_is_wf_trm:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "wf_trm (α v)"
proof -
  obtain τ n where "v = (τ, n)" by moura
  then obtain m where "α v = Var (τ, n + Suc m)"
    using transaction_renaming_subst_is_renaming[OF assms]
    by moura
  thus ?thesis by (metis wf_trm_Var)
qed

lemma transaction_renaming_subst_range_wf_trms:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "wf_trms (subst_range α)"
by (metis transaction_renaming_subst_is_wf_trm[OF assms] wf_trm_subst_range_iff)

lemma transaction_renaming_subst_range_notin_vars:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P 𝒜"
  shows "∃y. α x = Var y ∧ y ∉ ⋃(vars_transaction ‘ set P) ∪ vars_lsst 𝒜"
proof -
  obtain τ n where x: "x = (τ,n)" by (metis surj_pair)

  define y where "y ≡ λm. (τ,n+Suc m)"

  have "∃m ≥ max_var_set (⋃(vars_transaction ‘ set P) ∪ vars_lsst 𝒜). α x = Var (y m)"
    using assms x by (auto simp add: y_def transaction_renaming_subst_def var_rename_def)
  moreover have "finite (⋃(vars_transaction ‘ set P) ∪ vars_lsst 𝒜)" by auto
  ultimately show ?thesis using x unfolding y_def by force
qed

lemma transaction_renaming_subst_var_obtain:
```

```
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes x: "x ∈ fv_sst (S ·_sst α)"
    and α: "transaction_renaming_subst α P 𝒜"
  shows "∃y. α y = Var x"
proof -
  obtain y where y: "y ∈ fv_sst S" "x ∈ fv (α y)" using fv_sst_subst_obtain_var[OF x] by moura
  thus ?thesis using transaction_renaming_subst_is_renaming'[OF α, of y] by fastforce
qed

lemma transaction_fresh_subst_is_wf_trm:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T A"
  shows "wf_trm (σ v)"
proof (cases "v ∈ subst_domain σ")
  case True
  then obtain n where "σ v = Fun (Val n) []"
    using assms unfolding transaction_fresh_subst_def
    by moura
  thus ?thesis by auto
qed auto

lemma transaction_fresh_subst_wt:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T A"
    and "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
  shows "wt_subst σ"
proof -
  have 1: "subst_domain σ = set (transaction_fresh T)"
      and 2: "∀t ∈ subst_range σ. ∃n. t = Fun (Val n) []"
    using assms(1) unfolding transaction_fresh_subst_def by metis+

  { fix x::"('fun,'atom,'sets) prot_var"
    have "Γ (Var x) = Γ (σ x)" using assms(2) 1 2 by (cases "x ∈ subst_domain σ") force+
  } thus ?thesis by (simp add: wt_subst_def)
qed

lemma transaction_fresh_subst_domain:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
  shows "subst_domain σ = set (transaction_fresh T)"
using assms unfolding transaction_fresh_subst_def by fast

lemma transaction_fresh_subst_range_wf_trms:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
  shows "wf_trms (subst_range σ)"
by (metis transaction_fresh_subst_is_wf_trm[OF assms] wf_trm_subst_range_iff)

lemma transaction_fresh_subst_range_fresh:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
  shows "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst 𝒜)"
    and "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst (transaction_strand T))"
using assms unfolding transaction_fresh_subst_def by meson+

lemma transaction_fresh_subst_sends_to_val:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
    and "y ∈ set (transaction_fresh T)"
  obtains n where "σ y = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
proof -
  have "σ y ∈ subst_range σ" using assms unfolding transaction_fresh_subst_def by simp
  thus ?thesis
```

```
    using assms that unfolding transaction_fresh_subst_def
    by fastforce
qed

lemma transaction_fresh_subst_sends_to_val':
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
    and "y ∈ set (transaction_fresh T)"
  obtains n where "(σ ∘ₛ α) y · ℐ = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
proof -
  obtain n where "σ y = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
    using transaction_fresh_subst_sends_to_val[OF assms] by moura
  thus ?thesis using that by (fastforce simp add: subst_compose_def)
qed

lemma transaction_fresh_subst_grounds_domain:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
    and "y ∈ set (transaction_fresh T)"
  shows "fv (σ y) = {}"
proof -
  obtain n where "σ y = Fun (Val n) []"
    using transaction_fresh_subst_sends_to_val[OF assms]
    by moura
  thus ?thesis by simp
qed

lemma transaction_fresh_subst_transaction_renaming_subst_range:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
  shows "x ∈ set (transaction_fresh T) ⟹ ∃n. (σ ∘ₛ α) x = Fun (Val (n,False)) []"
    and "x ∉ set (transaction_fresh T) ⟹ ∃y. (σ ∘ₛ α) x = Var y"
proof -
  assume "x ∈ set (transaction_fresh T)"
  then obtain n where "σ x = Fun (Val (n,False)) []"
    using assms(1) unfolding transaction_fresh_subst_def by fastforce
  thus "∃n. (σ ∘ₛ α) x = Fun (Val (n,False)) []" using subst_compose[of σ α x] by simp
next
  assume "x ∉ set (transaction_fresh T)"
  hence "σ x = Var x"
    using assms(1) unfolding transaction_fresh_subst_def by fastforce
  thus "∃y. (σ ∘ₛ α) x = Var y"
    using transaction_renaming_subst_is_renaming[OF assms(2)] subst_compose[of σ α x]
    by (cases x) force
qed

lemma transaction_fresh_subst_transaction_renaming_subst_range':
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
    and "t ∈ subst_range (σ ∘ₛ α)"
  shows "(∃n. t = Fun (Val (n,False)) []) ∨ (∃x. t = Var x)"
proof -
  obtain x where "x ∈ subst_domain (σ ∘ₛ α)" "(σ ∘ₛ α) x = t"
    using assms(3) by auto
  thus ?thesis
    using transaction_fresh_subst_transaction_renaming_subst_range[OF assms(1,2), of x]
    by auto
qed

lemma transaction_fresh_subst_transaction_renaming_subst_range'':
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes s: "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
    and y: "y ∈ fv ((σ ∘ₛ α) x)"
```

```
  shows "σ x = Var x"
    and "α x = Var y"
    and "(σ ∘s α) x = Var y"
proof -
  have "∃z. z ∈ fv (σ x)"
    using y subst_compose_fv'
    by fast
  hence x: "x ∉ subst_domain σ"
    using y transaction_fresh_subst_domain[OF s(1)]
          transaction_fresh_subst_grounds_domain[OF s(1), of x]
    by blast
  thus "σ x = Var x" by blast
  thus "α x = Var y" "(σ ∘s α) x = Var y"
    using y transaction_renaming_subst_is_renaming'[OF s(2), of x]
    unfolding subst_compose_def by fastforce+
qed

lemma transaction_fresh_subst_transaction_renaming_subst_vars_subset:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
  shows "⋃(fv_transaction ` set P) ⊆ subst_domain (σ ∘s α)" (is ?A)
    and "fv_lsst 𝒜 ⊆ subst_domain (σ ∘s α)" (is ?B)
    and "T' ∈ set P ⟹ fv_transaction T' ⊆ subst_domain (σ ∘s α)" (is "T' ∈ set P ⟹ ?C")
    and "T' ∈ set P ⟹ fv_lsst (transaction_strand T' ·lsst (σ ∘s α)) ⊆ range_vars (σ ∘s α)"
      (is "T' ∈ set P ⟹ ?D")
proof -
  have *: "x ∈ subst_domain (σ ∘s α)" for x
  proof (cases "x ∈ subst_domain σ")
    case True
    hence "x ∉ {x. ∃y. σ x = Var y ∧ α y = Var x}"
      using transaction_fresh_subst_domain[OF σ]
            transaction_fresh_subst_grounds_domain[OF σ, of x]
      by auto
    thus ?thesis using subst_domain_subst_compose[of σ α] by blast
  next
    case False
    hence "(σ ∘s α) x = α x" unfolding subst_compose_def by fastforce
    moreover have "α x ≠ Var x"
      using transaction_renaming_subst_is_renaming[OF α, of "fst x" "snd x"] by (cases x) auto
    ultimately show ?thesis by fastforce
  qed

  show ?A ?B using * by blast+

  show ?C when T: "T' ∈ set P" using T * by blast
  hence "fv_sst (unlabel (transaction_strand T') ·sst σ ∘s α) ⊆ range_vars (σ ∘s α)"
    when T: "T' ∈ set P"
    using T fv_sst_subst_subset_range_vars_if_subset_domain by blast
  thus ?D when T: "T' ∈ set P" by (metis T unlabel_subst)
qed

lemma transaction_fresh_subst_transaction_renaming_subst_vars_disj:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
  shows "fv_set ((σ ∘s α) ` (⋃(vars_transaction ` set P))) ∩ (⋃(vars_transaction ` set P)) = {}"
      (is ?A)
    and "x ∈ ⋃(vars_transaction ` set P) ⟹ fv ((σ ∘s α) x) ∩ (⋃(vars_transaction ` set P)) = {}"
      (is "?B' ⟹ ?B")
    and "T' ∈ set P ⟹ vars_transaction T' ∩ range_vars (σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C1")
    and "T' ∈ set P ⟹ bvars_transaction T' ∩ range_vars (σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C2")
    and "T' ∈ set P ⟹ fv_transaction T' ∩ range_vars (σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C3")
```

```
    and "vars_lsst 𝒜 ∩ range_vars (σ ∘s α) = {}" (is ?D1)
    and "bvars_lsst 𝒜 ∩ range_vars (σ ∘s α) = {}" (is ?D2)
    and "fv_lsst 𝒜 ∩ range_vars (σ ∘s α) = {}" (is ?D3)
proof -
  note 0 = transaction_renaming_subst_vars_disj[OF α]

  show ?A
  proof (cases "fv_set ((σ ∘s α) ‘ (⋃(vars_transaction ‘ set P))) = {}")
    case False
    hence "∀x ∈ (⋃(vars_transaction ‘ set P)). (σ ∘s α) x = α x ∨ fv ((σ ∘s α) x) = {}"
      using transaction_fresh_subst_transaction_renaming_subst_range''[OF σ α] by auto
    thus ?thesis using 0(1) by force
  qed blast
  thus "?B' ⟹ ?B" by auto

  have 1: "range_vars (σ ∘s α) ⊆ range_vars α"
    using range_vars_subst_compose_subset[of σ α]
          transaction_fresh_subst_domain[OF σ]
          transaction_fresh_subst_grounds_domain[OF σ]
    by force

  show ?C1 ?C2 ?C3 when T: "T' ∈ set P" using T 1 0(3,4,5)[of T'] by blast+

  show ?D1 ?D2 ?D3 using 1 0(6,7,8) by blast+
qed

lemma transaction_fresh_subst_transaction_renaming_subst_trms:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
    and "bvars_lsst S ∩ subst_domain σ = {}"
    and "bvars_lsst S ∩ subst_domain α = {}"
  shows "subterms_set (trms_lsst (S ·lsst (σ ∘s α))) = subterms_set (trms_lsst S) ·set (σ ∘s α)"
proof -
  have 1: "∀x ∈ fv_set (trms_lsst S). (∃f. (σ ∘s α) x = Fun f []) ∨ (∃y. (σ ∘s α) x = Var y)"
    using transaction_fresh_subst_transaction_renaming_subst_range[OF assms(1,2)] by blast

  have 2: "bvars_lsst S ∩ subst_domain (σ ∘s α) = {}"
    using assms(3,4) subst_domain_compose[of σ α] by blast

  show ?thesis using subterms_subst_lsst[OF 1 2] by simp
qed

lemma transaction_fresh_subst_transaction_renaming_wt:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
    and "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  shows "wt_subst (σ ∘s α)"
using transaction_renaming_subst_wt[OF assms(2)]
      transaction_fresh_subst_wt[OF assms(1,3)]
by (metis wt_subst_compose)

lemma transaction_fresh_subst_transaction_renaming_fv:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and x: "x ∈ fv_lsst (dual_lsst (transaction_strand T ·lsst σ ∘s α))"
  shows "∃y ∈ fv_transaction T - set (transaction_fresh T). (σ ∘s α) y = Var x"
proof -
  have "x ∈ fv_sst (unlabel (transaction_strand T) ·sst σ ∘s α)"
    using x fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "σ ∘s α"]
    by argo
  then obtain y where "y ∈ fv_transaction T" "x ∈ fv ((σ ∘s α) y)"
```

```
    by (metis fv_sst_subst_obtain_var)
  thus ?thesis
    using transaction_fresh_subst_transaction_renaming_subst_range[OF σ α, of y]
    by (cases "y ∈ set (transaction_fresh T)") force+
qed

lemma transaction_fresh_subst_transaction_renaming_subst_occurs_fact_send_receive:
  fixes t::"('fun,'atom,'sets) prot_term"
  assumes σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and T: "wellformed_transaction T"
  shows "send⟨occurs t⟩ ∈ set (unlabel (transaction_strand T ·lsst σ ∘s α))
         ⟹ ∃s. send⟨occurs s⟩ ∈ set (unlabel (transaction_send T)) ∧ t = s · σ ∘s α"
      (is "?A ⟹ ?A'")
    and "receive⟨occurs t⟩ ∈ set (unlabel (transaction_strand T ·lsst σ ∘s α))
         ⟹ ∃s. receive⟨occurs s⟩ ∈ set (unlabel (transaction_receive T)) ∧ t = s · σ ∘s α"
      (is "?B ⟹ ?B'")
proof -
  assume ?A
  then obtain s where s: "send⟨s⟩ ∈ set (unlabel (transaction_strand T))" "occurs t = s · σ ∘s α"
    using stateful_strand_step_subst_inv_cases(1)[
            of "occurs t" "unlabel (transaction_strand T)" "σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "σ ∘s α"]
    by auto

  note 0 = s(2) transaction_fresh_subst_transaction_renaming_subst_range[OF σ α]

  have "∃u. s = occurs u"
  proof (cases s)
    case (Var x)
    hence "(∃n. s · σ ∘s α = Fun (Val (n, False)) []) ∨ (∃y. s · σ ∘s α = Var y)"
      using 0(2,3)[of x] by (auto simp del: subst_subst_compose)
    thus ?thesis
      using 0(1) by simp
  next
    case (Fun f T)
    hence 1: "f = OccursFact" "length T = 2" "T ! 0 · σ ∘s α = Fun OccursSec []" "T ! 1 · σ ∘s α = t"
      using 0(1) by auto
    have "T ! 0 = Fun OccursSec []"
    proof (cases "T ! 0")
      case (Var x) thus ?thesis using 0(2,3)[of x] 1(3) by (auto simp del: subst_subst_compose)
    qed (use 1(3) in simp)
    thus ?thesis using Fun 1 0(1) by (auto simp del: subst_subst_compose)
  qed
  then obtain u where u: "s = occurs u" by moura
  hence "t = u · σ ∘s α" using s(2) by fastforce
  thus ?A' using s u wellformed_transaction_strand_unlabel_memberD(8)[OF T] by metis
next
  assume ?B
  then obtain s where s: "receive⟨s⟩ ∈ set (unlabel (transaction_strand T))" "occurs t = s · σ ∘s α"
    using stateful_strand_step_subst_inv_cases(2)[
            of "occurs t" "unlabel (transaction_strand T)" "σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "σ ∘s α"]
    by auto

  note 0 = s(2) transaction_fresh_subst_transaction_renaming_subst_range[OF σ α]

  have "∃u. s = occurs u"
  proof (cases s)
    case (Var x)
    hence "(∃n. s · σ ∘s α = Fun (Val (n, False)) []) ∨ (∃y. s · σ ∘s α = Var y)"
      using 0(2,3)[of x] by (auto simp del: subst_subst_compose)
    thus ?thesis
```

```
      using 0(1) by simp
  next
    case (Fun f T)
    hence 1: "f = OccursFact" "length T = 2" "T ! 0 · σ ∘s α = Fun OccursSec []" "T ! 1 · σ ∘s α = t"
      using 0(1) by auto
    have "T ! 0 = Fun OccursSec []"
    proof (cases "T ! 0")
      case (Var x) thus ?thesis using 0(2,3)[of x] 1(3) by (auto simp del: subst_subst_compose)
    qed (use 1(3) in simp)
    thus ?thesis using Fun 1 0(1) by (auto simp del: subst_subst_compose)
  qed
  then obtain u where u: "s = occurs u" by moura
  hence "t = u · σ ∘s α" using s(2) by fastforce
  thus ?B' using s u wellformed_transaction_strand_unlabel_memberD(1)[OF T] by metis
qed

lemma transaction_fresh_subst_proj:
  assumes "transaction_fresh_subst σ T A"
  shows "transaction_fresh_subst σ (transaction_proj n T) (proj n A)"
using assms transaction_proj_fresh_eq[of n T]
      contra_subsetD[OF subterms_set_mono[OF transaction_proj_trms_subset[of n T]]]
      contra_subsetD[OF subterms_set_mono[OF trms_sst_proj_subset(1)[of n A]]]
unfolding transaction_fresh_subst_def by metis

lemma transaction_renaming_subst_proj:
  assumes "transaction_renaming_subst α P A"
  shows "transaction_renaming_subst α (map (transaction_proj n) P) (proj n A)"
proof -
  let ?X = "λP A. ⋃(vars_transaction ‘ set P) ∪ vars_lsst A"
  define Y where "Y ≡ ?X (map (transaction_proj n) P) (proj n A)"
  define Z where "Z ≡ ?X P A"

  have "Y ⊆ Z"
    using sst_vars_proj_subset(3)[of n A] transaction_proj_vars_subset[of n]
    unfolding Y_def Z_def by fastforce
  hence "insert 0 (snd ‘ Y) ⊆ insert 0 (snd ‘ Z)" by blast
  moreover have "finite (insert 0 (snd ‘ Z))" "finite (insert 0 (snd ‘ Y))"
    unfolding Y_def Z_def by auto
  ultimately have 0: "max_var_set Y ≤ max_var_set Z" using Max_mono by blast

  have "∃n≥max_var_set Z. α = var_rename n"
    using assms unfolding transaction_renaming_subst_def Z_def by blast
  hence "∃n≥max_var_set Y. α = var_rename n" using 0 le_trans by fast
  thus ?thesis unfolding transaction_renaming_subst_def Y_def by blast
qed

lemma protocol_transaction_wf_subst:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes T: "wf'_sst (set (transaction_fresh T)) (unlabel (dual_lsst (transaction_strand T)))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
  shows "wf'_sst {} (unlabel (dual_lsst (transaction_strand T ·lsst σ ∘s α)))"
proof -
  have 0: "range_vars σ ∩ bvars_lsst (dual_lsst (transaction_strand T)) = {}"
          "ground (σ ‘ set (transaction_fresh T))" "ground (α ‘ {})"
    using transaction_fresh_subst_domain[OF σ] transaction_fresh_subst_grounds_domain[OF σ]
    by fastforce+

  have "wf'_sst {} ((unlabel (dual_lsst (transaction_strand T)) ·sst σ) ·sst α)"
    by (metis wf_sst_subst_apply[OF wf_sst_subst_apply[OF T]] 0(2,3))
  thus ?thesis
    by (metis dual_lsst_subst unlabel_subst labeled_stateful_strand_subst_comp[OF 0(1)])
qed
```

2.3.8 Lemmata: Reachable Constraints

```
lemma reachable_constraints_wf_trms:
  assumes "∀T ∈ set P. wf_trms (trms_transaction T)"
    and "A ∈ reachable_constraints P"
  shows "wf_trms (trms_lsst A)"
  using assms(2)
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  have "wf_trms (trms_transaction T)"
    using assms(1) step.hyps(2) by blast
  moreover have "wf_trms (subst_range (σ ∘s α))"
    using wf_trms_subst_compose[of σ α]
          transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
          transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
    by fastforce
  ultimately have "wf_trms (trms_transaction T ·set σ ∘s α)" by (metis wf_trms_subst)
  hence "wf_trms (trms_lsst (transaction_strand T ·lsst σ ∘s α))"
    using wf_trms_trms_sst_subst unlabel_subst[of "transaction_strand T" "σ ∘s α"] by metis
  hence "wf_trms (trms_lsst (dual_lsst (transaction_strand T ·lsst σ ∘s α)))"
    using trms_sst_unlabel_dual_lsst_eq by blast
  thus ?case using step.IH unlabel_append[of A] trms_sst_append[of "unlabel A"] by auto
qed simp

lemma reachable_constraints_TAtom_types:
  assumes "A ∈ reachable_constraints P"
    and "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  shows "Γv ‘ fv_lsst A ⊆ (⋃T ∈ set P. Γv ‘ fv_transaction T)" (is "?A A")
    and "Γv ‘ bvars_lsst A ⊆ (⋃T ∈ set P. Γv ‘ bvars_transaction T)" (is "?B A")
    and "Γv ‘ vars_lsst A ⊆ (⋃T ∈ set P. Γv ‘ vars_transaction T)" (is "?C A")
using assms(1)
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·lsst σ ∘s α)"

  have 2: "wt_subst (σ ∘s α)"
    using transaction_renaming_subst_wt[OF step.hyps(4)]
          transaction_fresh_subst_wt[OF step.hyps(3)]
    by (metis step.hyps(2) assms(2) wt_subst_compose)

  have 3: "∀t ∈ subst_range (σ ∘s α). fv t = {} ∨ (∃x. t = Var x)"
    using transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)]
    by fastforce

  have "fv_lsst T' = fv_lsst (transaction_strand T ·lsst σ ∘s α)"
       "bvars_lsst T' = bvars_lsst (transaction_strand T ·lsst σ ∘s α)"
       "vars_lsst T' = vars_lsst (transaction_strand T ·lsst σ ∘s α)"
    unfolding T'_def
    by (metis fv_sst_unlabel_dual_lsst_eq,
        metis bvars_sst_unlabel_dual_lsst_eq,
        metis vars_sst_unlabel_dual_lsst_eq)
  hence "Γ ‘ Var ‘ fv_lsst T' ⊆ Γ ‘ Var ‘ fv_transaction T"
        "Γ ‘ Var ‘ bvars_lsst T' = Γ ‘ Var ‘ bvars_transaction T"
        "Γ ‘ Var ‘ vars_lsst T' ⊆ Γ ‘ Var ‘ vars_transaction T"
    using wt_subst_lsst_vars_type_subset[OF 2 3, of "transaction_strand T"]
    by argo+
  hence "Γv ‘ fv_lsst T' ⊆ Γv ‘ fv_transaction T"
        "Γv ‘ bvars_lsst T' = Γv ‘ bvars_transaction T"
        "Γv ‘ vars_lsst T' ⊆ Γv ‘ vars_transaction T"
    by (metis Γv_Var_image)+
  hence 4: "Γv ‘ fv_lsst T' ⊆ (⋃T ∈ set P. Γv ‘ fv_transaction T)"
           "Γv ‘ bvars_lsst T' ⊆ (⋃T ∈ set P. Γv ‘ bvars_transaction T)"
           "Γv ‘ vars_lsst T' ⊆ (⋃T ∈ set P. Γv ‘ vars_transaction T)"
```

```
    using step.hyps(2) by fast+

  have 5: "Γv ‘ fvlsst (A @ T') = (Γv ‘ fvlsst A) ∪ (Γv ‘ fvlsst T')"
          "Γv ‘ bvarslsst (A @ T') = (Γv ‘ bvarslsst A) ∪ (Γv ‘ bvarslsst T')"
          "Γv ‘ varslsst (A @ T') = (Γv ‘ varslsst A) ∪ (Γv ‘ varslsst T')"
    using unlabel_append[of A T']
          fvsst_append[of "unlabel A" "unlabel T'"]
          bvarssst_append[of "unlabel A" "unlabel T'"]
          varssst_append[of "unlabel A" "unlabel T'"]
    by auto

  { case 1 thus ?case
      using step.IH(1) 4(1) 5(1)
      unfolding T'_def by (simp del: subst_subst_compose fvsst_def)
  }

  { case 2 thus ?case
      using step.IH(2) 4(2) 5(2)
      unfolding T'_def by (simp del: subst_subst_compose bvarssst_def)
  }

  { case 3 thus ?case
      using step.IH(3) 4(3) 5(3)
      unfolding T'_def by (simp del: subst_subst_compose)
  }
qed simp_all

lemma reachable_constraints_no_bvars:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. bvarslsst (transaction_strand T) = {}"
  shows "bvarslsst A = {}"
using assms proof (induction)
  case init
  then show ?case
    unfolding unlabel_def by auto
next
  case (step A T σ α)
  then have "bvarslsst A = {}"
    by metis
  moreover
  have "bvarslsst (duallsst (transaction_strand T ·lsst σ ∘s α)) = {}"
    using step by (metis bvarslsst_subst bvarssst_unlabel_duallsst_eq)
  ultimately
  show ?case
    using bvarssst_append unlabel_append by (metis sup_bot.left_neutral)
qed

lemma reachable_constraints_fv_bvars_disj:
  assumes A_reach: "A ∈ reachable_constraints P"
    and P: "∀S ∈ set P. admissible_transaction S"
  shows "fvlsst A ∩ bvarslsst A = {}"
proof -
  let ?X = "⋃T ∈ set P. bvars_transaction T"

  note 0 = transactions_fv_bvars_disj[OF P]

  have 1: "bvarslsst A ⊆ ?X" using A_reach
  proof (induction A rule: reachable_constraints.induct)
    case (step A T σ α)
    have "bvarslsst (duallsst (transaction_strand T ·lsst σ ∘s α)) = bvars_transaction T"
      using bvarssst_subst[of "unlabel (transaction_strand T)" "σ ∘s α"]
            bvarssst_unlabel_duallsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
            duallsst_subst[of "transaction_strand T" "σ ∘s α"]
```

```
        unlabel_subst[of "transaction_strand T" "σ ∘s α"]
    by argo
  hence "bvars_lsst (dual_lsst (transaction_strand T ·_lsst σ ∘s α)) ⊆ ?X"
    using step.hyps(2)
    by blast
  thus ?case
    using step.IH bvars_sst_append
    by auto
qed (simp add: unlabel_def bvars_sst_def)

have 2: "fv_lsst A ∩ ?X = {}" using A_reach
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  have "x ≠ y" when x: "x ∈ ?X" and y: "y ∈ fv_lsst (transaction_strand T ·_lsst σ ∘s α)" for x y
  proof -
    obtain y' where y': "y' ∈ fv_transaction T" "y ∈ fv ((σ ∘s α) y')"
      using y unlabel_subst[of "transaction_strand T" "σ ∘s α"]
      by (metis fv_sst_subst_obtain_var)

    have "y ∉ ⋃(vars_transaction ` set P)"
      using transaction_fresh_subst_transaction_renaming_subst_range''[OF step.hyps(3,4) y'(2)]
            transaction_renaming_subst_range_notin_vars[OF step.hyps(4), of y']
      by auto
    thus ?thesis using x vars_sst_is_fv_sst_bvars_sst by fast
  qed
  hence "fv_lsst (transaction_strand T ·_lsst σ ∘s α) ∩ ?X = {}"
    by blast
  thus ?case
    using step.IH
          fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·_lsst σ ∘s α"]
          dual_lsst_subst[of "transaction_strand T" "σ ∘s α"]
          unlabel_subst[of "transaction_strand T" "σ ∘s α"]
          fv_sst_append[of "unlabel A" "unlabel (transaction_strand T ·_lsst σ ∘s α)"]
          unlabel_append[of A "transaction_strand T"]
    by force
qed (simp add: unlabel_def fv_sst_def)

show ?thesis using 0 1 2 by blast
qed

lemma reachable_constraints_vars_TAtom_typed:
  assumes A_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "x ∈ vars_lsst A"
  shows "Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
proof -
  have A_wf_trms: "wf_trms (trms_lsst A)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P A_reach)

  have T_adm: "admissible_transaction T" when "T ∈ set P" for T
    by (meson that Ball_set P)

  have "∀T∈set P. ∀x∈set (transaction_fresh T). Γ_v x = TAtom Value"
    using protocol_transaction_vars_TAtom_typed(3) P by blast
  hence *: "Γ_v ` vars_lsst A ⊆ (⋃T∈set P. Γ_v ` vars_transaction T)"
    using reachable_constraints_TAtom_types[of A P, OF A_reach] by auto

  have "Γ_v ` vars_lsst A ⊆ TAtom ` insert Value (range Atom)"
  proof -
    have "Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
      when "T ∈ set P" "x ∈ vars_transaction T" for T x
      using that protocol_transaction_vars_TAtom_typed(1)[of T] P
      unfolding admissible_transaction_def
```

```
    by blast
  hence "(⋃T∈set P. Γv ‘ vars_transaction T) ⊆ TAtom ‘ insert Value (range Atom)"
    using P by blast
  thus "Γv ‘ varslsst A ⊆ TAtom ‘ insert Value (range Atom)"
    using * by auto
 qed
 thus ?thesis using x by auto
qed

lemma reachable_constraints_Value_vars_are_fv:
 assumes A_reach: "A ∈ reachable_constraints P"
  and P: "∀T ∈ set P. admissible_transaction T"
  and x: "x ∈ varslsst A"
  and "Γv x = TAtom Value"
 shows "x ∈ fvlsst A"
proof -
  have "∀T∈set P. bvars_transaction T = {}"
  using P unfolding list_all_iff admissible_transaction_def by metis
 hence A_no_bvars: "bvarslsst A = {}"
  using reachable_constraints_no_bvars[OF A_reach] by metis
 thus ?thesis using x varssst_is_fvsst_bvarssst[of "unlabel A"] by blast
qed

lemma reachable_constraints_subterms_subst:
 assumes A_reach: "A ∈ reachable_constraints P"
  and I: "welltyped_constraint_model I A"
  and P: "∀T ∈ set P. admissible_transaction T"
 shows "subtermsset (trmslsst (A ·lsst I)) = (subtermsset (trmslsst A)) ·set I"
proof -
 have A_wftrms: "wftrms (trmslsst A)"
  by (metis reachable_constraints_wftrms admissible_transactions_wftrms P A_reach)

 from I have I': "welltyped_constraint_model I A"
  using welltyped_constraint_model_prefix by auto

 have 1: "∀x ∈ fvset (trmslsst A). (∃f. I x = Fun f []) ∨ (∃y. I x = Var y)"
 proof
  fix x
  assume xa: "x ∈ fvset (trmslsst A)"
  have "∃f T. I x = Fun f T"
   using I interpretation_grounds[of I "Var x"]
   unfolding welltyped_constraint_model_def constraint_model_def
   by (cases "I x") auto
  then obtain f T where fT_p: "I x = Fun f T"
   by auto
  hence "wftrm (Fun f T)"
   using I
   unfolding welltyped_constraint_model_def constraint_model_def
   using wf_trm_subst_rangeD
   by metis
  moreover
  have "x ∈ varslsst A"
   using xa var_subterm_trmssst_is_varssst[of x "unlabel A"] vars_iff_subtermeq[of x]
   by auto
  hence "∃a. Γv x = TAtom a"
   using reachable_constraints_vars_TAtom_typed[OF A_reach P] by blast
  hence "∃a. Γ (Var x) = TAtom a"
   by simp
  hence "∃a. Γ (Fun f T) = TAtom a"
   by (metis (no_types, hide_lams) I' welltyped_constraint_model_def fT_p wtsubst_def)
  ultimately show "(∃f. I x = Fun f []) ∨ (∃y. I x = Var y)"
   using TAtom_term_cases fT_p by metis
 qed
```

```
  have "∀T∈set P. bvars_transaction T = {}"
    using assms unfolding list_all_iff admissible_transaction_def by metis
  then have "bvars_lsst 𝒜 = {}"
    using reachable_constraints_no_bvars assms by metis
  then have 2: "bvars_lsst 𝒜 ∩ subst_domain ℐ = {}"
    by auto

  show ?thesis
    using subterms_subst_lsst[OF _ 2] 1
    by simp
qed

lemma reachable_constraints_val_funs_private:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and f: "f ∈ ⋃(funs_term ‘ trms_lsst 𝒜)"
  shows "is_Val f ⟹ ¬public f"
    and "¬is_Abs f"
proof -
  have "(is_Val f ⟶ ¬public f) ∧ ¬is_Abs f" using 𝒜_reach f
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step 𝒜 T σ α)
    let ?T' = "unlabel (transaction_strand T) ·sst σ ∘s α"
    let ?T'' = "transaction_strand T ·lsst σ ∘s α"

    have T: "admissible_transaction_terms T"
      using P step.hyps(2) unfolding admissible_transaction_def by metis

    show ?thesis using step
    proof (cases "f ∈ ⋃(funs_term ‘ trms_lsst 𝒜)")
      case False
      then obtain t where t: "t ∈ trms_sst ?T'" "f ∈ funs_term t"
        using step.prems trms_sst_unlabel_dual_lsst_eq[of ?T'']
              trms_sst_append[of "unlabel 𝒜" "unlabel (dual_lsst ?T'')"]
              unlabel_append[of 𝒜 "dual_lsst ?T''"] unlabel_subst[of "transaction_strand T"]
        by fastforce
      show ?thesis using trms_sst_funs_term_cases[OF t]
      proof
        assume "∃u ∈ trms_transaction T. f ∈ funs_term u"
        thus ?thesis using T unfolding admissible_transaction_terms_def by blast
      next
        assume "∃x ∈ fv_transaction T. f ∈ funs_term ((σ ∘s α) x)"
        then obtain x where "x ∈ fv_transaction T" "f ∈ funs_term ((σ ∘s α) x)" by moura
        thus ?thesis
          using transaction_fresh_subst_transaction_renaming_subst_range[OF step.hyps(3,4), of x]
          by (force simp del: subst_subst_compose)
      qed
    qed simp
  qed simp
  thus "is_Val f ⟹ ¬public f" "¬is_Abs f" by simp_all
qed

lemma reachable_constraints_occurs_fact_ik_case:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and occ: "occurs t ∈ ik_lsst A"
  shows "∃n. t = Fun (Val (n,False)) []"
using 𝒜_reach occ
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  define ϑ where "ϑ ≡ σ ∘s α"
```

```
  have T: "wellformed_transaction T" "admissible_transaction_occurs_checks T"
    using P step.hyps(2) unfolding list_all_iff admissible_transaction_def by blast+

  show ?case
  proof (cases "occurs t ∈ ik_lsst A")
    case False
    hence "occurs t ∈ ik_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))"
      using step.prems unfolding ϑ_def by simp
    hence "receive⟨occurs t⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ)))"
      unfolding ik_sst_def by force
    hence "send⟨occurs t⟩ ∈ set (unlabel (transaction_strand T ·_lsst ϑ))"
      using dual_lsst_unlabel_steps_iff(1) by blast
    then obtain s where s:
        "send⟨s⟩ ∈ set (unlabel (transaction_strand T))" "s · ϑ = occurs t"
      by (metis (no_types) stateful_strand_step_subst_inv_cases(1) unlabel_subst)

    note 0 = transaction_fresh_subst_transaction_renaming_subst_range[OF step.hyps(3,4)]

    have 1: "send⟨s⟩ ∈ set (unlabel (transaction_send T))"
      using s(1) wellformed_transaction_strand_unlabel_memberD(8)[OF T(1)] by blast

    have 2: "is_Send (send⟨s⟩)"
      unfolding is_Send_def by simp

    have 3: "∃u. s = occurs u"
    proof -
      { fix z
        have "(∃n. ϑ z = Fun (Val (n, False)) []) ∨ (∃y. ϑ z = Var y)"
          using 0
          unfolding ϑ_def
          by blast
        hence "∄u. ϑ z = occurs u" "ϑ z ≠ Fun OccursSec []" by auto
      } note * = this

      obtain u u' where T: "s = Fun OccursFact [u,u']"
        using *(1) s(2) by (cases s) auto
      thus ?thesis using *(2) s(2) by (cases u) auto
    qed

    obtain x where x: "x ∈ set (transaction_fresh T)" "s = occurs (Var x)"
      using T(2) 1 2 3
      unfolding admissible_transaction_occurs_checks_def
      by fastforce

    have "t = ϑ x"
      using s(2) x(2) by auto
    thus ?thesis
      using 0(1)[OF x(1)] unfolding ϑ_def by fast
  qed (simp add: step.IH)
qed simp

lemma reachable_constraints_occurs_fact_send_ex:
  assumes A_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "Γ_v x = TAtom Value" "x ∈ fv_lsst A"

  shows "send⟨occurs (Var x)⟩ ∈ set (unlabel A)"
using A_reach x(2)
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  have T: "admissible_transaction_occurs_checks T"
    using P step.hyps(2) unfolding list_all_iff admissible_transaction_def by blast
```

```
  show ?case
  proof (cases "x ∈ fv_lsst A")
    case True
    show ?thesis
      using step.IH[OF True] unlabel_append[of A]
      by auto
  next
    case False
    then obtain y where y: "y ∈ fv_transaction T - set (transaction_fresh T)" "(σ ∘s α) y = Var x"
      using transaction_fresh_subst_transaction_renaming_fv[OF step.hyps(3,4), of x]
            step.prems(1) fv_sst_append[of "unlabel A"] unlabel_append[of A]
      by auto

    have "σ y = Var y" using y(1) step.hyps(3) unfolding transaction_fresh_subst_def by auto
    hence "α y = Var x" using y(2) unfolding subst_compose_def by simp
    hence y_val: "fst y = TAtom Value"
      using x(1) Γv_TAtom''[of x] Γv_TAtom''[of y]
            wt_subst_trm''[OF transaction_renaming_subst_wt[OF step.hyps(4)], of "Var y"]
      by force
    hence "receive⟨occurs (Var y)⟩ ∈ set (unlabel (transaction_receive T))"
      using y(1) T unfolding admissible_transaction_occurs_checks_def by fast
    hence *: "receive⟨occurs (Var y)⟩ ∈ set (unlabel (transaction_strand T))"
      using transaction_strand_subsets(6) by blast

    have "receive⟨occurs (Var x)⟩ ∈ set (unlabel (transaction_strand T ·lsst σ ∘s α))"
      using y(2) unlabel_subst[of "transaction_strand T" "σ ∘s α"]
            stateful_strand_step_subst_inI(2)[OF *, of "σ ∘s α"]
      by (auto simp del: subst_subst_compose)
    hence "send⟨occurs (Var x)⟩ ∈ set (unlabel (dual_lsst (transaction_strand T ·lsst σ ∘s α)))"
      using dual_lsst_unlabel_steps_iff(2) by blast
    thus ?thesis using unlabel_append[of A] by fastforce
  qed
qed simp

lemma reachable_constraints_db_lsst_set_args_empty:
  assumes A: "A ∈ reachable_constraints P"
    and PP: "list_all wellformed_transaction P"
    and admissible_transaction_updates:
      "let f = (λT. ∀x ∈ set (unlabel (transaction_updates T)).
                    is_Update x ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
                    fst (the_Var (the_elem_term x)) = TAtom Value)
      in list_all f P"
    and d: "(t, s) ∈ set (db_lsst A I)"
  shows "∃ss. s = Fun (Set ss) []"
  using A d
proof (induction)
  case (step A TT σ α)
  let ?TT = "transaction_strand TT ·lsst σ ∘s α"
  let ?TTu = "unlabel ?TT"
  let ?TTd = "dual_lsst ?TT"
  let ?TTdu = "unlabel ?TTd"
  from step(6) have "(t, s) ∈ set (db'_sst ?TTdu I (db'_sst (unlabel A) I []))"
    unfolding db_sst_def by (simp add: db_sst_append)
  hence "(t, s) ∈ set (db'_sst (unlabel A) I []) ∨
    (∃t' s'. insert⟨t',s'⟩ ∈ set ?TTdu ∧ t = t' · I ∧ s = s' · I)"
    using db_sst_in_cases[of t "s" ?TTdu I] by metis
  thus ?case
  proof
    assume "∃t' s'. insert⟨t',s'⟩ ∈ set ?TTdu ∧ t = t' · I ∧ s = s' · I"
    then obtain t' s' where t's'_p: "insert⟨t',s'⟩ ∈ set ?TTdu" "t = t' · I" "s = s' · I" by metis
    then obtain lll where "(lll, insert⟨t',s'⟩) ∈ set ?TTd" by (meson unlabel_mem_has_label)
    hence "(lll, insert⟨t',s'⟩) ∈ set (transaction_strand TT ·lsst σ ∘s α)"
      using dual_lsst_steps_iff(4) by blast
```

```
    hence "insert⟨t',s'⟩ ∈ set ?TTu" by (meson unlabel_in)
    hence "insert⟨t',s'⟩ ∈ set ((unlabel (transaction_strand TT)) ·sst σ ∘s α)"
      by (simp add: subst_lsst_unlabel)
    hence "insert⟨t',s'⟩ ∈ (λx. x ·sstp σ ∘s α) ‘ set (unlabel (transaction_strand TT))"
      unfolding subst_apply_stateful_strand_def by auto
    then obtain u where "u ∈ set (unlabel (transaction_strand TT)) ∧ u ·sstp σ ∘s α = insert⟨t',s'⟩"
      by auto
    hence "∃t'' s''. insert⟨t'',s''⟩ ∈ set (unlabel (transaction_strand TT)) ∧
                t' = t'' · σ ∘s α ∧ s' = s'' · σ ∘s α"
      by (cases u) auto
    then obtain t'' s'' where t''s''_p:
        "insert⟨t'',s''⟩ ∈ set (unlabel (transaction_strand TT)) ∧
          t' = t'' · σ ∘s α ∧ s' = s'' · σ ∘s α"
      by auto
    hence "insert⟨t'',s''⟩ ∈ set (unlabel (transaction_updates TT))"
      using is_Update_in_transaction_updates[of "insert⟨t'',s''⟩" TT]
      using PP step(2) unfolding list_all_iff by auto
    moreover have "∀x∈set (unlabel (transaction_updates TT)). is_Fun_Set (the_set_term x)"
      using step(2) admissible_transaction_updates unfolding is_Fun_Set_def list_all_iff by auto
    ultimately have "is_Fun_Set (the_set_term (insert⟨t'',s''⟩))" by auto
    moreover have "s' = s'' · σ ∘s α" using t''s''_p by blast
    ultimately have "is_Fun_Set (the_set_term (insert⟨t',s'⟩))" by (auto simp add: is_Fun_Set_subst)
    hence "is_Fun_Set s" by (simp add: t's'_p(3) is_Fun_Set_subst)
    thus ?case using is_Fun_Set_exi by auto
  qed (auto simp add: step dbsst_def)
qed auto

lemma reachable_constraints_occurs_fact_ik_ground:
  assumes A_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "occurs t ∈ iklsst A"
  shows "fv (occurs t) = {}"
proof -
  have 0: "admissible_transaction T"
    when "T ∈ set P" for T
    using P that unfolding list_all_iff by simp

  have 1: "wellformed_transaction T"
    when "T ∈ set P" for T
    using 0[OF that] unfolding admissible_transaction_def by simp

  have 2: "iklsst (A@duallsst (transaction_strand T ·lsst ϑ)) =
          (iklsst A) ∪ (trmslsst (transaction_send T) ·set ϑ)"
    when "T ∈ set P" for T ϑ and A::"('fun,'atom,'sets,'lbl) prot_constr"
    using dual_transaction_ik_is_transaction_send'[OF 1[OF that]] by fastforce

  have 3: "admissible_transaction_occurs_checks T"
    when "T ∈ set P" for T
    using 0[OF that] unfolding admissible_transaction_def by simp

  show ?thesis using A_reach t
  proof (induction A rule: reachable_constraints.induct)
    case (step A T σ α) thus ?case
    proof (cases "occurs t ∈ iklsst A")
      case False
      hence "occurs t ∈ trmslsst (transaction_send T) ·set σ ∘s α"
        using 2[OF step.hyps(2)] step.prems by blast
      hence "send⟨occurs t⟩ ∈ set (unlabel (transaction_send T ·lsst σ ∘s α))"
        using wellformed_transaction_send_receive_subst_trm_cases(2)[OF 1[OF step.hyps(2)]]
        by blast
      then obtain s where s:
          "send⟨occurs s⟩ ∈ set (unlabel (transaction_send T))" "t = s · σ ∘s α"
        using transaction_fresh_subst_transaction_renaming_subst_occurs_fact_send_receive(1)[
```

```
            OF step.hyps(3,4) 1[OF step.hyps(2)]]
          transaction_strand_subst_subsets(10)
    by blast

  obtain x where x: "x ∈ set (transaction_fresh T)" "s = Var x"
    using s(1) 3[OF step.hyps(2)]
    unfolding admissible_transaction_occurs_checks_def
    by fastforce

  have "fv t = {}"
    using transaction_fresh_subst_transaction_renaming_subst_range(1)[OF step.hyps(3,4) x(1)]
          s(2) x(2)
    by (auto simp del: subst_subst_compose)
  thus ?thesis by simp
 qed simp
 qed simp
qed

lemma reachable_constraints_occurs_fact_ik_funs_terms:
 fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
 assumes 𝒜_reach: "A ∈ reachable_constraints P"
   and ℐ: "welltyped_constraint_model I A"
   and P: "∀T ∈ set P. admissible_transaction T"
 shows "∀s ∈ subterms_set (ik_lsst A ·_set I). OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana s)))" (is "?A A")
   and "∀s ∈ subterms_set (ik_lsst A ·_set I). OccursSec ∉ ⋃(funs_term ‘ set (snd (Ana s)))" (is "?B A")
   and "Fun OccursSec [] ∉ ik_lsst A ·_set I" (is "?C A")
   and "∀x ∈ vars_lsst A. I x ≠ Fun OccursSec []" (is "?D A")
proof -
 have T_adm: "admissible_transaction T" when "T ∈ set P" for T
   using P that unfolding list_all_iff by simp

 have T_valid: "wellformed_transaction T" when "T ∈ set P" for T
   using T_adm[OF that] unfolding admissible_transaction_def by blast

 have T_occ: "admissible_transaction_occurs_checks T" when "T ∈ set P" for T
   using T_adm[OF that] unfolding admissible_transaction_def by blast

 have ℐ_wt: "wt_subst I" by (metis ℐ welltyped_constraint_model_def)

 have ℐ_wf_trms: "wf_trms (subst_range I)"
   by (metis ℐ welltyped_constraint_model_def constraint_model_def)

 have ℐ_grounds: "fv (I x) = {}" "∃f T. I x = Fun f T" for x
   using ℐ interpretation_grounds[of I, of "Var x"] empty_fv_exists_fun[of "I x"]
   unfolding welltyped_constraint_model_def constraint_model_def by auto

 have 00: "fv_set (trms_lsst (transaction_send T)) ⊆ vars_transaction T"
          "fv_set (subterms_set (trms_lsst (transaction_send T))) = fv_set (trms_lsst (transaction_send T))"
   for T::"('fun,'atom,'sets,'lbl) prot_transaction"
   using fv_trms_sst_subset(1)[of "unlabel (transaction_send T)"] vars_transaction_unfold
         fv_subterms_set[of "trms_lsst (transaction_send T)"]
   by blast+

 have 0: "∀x ∈ fv_set (trms_lsst (transaction_send T)). ∃a. Γ (Var x) = TAtom a"
         "∀x ∈ fv_set (trms_lsst (transaction_send T)). Γ (Var x) ≠ TAtom OccursSecType"
         "∀x ∈ fv_set (subterms_set (trms_lsst (transaction_send T))). ∃a. Γ (Var x) = TAtom a"
         "∀x ∈ fv_set (subterms_set (trms_lsst (transaction_send T))). Γ (Var x) ≠ TAtom OccursSecType"
         "∀x ∈ vars_transaction T. ∃a. Γ (Var x) = TAtom a"
         "∀x ∈ vars_transaction T. Γ (Var x) ≠ TAtom OccursSecType"
   when "T ∈ set P" for T
   using admissible_transaction_occurs_fv_types[OF T_adm[OF that]] 00
   by blast+
```

```
have 1: "ik_lsst (A@dual_lsst (transaction_strand T ·_lsst ϑ)) ·_set I =
         (ik_lsst A ·_set I) ∪ (trms_lsst (transaction_send T) ·_set ϑ ·_set I)"
  when "T ∈ set P" for T ϑ and A::"('fun,'atom,'sets,'lbl) prot_constr"
  using dual_transaction_ik_is_transaction_send'[OF T_valid[OF that]]
  by fastforce

have 2: "subterms_set (trms_lsst (transaction_send T) ·_set ϑ ·_set I) =
         subterms_set (trms_lsst (transaction_send T)) ·_set ϑ ·_set I"
  when "T ∈ set P" and ϑ: "wt_subst ϑ" "wf_trms (subst_range ϑ)" for T ϑ
  using wt_subst_TAtom_subterms_set_subst[OF wt_subst_compose[OF ϑ(1) I_wt] 0(1)[OF that(1)]]
        wf_trm_subst_rangeD[OF wf_trms_subst_compose[OF ϑ(2) I_wf_trms]]
  by auto

have 3: "wt_subst (σ ∘_s α)" "wf_trms (subst_range (σ ∘_s α))"
  when "T ∈ set P" "transaction_fresh_subst σ T A" "transaction_renaming_subst α P A"
  for σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
  using protocol_transaction_vars_TAtom_typed(3)[of T] P that(1)
        transaction_fresh_subst_transaction_renaming_wt[OF that(2,3)]
        transaction_fresh_subst_range_wf_trms[OF that(2)]
        transaction_renaming_subst_range_wf_trms[OF that(3)]
        wf_trms_subst_compose
  by simp_all

have 4: "∀s ∈ subterms_set (trms_lsst (transaction_send T)).
           OccursFact ∉ ⋃(funs_term ` set (snd (Ana s))) ∧
           OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))"
  when T: "T ∈ set P" for T
proof
  fix t assume t: "t ∈ subterms_set (trms_lsst (transaction_send T))"
  then obtain s where s: "send⟨s⟩ ∈ set (unlabel (transaction_send T))" "t ∈ subterms s"
    using wellformed_transaction_unlabel_cases(5)[OF T_valid[OF T]]
    by fastforce

  have s_occ: "∃x. s = occurs (Var x)" when "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t"
  proof -
    have "OccursFact ∈ funs_term s ∨ OccursSec ∈ funs_term s"
      using that subtermeq_imp_funs_term_subset[OF s(2)]
      by blast
    thus ?thesis
      using s T_occ[OF T]
      unfolding admissible_transaction_occurs_checks_def
      by fastforce
  qed

  obtain K T' where K: "Ana t = (K,T')" by moura

  show "OccursFact ∉ ⋃(funs_term ` set (snd (Ana t))) ∧
        OccursSec ∉ ⋃(funs_term ` set (snd (Ana t)))"
  proof (rule ccontr)
    assume "¬(OccursFact ∉ ⋃(funs_term ` set (snd (Ana t))) ∧
              OccursSec ∉ ⋃(funs_term ` set (snd (Ana t))))"
    hence a: "OccursFact ∈ ⋃(funs_term ` set (snd (Ana t))) ∨
              OccursSec ∈ ⋃(funs_term ` set (snd (Ana t)))"
      by simp
    hence "OccursFact ∈ ⋃(funs_term ` set T') ∨ OccursSec ∈ ⋃(funs_term ` set T')"
      using K by simp
    hence "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t"
      using Ana_subterm[OF K] funs_term_subterms_eq(1)[of t] by blast
    then obtain x where x: "t ∈ subterms (occurs (Var x))"
      using s(2) s_occ by blast
    thus False using a by fastforce
  qed
```

```
qed

have 5: "OccursFact ∉ ⋃(funs_term ‘ subst_range (σ ∘s α))"
        "OccursSec ∉ ⋃(funs_term ‘ subst_range (σ ∘s α))"
  when σα: "transaction_fresh_subst σ T A" "transaction_renaming_subst α P A"
  for σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  have "OccursFact ∉ funs_term t" "OccursSec ∉ funs_term t"
    when "t ∈ subst_range (σ ∘s α)" for t
    using transaction_fresh_subst_transaction_renaming_subst_range'[OF σα that]
    by auto
  thus "OccursFact ∉ ⋃(funs_term ‘ subst_range (σ ∘s α))"
       "OccursSec ∉ ⋃(funs_term ‘ subst_range (σ ∘s α))"
    by blast+
qed

have 6: "I x ≠ Fun OccursSec []" "∄t. I x = occurs t" "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
  when T: "T ∈ set P"
    and σα: "transaction_fresh_subst σ T A" "transaction_renaming_subst α P A"
    and x: "Var x ∈ trms_lsst (transaction_send T) ·set σ ∘s α"
  for x σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  obtain t where t: "t ∈ trms_lsst (transaction_send T)" "t · (σ ∘s α) = Var x"
    using x by moura
  then obtain y where y: "t = Var y" by (cases t) auto

  have "∃a. Γ t = TAtom a ∧ a ≠ OccursSecType"
    using 0(1,2)[OF T] t(1) y
    by force
  thus "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
    using wt_subst_trm''[OF 3(1)[OF T σα]] wt_subst_trm''[OF I_wt] t(2)
    by (metis subst_apply_term.simps(1))
  thus "I x ≠ Fun OccursSec []" "∄t. I x = occurs t"
    by auto
qed

have 7: "I x ≠ Fun OccursSec []" "∄t. I x = occurs t" "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
  when T: "T ∈ set P"
    and σα: "transaction_fresh_subst σ T A" "transaction_renaming_subst α P A"
    and x: "x ∈ fv_set ((σ ∘s α) ‘ vars_transaction T)"
  for x σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  obtain y where y: "y ∈ vars_transaction T" "x ∈ fv ((σ ∘s α) y)"
    using x by auto
  hence y': "(σ ∘s α) y = Var x"
    using transaction_fresh_subst_transaction_renaming_subst_range'[OF σα]
    by (cases "(σ ∘s α) y ∈ subst_range (σ ∘s α)") force+

  have "∃a. Γ (Var y) = TAtom a ∧ a ≠ OccursSecType"
    using 0(5,6)[OF T] y
    by force
  thus "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
    using wt_subst_trm''[OF 3(1)[OF T σα]] wt_subst_trm''[OF I_wt] y'
    by (metis subst_apply_term.simps(1))
  thus "I x ≠ Fun OccursSec []" "∄t. I x = occurs t"
    by auto
qed

have 8: "I x ≠ Fun OccursSec []" "∄t. I x = occurs t" "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
  when T: "T ∈ set P"
```

```
    and σα: "transaction_fresh_subst σ T A" "transaction_renaming_subst α P A"
    and x: "Var x ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α"
  for x σ α and T::"('fun,'atom,'sets,'lbl) prot_transaction"
    and A::"('fun,'atom,'sets,'lbl) prot_constr"
proof -
  obtain t where t: "t ∈ subterms_set (trms_lsst (transaction_send T))" "t · (σ ∘_s α) = Var x"
    using x by moura
  then obtain y where y: "t = Var y" by (cases t) auto

  have "∃a. Γ t = TAtom a ∧ a ≠ OccursSecType"
    using 0(3,4)[OF T] t(1) y
    by force
  thus "∃a. Γ (I x) = TAtom a ∧ a ≠ OccursSecType"
    using wt_subst_trm''[OF 3(1)[OF T σα]] wt_subst_trm''[OF I_wt] t(2)
    by (metis subst_apply_term.simps(1))
  thus "I x ≠ Fun OccursSec []" "∄t. I x = occurs t"
    by auto
qed

have s_fv: "fv s ⊆ fv_set ((σ ∘_s α) ‘ vars_transaction T)"
  when s: "s ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α"
    and T: "T ∈ set P"
  for s and σ α::"('fun,'atom,'sets) prot_subst" and T::"('fun,'atom,'sets,'lbl) prot_transaction"
proof
  fix x assume "x ∈ fv s"
  hence "x ∈ fv_set (subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α)"
    using s by auto
  hence *: "x ∈ fv_set (trms_lsst (transaction_send T) ·_set σ ∘_s α)"
    using fv_subterms_set_subst' by fast
  have **: "list_all is_Send (unlabel (transaction_send T))"
    using T_valid[OF T] unfolding wellformed_transaction_def by blast
  have "x ∈ fv_set ((σ ∘_s α) ‘ vars_lsst (transaction_send T))"
  proof -
    obtain t where t: "t ∈ trms_lsst (transaction_send T)" "x ∈ fv (t · σ ∘_s α)"
      using * by fastforce
    hence "fv t ⊆ vars_lsst (transaction_send T)"
      using fv_trms_sst_subset(1)[of "unlabel (transaction_send T)"]
      by auto
    thus ?thesis using t(2) subst_apply_fv_subset by fast
  qed
  thus "x ∈ fv_set ((σ ∘_s α) ‘ vars_transaction T)"
    using vars_transaction_unfold[of T] by fastforce
qed

show "?A A" using A_reach
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  have *: "∀s ∈ subterms_set (trms_lsst (transaction_send T)).
           OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
    using 4[OF step.hyps(2)] by blast

  have "∀s ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α ·_set I.
        OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
  proof
    fix t assume t: "t ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α ·_set I"
    then obtain s u where su:
        "s ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α" "s · I = t"
        "u ∈ subterms_set (trms_lsst (transaction_send T))" "u · σ ∘_s α = s"
      by force

    obtain Ku Tu where KTu: "Ana u = (Ku,Tu)" by moura

    have *: "OccursFact ∉ ⋃(funs_term ‘ set Tu)"
```

```
          "OccursFact ∉ ⋃(funs_term ‘ subst_range (σ ∘s α))"
          "OccursFact ∉ ⋃(funs_term ‘ ⋃(((set ∘ snd ∘ Ana) ‘ subst_range (σ ∘s α))))"
      using transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)]
          4[OF step.hyps(2)] su(3) KTu
      by fastforce+

    have "OccursFact ∉ ⋃(funs_term ‘ set (Tu ·list σ ∘s α))"
    proof -
      { fix f assume f: "f ∈ ⋃(funs_term ‘ set (Tu ·list σ ∘s α))"
        then obtain tf where tf: "tf ∈ set Tu" "f ∈ funs_term (tf · σ ∘s α)" by moura
        hence "f ∈ funs_term tf ∨ f ∈ ⋃(funs_term ‘ subst_range (σ ∘s α))"
          using funs_term_subst[of tf "σ ∘s α"] by force
        hence "f ≠ OccursFact" using *(1,2) tf(1) by blast
      } thus ?thesis by metis
    qed
    hence **: "OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
    proof (cases u)
      case (Var xu)
      hence "s = (σ ∘s α) xu" using su(4) by (metis subst_apply_term.simps(1))
      thus ?thesis using *(3) by fastforce
    qed (use su(4) KTu Ana_subst'[of _ _ Ku Tu "σ ∘s α"] in simp)

    show "OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana t)))"
    proof (cases s)
      case (Var sx)
      then obtain a where a: "Γ (I sx) = Var a"
        using su(1) 8(3)[OF step.hyps(2,3,4), of sx] by fast
      hence "Ana (I sx) = ([],[])" by (metis 𝓘_grounds(2) const_type_inv[THEN Ana_const])
      thus ?thesis using Var su(2) by simp
    next
      case (Fun f S)
      hence snd_Ana_t: "snd (Ana t) = snd (Ana s) ·list I"
        using su(2) Ana_subst'[of f S _ "snd (Ana s)" I] by (cases "Ana s") simp_all

      { fix g assume "g ∈ ⋃(funs_term ‘ set (snd (Ana t)))"
        hence "g ∈ ⋃(funs_term ‘ set (snd (Ana s))) ∨
              (∃x ∈ fv_set (set (snd (Ana s))). g ∈ funs_term (I x))"
          using snd_Ana_t funs_term_subst[of _ I] by auto
        hence "g ≠ OccursFact"
        proof
          assume "∃x ∈ fv_set (set (snd (Ana s))). g ∈ funs_term (I x)"
          then obtain x where x: "x ∈ fv_set (set (snd (Ana s)))" "g ∈ funs_term (I x)" by moura
          have "x ∈ fv s" using x(1) Ana_vars(2)[of s] by (cases "Ana s") auto
          hence "x ∈ fv_set ((σ ∘s α) ‘ vars_transaction T)"
            using s_fv[OF su(1) step.hyps(2)] by blast
          then obtain a h U where h:
              "I x = Fun h U" "Γ (I x) = Var a" "a ≠ OccursSecType" "arity h = 0"
            using 𝓘_grounds(2) 7(3)[OF step.hyps(2,3,4)] const_type_inv
            by metis
          hence "h ≠ OccursFact" by auto
          moreover have "U = []" using h(1,2,4) const_type_inv_wf[of h U a] 𝓘_wf_trms by fastforce
          ultimately show ?thesis using h(1) x(2) by auto
        qed (use ** in blast)
      } thus ?thesis by blast
    qed
  qed
  thus ?case
    using step.IH step.prems 1[OF step.hyps(2), of A "σ ∘s α"]
        2[OF step.hyps(2) 3[OF step.hyps(2,3,4)]]
    by auto
qed simp

show "?B A" using 𝒜_reach
```

```
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  have "∀s ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α ·_set I.
        OccursSec ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
  proof
    fix t assume t: "t ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α ·_set I"
    then obtain s u where su:
        "s ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α" "s · I = t"
        "u ∈ subterms_set (trms_lsst (transaction_send T))" "u · σ ∘_s α = s"
      by force

    obtain Ku Tu where KTu: "Ana u = (Ku,Tu)" by moura

    have *: "OccursSec ∉ ⋃(funs_term ‘ set Tu)"
            "OccursSec ∉ ⋃(funs_term ‘ subst_range (σ ∘_s α))"
            "OccursSec ∉ ⋃(funs_term ‘ ⋃(((set ∘ snd ∘ Ana) ‘ subst_range (σ ∘_s α))))"
      using transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)]
            4[OF step.hyps(2)] su(3) KTu
      by fastforce+

    have "OccursSec ∉ ⋃(funs_term ‘ set (Tu ·_list σ ∘_s α))"
    proof -
      { fix f assume f: "f ∈ ⋃(funs_term ‘ set (Tu ·_list σ ∘_s α))"
        then obtain tf where tf: "tf ∈ set Tu" "f ∈ funs_term (tf · σ ∘_s α)" by moura
        hence "f ∈ funs_term tf ∨ f ∈ ⋃(funs_term ‘ subst_range (σ ∘_s α))"
          using funs_term_subst[of tf "σ ∘_s α"] by force
        hence "f ≠ OccursSec" using *(1,2) tf(1) by blast
      } thus ?thesis by metis
    qed
    hence **: "OccursSec ∉ ⋃(funs_term ‘ set (snd (Ana s)))"
    proof (cases u)
      case (Var xu)
      hence "s = (σ ∘_s α) xu" using su(4) by (metis subst_apply_term.simps(1))
      thus ?thesis using *(3) by fastforce
    qed (use su(4) KTu Ana_subst'[of _ _ Ku Tu "σ ∘_s α"] in simp)

    show "OccursSec ∉ ⋃(funs_term ‘ set (snd (Ana t)))"
    proof (cases s)
      case (Var sx)
      then obtain a where a: "Γ (I sx) = Var a"
        using su(1) 8(3)[OF step.hyps(2,3,4), of sx] by fast
      hence "Ana (I sx) = ([],[])" by (metis ℐ_grounds(2) const_type_inv[THEN Ana_const])
      thus ?thesis using Var su(2) by simp
    next
      case (Fun f S)
      hence snd_Ana_t: "snd (Ana t) = snd (Ana s) ·_list I"
        using su(2) Ana_subst'[of f S _ "snd (Ana s)" I] by (cases "Ana s") simp_all

      { fix g assume "g ∈ ⋃(funs_term ‘ set (snd (Ana t)))"
        hence "g ∈ ⋃(funs_term ‘ set (snd (Ana s))) ∨
               (∃x ∈ fv_set (set (snd (Ana s))). g ∈ funs_term (I x))"
          using snd_Ana_t funs_term_subst[of _ I] by auto
        hence "g ≠ OccursSec"
        proof
          assume "∃x ∈ fv_set (set (snd (Ana s))). g ∈ funs_term (I x)"
          then obtain x where x: "x ∈ fv_set (set (snd (Ana s)))" "g ∈ funs_term (I x)" by moura
          have "x ∈ fv s" using x(1) Ana_vars(2)[of s] by (cases "Ana s") auto
          hence "x ∈ fv_set ((σ ∘_s α) ‘ vars_transaction T)"
            using s_fv[OF su(1) step.hyps(2)] by blast
          then obtain a h U where h:
              "I x = Fun h U" "Γ (I x) = Var a" "a ≠ OccursSecType" "arity h = 0"
            using ℐ_grounds(2) 7(3)[OF step.hyps(2,3,4)] const_type_inv
            by metis
```

```
          hence "h ≠ OccursSec" by auto
          moreover have "U = []" using h(1,2,4) const_type_inv_wf[of h U a] ℐ_wf_trms by fastforce
          ultimately show ?thesis using h(1) x(2) by auto
        qed (use ** in blast)
      } thus ?thesis by blast
    qed
  qed
  thus ?case
    using step.IH step.prems 1[OF step.hyps(2), of A "σ ∘s α"]
          2[OF step.hyps(2) 3[OF step.hyps(2,3,4)]]
    by auto
qed simp

show "?C A" using 𝒜_reach
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  have *: "Fun OccursSec [] ∉ trms_lsst (transaction_send T)"
    using wellformed_transaction_unlabel_cases(5)[OF T_valid[OF step.hyps(2)]]
          T_occ[OF step.hyps(2)]
    unfolding admissible_transaction_occurs_checks_def
    by fastforce

  have **: "Fun OccursSec [] ∉ subst_range (σ ∘s α)"
    using transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)]
    by auto

  have "Fun OccursSec [] ∉ trms_lsst (transaction_send T) ·set σ ∘s α ·set I"
  proof
    assume "Fun OccursSec [] ∈ trms_lsst (transaction_send T) ·set σ ∘s α ·set I"
    then obtain s where "s ∈ trms_lsst (transaction_send T) ·set σ ∘s α" "s · I = Fun OccursSec []"
      by moura
    moreover have "Fun OccursSec [] ∉ trms_lsst (transaction_send T) ·set σ ∘s α"
    proof
      assume "Fun OccursSec [] ∈ trms_lsst (transaction_send T) ·set σ ∘s α"
      then obtain u where "u ∈ trms_lsst (transaction_send T)" "u · σ ∘s α = Fun OccursSec []"
        by moura
      thus False using * ** by (cases u) (force simp del: subst_subst_compose)+
    qed
    ultimately show False using 6[OF step.hyps(2,3,4)] by (cases s) auto
  qed
  thus ?case using step.IH step.prems 1[OF step.hyps(2), of A "σ ∘s α"] by fast
qed simp

show "?D A" using 𝒜_reach
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  { fix x assume x: "x ∈ vars_lsst (dual_lsst (transaction_strand T ·lsst σ ∘s α))"
    hence x': "x ∈ vars_sst (unlabel (transaction_strand T) ·sst σ ∘s α)"
      by (metis vars_sst_unlabel_dual_lsst_eq unlabel_subst)
    hence "x ∈ vars_transaction T ∨ x ∈ fv_set ((σ ∘s α) ‘ vars_transaction T)"
      using vars_sst_subst_cases[OF x'] by metis
    moreover have "I x ≠ Fun OccursSec []" when "x ∈ vars_transaction T"
      using that 0(5,6)[OF step.hyps(2)] wt_subst_trm''[OF ℐ_wt, of "Var x"]
      by fastforce
    ultimately have "I x ≠ Fun OccursSec []"
      using 7(1)[OF step.hyps(2,3,4), of x]
      by blast
  } thus ?case using step.IH by auto
qed simp
qed

lemma reachable_constraints_occurs_fact_ik_subst_aux:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
```

```
    and 𝓘: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ ik_lsst A" "t · I = occurs s"
  shows "∃u. t = occurs u"
proof -
  have "wt_subst I"
    using 𝓘 unfolding welltyped_constraint_model_def constraint_model_def by metis
  hence 0: "Γ t = Γ (occurs s)"
    using t(2) wt_subst_trm'' by metis

  have 1: "Γ_v ‘ fv_lsst A ⊆ (⋃T ∈ set P. Γ_v ‘ fv_transaction T)"
          "∀T ∈ set P. ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    using reachable_constraints_TAtom_types(1)[OF 𝒜_reach]
          protocol_transaction_vars_TAtom_typed(2,3) P
    by fast+

  show ?thesis
  proof (cases t)
    case (Var x)
    thus ?thesis
      using 0 1 t(1) var_subterm_ik_sst_is_fv_sst[of x "unlabel A"]
      by fastforce
  next
    case (Fun f T)
    hence 2: "f = OccursFact" "length T = Suc (Suc 0)" "T ! 0 · I = Fun OccursSec []"
      using t(2) by auto

    have "T ! 0 = Fun OccursSec []"
    proof (cases "T ! 0")
      case (Var y)
      hence "I y = Fun OccursSec []" using Fun 2(3) by simp
      moreover have "Var y ∈ set T" using Var 2(2) length_Suc_conv[of T 1] by auto
      hence "y ∈ fv_set (ik_lsst A)" using Fun t(1) by force
      hence "y ∈ vars_lsst A"
        using fv_ik_subset_fv_sst'[of "unlabel A"] vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
        by blast
      ultimately have False
        using reachable_constraints_occurs_fact_ik_funs_terms(4)[OF 𝒜_reach 𝓘 P]
        by blast
      thus ?thesis by simp
    qed (use 2(3) in simp)
    moreover have "∃u u'. T = [u,u']"
      using 2(2) by (metis (no_types) length_0_conv length_Suc_conv)
    ultimately show ?thesis using Fun 2(1,2) by force
  qed
qed

lemma reachable_constraints_occurs_fact_ik_subst:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and 𝓘: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "occurs t ∈ ik_lsst A ·_set I"
  shows "occurs t ∈ ik_lsst A"
proof -
  have 𝓘_wt: "wt_subst I"
    using 𝓘 unfolding welltyped_constraint_model_def constraint_model_def by metis

  obtain s where s: "s ∈ ik_lsst A" "s · I = occurs t"
    using t by auto
  hence u: "∃u. s = occurs u"
    using 𝓘_wt reachable_constraints_occurs_fact_ik_subst_aux[OF 𝒜_reach 𝓘 P]
    by blast
  hence "fv s = {}"
```

```
    using reachable_constraints_occurs_fact_ik_ground[OF 𝒜_reach P] s
    by fast
  thus ?thesis
    using s u subst_ground_ident[of s I]
    by argo
qed

lemma reachable_constraints_occurs_fact_send_in_ik:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "send⟨occurs (Var x)⟩ ∈ set (unlabel A)"
  shows "occurs (I x) ∈ ik_lsst A"
using 𝒜_reach ℐ x
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  define ϑ where "ϑ ≡ σ ∘_s α"
  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst ϑ)"

  have T_adm: "admissible_transaction T"
    using P step.hyps(2) unfolding list_all_iff by blast

  have T_valid: "wellformed_transaction T"
    using T_adm unfolding admissible_transaction_def by blast

  have T_adm_occ: "admissible_transaction_occurs_checks T"
    using T_adm unfolding admissible_transaction_def by blast

  have ℐ_is_T_model: "strand_sem_stateful (ik_lsst A ·_set I) (set (db_lsst A I)) (unlabel T') I"
    using step.prems unlabel_append[of A T'] db_sst_set_is_dbupd_sst[of "unlabel A" I "[]"]
          strand_sem_append_stateful[of "{}" "{}" "unlabel A" "unlabel T'" I]
    by (simp add: T'_def ϑ_def welltyped_constraint_model_def constraint_model_def db_sst_def)

  show ?case
  proof (cases "send⟨occurs (Var x)⟩ ∈ set (unlabel A)")
    case False
    hence "send⟨occurs (Var x)⟩ ∈ set (unlabel T')"
      using step.prems(2) unfolding T'_def ϑ_def by simp
    hence "receive⟨occurs (Var x)⟩ ∈ set (unlabel (transaction_strand T ·_lsst ϑ))"
      using dual_lsst_unlabel_steps_iff(2) unfolding T'_def by blast
    then obtain y where y:
        "receive⟨occurs (Var y)⟩ ∈ set (unlabel (transaction_receive T))"
        "ϑ y = Var x"
      using transaction_fresh_subst_transaction_renaming_subst_occurs_fact_send_receive(2)[
              OF step.hyps(3,4) T_valid]
            subst_to_var_is_var[of _ ϑ x]
      unfolding ϑ_def by (force simp del: subst_subst_compose)
    hence "receive⟨occurs (Var y) · ϑ⟩ ∈ set (unlabel (transaction_receive T ·_lsst ϑ))"
      using subst_lsst_unlabel_member[of "receive⟨occurs (Var y)⟩" "transaction_receive T" ϑ]
      by fastforce
    hence "ik_lsst A ·_set I ⊢ occurs (Var y) · ϑ · I"
      using wellformed_transaction_sem_receives[
              OF T_valid, of "ik_lsst A ·_set I" "set (db_lsst A I)" ϑ I "occurs (Var y) · ϑ"]
            ℐ_is_T_model
      by (metis T'_def)
    hence *: "ik_lsst A ·_set I ⊢ occurs (ϑ y · I)"
      by auto

    have "occurs (ϑ y · I) ∈ ik_lsst A"
      using deduct_occurs_in_ik[OF *]
            reachable_constraints_occurs_fact_ik_subst[
              OF step.hyps(1) welltyped_constraint_model_prefix[OF step.prems(1)] P, of "ϑ y · I"]
            reachable_constraints_occurs_fact_ik_funs_terms[
```

```
            OF step.hyps(1) welltyped_constraint_model_prefix[OF step.prems(1)] P]
      by blast
    thus ?thesis using y(2) by simp
  qed (simp add: step.IH[OF welltyped_constraint_model_prefix[OF step.prems(1)]])
qed simp

lemma reachable_contraints_fv_bvars_subset:
  assumes A: "A ∈ reachable_constraints P"
  shows "bvars_lsst A ⊆ (⋃T ∈ set P. bvars_transaction T)"
using assms
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T σ α)
  let ?T' = "transaction_strand T ·_lsst σ ∘_s α"

  show ?case
    using step.IH step.hyps(2)
          bvars_sst_unlabel_dual_lsst_eq[of ?T']
          bvars_lsst_subst[of "transaction_strand T" "σ ∘_s α"]
          bvars_sst_append[of "unlabel 𝒜" "unlabel (dual_lsst ?T')"]
          unlabel_append[of 𝒜 "dual_lsst ?T'"]
    by (metis (no_types, lifting) SUP_upper Un_subset_iff)
qed simp

lemma reachable_contraints_fv_disj:
  assumes A: "A ∈ reachable_constraints P"
  shows "fv_lsst A ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T σ α)
  define T' where "T' ≡ transaction_strand T ·_lsst σ ∘_s α"
  define X where "X ≡ ⋃T ∈ set P. bvars_transaction T"
  have "fv_lsst T' ∩ X = {}"
    using transaction_fresh_subst_transaction_renaming_subst_vars_disj(4)[OF step.hyps(3,4)]
          transaction_fresh_subst_transaction_renaming_subst_vars_subset(4)[OF step.hyps(3,4,2)]
    unfolding T'_def X_def by blast
  hence "fv_lsst (𝒜@dual_lsst T') ∩ X = {}"
    using step.IH[unfolded X_def[symmetric]] fv_sst_unlabel_dual_lsst_eq[of T'] by auto
  thus ?case unfolding T'_def X_def by blast
qed simp

lemma reachable_contraints_fv_bvars_disj:
  assumes P: "∀T ∈ set P. wellformed_transaction T"
    and A: "A ∈ reachable_constraints P"
  shows "fv_lsst A ∩ bvars_lsst A = {}"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

  note 0 = transaction_fresh_subst_transaction_renaming_subst_vars_disj[OF step.hyps(3,4)]
  note 1 = transaction_fresh_subst_transaction_renaming_subst_vars_subset[OF step.hyps(3,4)]

  have 2: "bvars_lsst 𝒜 ∩ fv_lsst T' = {}"
    using 0(7) 1(4)[OF step.hyps(2)] fv_sst_unlabel_dual_lsst_eq
    unfolding T'_def by (metis (no_types) disjoint_iff_not_equal subset_iff)

  have "bvars_lsst T' ⊆ ⋃(bvars_transaction ` set P)"
       "fv_lsst 𝒜 ∩ ⋃(bvars_transaction ` set P) = {}"
    using reachable_contraints_fv_bvars_subset[OF reachable_constraints.step[OF step.hyps]]
          reachable_contraints_fv_disj[OF reachable_constraints.step[OF step.hyps]]
    unfolding T'_def by auto
  hence 3: "fv_lsst 𝒜 ∩ bvars_lsst T' = {}" by blast
```

```
  have "fv_lsst (transaction_strand T ·_lsst σ ∘_s α) ∩ bvars_transaction T = {}"
    using 0(4)[OF step.hyps(2)] 1(4)[OF step.hyps(2)] by blast
  hence 4: "fv_lsst T' ∩ bvars_lsst T' = {}"
    by (metis (no_types) T'_def fv_sst_unlabel_dual_lsst_eq bvars_sst_unlabel_dual_lsst_eq
              unlabel_subst bvars_sst_subst)

  have "fv_lsst (𝒜@T') ∩ bvars_lsst (𝒜@T') = {}"
    using 2 3 4 step.IH
    unfolding unlabel_append[of 𝒜 T']
              fv_sst_append[of "unlabel 𝒜" "unlabel T'"]
              bvars_sst_append[of "unlabel 𝒜" "unlabel T'"]
    by fast
  thus ?case unfolding T'_def by blast
qed simp

lemma reachable_constraints_wf:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    and A: "A ∈ reachable_constraints P"
  shows "wf_sst (unlabel A)"
    and "wf_trms (trms_lsst A)"
proof -
  have "wellformed_transaction T"
    when "T ∈ set P" for T
    using P(1) that by fast+
  hence 0: "wf'_sst (set (transaction_fresh T)) (unlabel (dual_lsst (transaction_strand T)))"
           "fv_lsst (dual_lsst (transaction_strand T)) ∩ bvars_lsst (dual_lsst (transaction_strand T)) = {}"
           "wf_trms (trms_transaction T)"
    when T: "T ∈ set P" for T
    unfolding admissible_transaction_terms_def
    by (metis T wellformed_transaction_wf_sst(1),
        metis T wellformed_transaction_wf_sst(2) fv_sst_unlabel_dual_lsst_eq bvars_sst_unlabel_dual_lsst_eq,
        metis T wf_trms_code P(2))

  from A have "wf_sst (unlabel A) ∧ wf_trms (trms_lsst A)"
  proof (induction A rule: reachable_constraints.induct)
    case (step A T σ α)
    let ?T' = "dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

    have IH: "wf'_sst {} (unlabel A)" "fv_lsst A ∩ bvars_lsst A = {}" "wf_trms (trms_lsst A)"
      using step.IH by metis+

    have 1: "wf'_sst {} (unlabel (A@?T'))"
      using protocol_transaction_wf_subst[OF 0(1)[OF step.hyps(2)] step.hyps(3,4)]
            wf_sst_vars_mono[of "{}"] wf_sst_append[OF IH(1)]
      by simp

    have 2: "fv_lsst (A@?T') ∩ bvars_lsst (A@?T') = {}"
      using reachable_contraints_fv_bvars_disj[OF P(1)]
            reachable_constraints.step[OF step.hyps]
      by blast

    have "wf_trms (trms_lsst ?T')"
      using trms_sst_unlabel_dual_lsst_eq unlabel_subst
            wf_trms_subst[
              OF wf_trms_subst_compose[
                OF transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
                   transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]],
              THEN wf_trms_trms_sst_subst,
              OF 0(3)[OF step.hyps(2)]]
      by metis
    hence 3: "wf_trms (trms_lsst (A@?T'))"
```

```
      using IH(3) by auto

    show ?case using 1 2 3 by force
  qed simp
  thus "wf_sst (unlabel A)" "wf_trms (trms_lsst A)" by metis+
qed

lemma reachable_constraints_no_Ana_Attack:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ subterms_set (ik_lsst 𝒜)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
proof -
  have T_adm: "admissible_transaction T" when "T ∈ set P" for T
    using P that by blast

  have T_adm_term: "admissible_transaction_terms T" when "T ∈ set P" for T
    using T_adm[OF that] unfolding admissible_transaction_def by blast

  have T_valid: "wellformed_transaction T" when "T ∈ set P" for T
    using T_adm[OF that] unfolding admissible_transaction_def by blast

  show ?thesis
  using 𝒜 t
  proof (induction 𝒜 rule: reachable_constraints.induct)
    case (step A T σ α) thus ?case
    proof (cases "t ∈ subterms_set (ik_lsst A)")
      case False
      hence "t ∈ subterms_set (ik_lsst (dual_lsst (transaction_strand T ·_lsst σ ∘_s α)))"
        using step.prems by simp
      hence "t ∈ subterms_set (trms_lsst (transaction_send T) ·_set σ ∘_s α)"
        using dual_transaction_ik_is_transaction_send'[OF T_valid[OF step.hyps(2)]]
        by metis
      hence "t ∈ subterms_set (trms_lsst (transaction_send T)) ·_set σ ∘_s α"
        using transaction_fresh_subst_transaction_renaming_subst_trms[
                OF step.hyps(3,4), of "transaction_send T"]
              wellformed_transaction_unlabel_cases(5)[OF T_valid[OF step.hyps(2)]]
        by fastforce
      then obtain s where s: "s ∈ subterms_set (trms_lsst (transaction_send T))" "t = s · σ ∘_s α"
        by moura
      hence s': "attack⟨n⟩ ∉ set (snd (Ana s))"
        using admissible_transaction_no_Ana_Attack[OF T_adm_term[OF step.hyps(2)]]
              trms_transaction_unfold[of T]
        by blast

      note * = transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)]

      show ?thesis
      proof
        assume n: "attack⟨n⟩ ∈ set (snd (Ana t))"
        thus False
        proof (cases s)
          case (Var x) thus ?thesis using Var * n s(2) by (force simp del: subst_subst_compose)
        next
          case (Fun f T)
          hence "attack⟨n⟩ ∈ set (snd (Ana s)) ·_set σ ∘_s α"
            using Ana_subst'[of f T _ "snd (Ana s)" "σ ∘_s α"] s(2) s' n
            by (cases "Ana s") auto
          hence "attack⟨n⟩ ∈ set (snd (Ana s)) ∨ attack⟨n⟩ ∈ subst_range (σ ∘_s α)"
            using const_mem_subst_cases' by fast
          thus ?thesis using * s' by blast
        qed
      qed
```

```
    qed simp
  qed simp
qed

lemma constraint_model_Value_term_is_Val:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "Γ_v x = TAtom Value" "x ∈ fv_lsst A"
  shows "∃n. I x = Fun (Val (n,False)) []"
using reachable_constraints_occurs_fact_send_ex[OF 𝒜_reach P x]
      reachable_constraints_occurs_fact_send_in_ik[OF 𝒜_reach ℐ P]
      reachable_constraints_occurs_fact_ik_case[OF 𝒜_reach P]
by fast

lemma constraint_model_Value_term_is_Val':
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "(TAtom Value, m) ∈ fv_lsst A"
  shows "∃n. I (TAtom Value, m) = Fun (Val (n,False)) []"
using constraint_model_Value_term_is_Val[OF 𝒜_reach ℐ P _ x] by simp


lemma constraint_model_Value_var_in_constr_prefix:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "∀x ∈ fv_lsst 𝒜. Γ_v x = TAtom Value
         ⟶ (∃B. prefix B 𝒜 ∧ x ∉ fv_lsst B ∧ ℐ x ∈ subterms_set (trms_lsst B))" (is "?P 𝒜")
using 𝒜_reach ℐ
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T σ α)
  have IH: "?P 𝒜" using step welltyped_constraint_model_prefix by fast

  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

  have T_adm: "admissible_transaction T"
    by (metis P step.hyps(2))

  have T_valid: "wellformed_transaction T"
    by (metis T_adm admissible_transaction_def)

  have ℐ_is_T_model: "strand_sem_stateful (ik_lsst 𝒜 ·_set ℐ) (set (db_lsst 𝒜 ℐ)) (unlabel T') ℐ"
    using step.prems unlabel_append[of 𝒜 T'] db_sst_set_is_dbupd_sst[of "unlabel 𝒜" ℐ "[]"]
          strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel T'" ℐ]
    by (simp add: T'_def welltyped_constraint_model_def constraint_model_def db_sst_def)

  have ℐ_interp: "interpretation_subst ℐ"
    and ℐ_wt: "wt_subst ℐ"
    and ℐ_wf_trms: "wf_trms (subst_range ℐ)"
    by (metis ℐ welltyped_constraint_model_def constraint_model_def,
        metis ℐ welltyped_constraint_model_def,
        metis ℐ welltyped_constraint_model_def constraint_model_def)

  have 1: "∃B. prefix B 𝒜 ∧ x ∉ fv_lsst B ∧ ℐ x ∈ subterms_set (trms_lsst B)"
    when x: "x ∈ fv_lsst T'" "Γ_v x = TAtom Value" for x
  proof -
    obtain n where n: "ℐ x = Fun n []" "is_Val n ∨ is_Abs n" "¬public n"
      using constraint_model_Value_term_is_Val[
              OF reachable_constraints.step[OF step.hyps] step.prems P x(2)]
            x(1) fv_sst_append[of "unlabel 𝒜" "unlabel T'"] unlabel_append[of 𝒜 T']
      unfolding T'_def by moura
```

```
have "x ∈ fv_lsst (transaction_strand T ·_lsst σ ∘_s α)"
  using x(1) fv_sst_unlabel_dual_lsst_eq unfolding T'_def by fastforce
then obtain y where y: "y ∈ fv_lsst (transaction_strand T)" "x ∈ fv ((σ ∘_s α) y)"
  using fv_sst_subst_obtain_var[of x "unlabel (transaction_strand T)" "σ ∘_s α"]
        unlabel_subst[of "transaction_strand T" "σ ∘_s α"]
  by auto

have y_x: "(σ ∘_s α) y = Var x"
  using y(2) transaction_fresh_subst_transaction_renaming_subst_range[OF step.hyps(3,4), of y]
  by force

have "Γ ((σ ∘_s α) y) = TAtom Value" using x(2) y_x by simp
moreover have "wt_subst (σ ∘_s α)"
  using protocol_transaction_vars_TAtom_typed(3) P(1) step.hyps(2)
        transaction_fresh_subst_transaction_renaming_wt[OF step.hyps(3,4)]
  by fast
ultimately have y_val: "Γ_v y = TAtom Value"
  by (metis wt_subst_def Γ.simps(1))

have y_not_fresh: "y ∉ set (transaction_fresh T)"
  using y(2) transaction_fresh_subst_transaction_renaming_subst_range(1)[OF step.hyps(3,4)]
  by fastforce

have y_n: "Fun n [] = (σ ∘_s α) y · I" using n y_x by simp
hence y_n': "Fun n [] = (σ ∘_s α ∘_s I) y"
  by (metis subst_subst_compose[of "Var y" "σ ∘_s α" I] subst_apply_term.simps(1))

have "y ∈ fv_lsst (transaction_receive T) ∨ y ∈ fv_lsst (transaction_selects T)"
  using wellformed_transaction_fv_in_receives_or_selects[OF T_valid] y(1) y_not_fresh by blast
hence n_cases:
  "Fun n [] ∈ subterms_set (trms_lsst A) ∨
   (∃z ∈ fv_lsst A. Γ_v z = TAtom Value ∧ I z = Fun n [])"
proof
  assume y_in: "y ∈ fv_lsst (transaction_receive T)"
  then obtain t where t: "receive⟨t⟩ ∈ set (unlabel (transaction_receive T))" "y ∈ fv t"
    using admissible_transaction_strand_step_cases(1)[OF T_adm]
    by force
  hence "receive⟨t · σ ∘_s α⟩ ∈ set (unlabel (transaction_receive T ·_lsst σ ∘_s α))"
    using subst_lsst_unlabel_member[of "receive⟨t⟩" "transaction_receive T" "σ ∘_s α"]
    by fastforce
  hence *: "ik_lsst A ·_set I ⊢ t · σ ∘_s α · I"
    using wellformed_transaction_sem_receives[
            OF T_valid, of "ik_lsst A ·_set I" "set (db_lsst A I)" "σ ∘_s α" I "t · σ ∘_s α"]
          I_is_T_model
    by (metis T'_def)

  have "∃a. Γ (I x) = Var a" when "x ∈ fv_lsst A" for x
    using that reachable_constraints_vars_TAtom_typed[OF step.hyps(1) P, of x]
          vars_sst_is_fv_sst_bvars_sst[of "unlabel A"] wt_subst_trm''[OF I_wt, of "Var x"]
    by force
  hence "∃f. I x = Fun f []" when "x ∈ fv_lsst A" for x
    using that wf_trm_subst[OF I_wf_trms, of "Var x"] wf_trm_Var[of x] const_type_inv_wf
          empty_fv_exists_fun[OF interpretation_grounds[OF I_interp], of "Var x"]
    by (metis subst_apply_term.simps(1)[of x I])
  hence A_ik_I_vals: "∀x ∈ fv_set (ik_lsst A). ∃f. I x = Fun f []"
    using fv_ik_subset_fv_sst'[of "unlabel A"] vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
    by blast
  hence "subterms_set (ik_lsst A ·_set I) = subterms_set (ik_lsst A) ·_set I"
    using ik_sst_subst[of "unlabel A" I] unlabel_subst[of A I]
          subterms_subst_lsst_ik[of A I]
    by metis
  moreover have "v ∈ fv_lsst A" when "v ∈ fv_set (ik_lsst A)" for v
```

```
    by (meson contra_subsetD fv_ik_subset_fv_sst' that)
  moreover have "Fun n [] ∈ subterms (t · σ ∘ₛ α · ℐ)"
    using imageI[of "Var y" "subterms t" "λx. x · σ ∘ₛ α ∘ₛ ℐ"]
          var_is_subterm[OF t(2)] subterms_subst_subset[of "σ ∘ₛ α ∘ₛ ℐ" t]
          subst_subst_compose[of t "σ ∘ₛ α" ℐ] y_n'
    by (auto simp del: subst_subst_compose)
  hence "Fun n [] ∈ subterms_set (ik_lsst 𝒜 ·_set ℐ)"
    using private_fun_deduct_in_ik[OF *, of n "[]"] n(2,3)
    unfolding is_Val_def is_Abs_def
    by auto
  hence "Fun n [] ∈ subterms_set (ik_lsst 𝒜) ∨
         (∃z ∈ fv_set (ik_lsst 𝒜). Fun n [] ∈ subterms (ℐ z))"
    using const_subterm_subst_cases[of n _ ℐ]
    by auto
  hence "Fun n [] ∈ subterms_set (ik_lsst 𝒜) ∨ (∃z ∈ fv_set (ik_lsst 𝒜). ℐ z = Fun n [])"
    using 𝒜_ik_ℐ_vals by fastforce
  hence "Fun n [] ∈ subterms_set (ik_lsst 𝒜) ∨
         (∃z ∈ fv_set (ik_lsst 𝒜). Γ_v z = TAtom Value ∧ ℐ z = Fun n [])"
    using ℐ_wt n(2) unfolding wt_subst_def is_Val_def is_Abs_def by force
  ultimately show ?thesis using ik_sst_trms_sst_subset[of "unlabel 𝒜"] by fast
next
  assume y_in: "y ∈ fv_lsst (transaction_selects T)"
  then obtain s where s: "select⟨Var y,Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))"
    using admissible_transaction_strand_step_cases(2)[OF T_adm]
    by force
  hence "select⟨(σ ∘ₛ α) y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T ·_lsst σ ∘ₛ α))"
    using subst_lsst_unlabel_member
    by fastforce
  hence n_in_db: "(Fun n [], Fun (Set s) []) ∈ set (db'_sst (unlabel 𝒜) ℐ [])"
    using wellformed_transaction_sem_selects[
            OF T_valid, of "ik_lsst 𝒜 ·_set ℐ" "set (db_lsst 𝒜 ℐ)" "σ ∘ₛ α" ℐ
                           "(σ ∘ₛ α) y" "Fun (Set s) []"]
          ℐ_is_T_model n y_x
    unfolding T'_def db_sst_def
    by fastforce

  obtain tn sn where tsn: "insert⟨tn,sn⟩ ∈ set (unlabel 𝒜)" "Fun n [] = tn · ℐ"
    using db_sst_in_cases[OF n_in_db] by force

  have "Fun n [] = tn ∨ (∃z. Γ_v z = TAtom Value ∧ tn = Var z)"
    using ℐ_wt tsn(2) n(2) unfolding wt_subst_def is_Val_def is_Abs_def by (cases tn) auto
  moreover have "tn ∈ subterms_set (trms_lsst 𝒜)" "fv tn ⊆ fv_lsst 𝒜"
    using tsn(1) in_subterms_Union by force+
  ultimately show ?thesis using tsn(2) by auto
qed

have x_nin_𝒜: "x ∉ fv_lsst 𝒜"
proof -
  have "x ∈ fv_lsst (transaction_strand T ·_lsst σ ∘ₛ α)"
    using x(1) fv_sst_unlabel_dual_lsst_eq
    unfolding T'_def
    by fast
  hence "x ∈ fv_sst ((unlabel (transaction_strand T) ·_sst σ) ·_sst α)"
    using transaction_fresh_subst_grounds_domain[OF step.hyps(3)] step.hyps(3)
          labeled_stateful_strand_subst_comp[of σ "transaction_strand T" α]
          unlabel_subst[of "transaction_strand T ·_lsst σ" α]
          unlabel_subst[of "transaction_strand T" σ]
    by (simp add: transaction_fresh_subst_def range_vars_alt_def)
  then obtain y where y: "α y = Var x"
    using transaction_renaming_subst_var_obtain[OF _ step.hyps(4)]
    by blast
  thus ?thesis
    using transaction_renaming_subst_range_notin_vars[OF step.hyps(4), of y]
```

```
            vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"]
        by auto
    qed

    from n_cases show ?thesis
    proof
      assume "∃z ∈ fv_lsst 𝒜. Γ_v z = TAtom Value ∧ ℐ z = Fun n []"
      then obtain B where B: "prefix B 𝒜" "Fun n [] ∈ subterms_set (trms_lsst B)"
        by (metis IH n(1))
      thus ?thesis
        using n x_nin_𝒜 trms_sst_unlabel_prefix_subset(1)[of B]
        by (metis (no_types, hide_lams) self_append_conv subset_iff subterms_set_mono prefix_def)
    qed (use n x_nin_𝒜 in fastforce)
  qed

  have "?P (𝒜@T')"
  proof (intro ballI impI)
    fix x assume x: "x ∈ fv_lsst (𝒜@T')" "Γ_v x = TAtom Value"
    show "∃B. prefix B (𝒜@T') ∧ x ∉ fv_lsst B ∧ ℐ x ∈ subterms_set (trms_lsst B)"
    proof (cases "x ∈ fv_lsst 𝒜")
      case False
      hence x': "x ∈ fv_lsst T'" using x(1) unlabel_append[of 𝒜] fv_sst_append[of "unlabel 𝒜"] by simp
      then obtain B where B: "prefix B 𝒜" "x ∉ fv_lsst B" "ℐ x ∈ subterms_set (trms_lsst B)"
        using x(2) 1 by moura
      thus ?thesis using prefix_prefix by fast
    qed (use x(2) IH prefix_prefix in fast)
  qed
  thus ?case unfolding T'_def by blast
qed simp

lemma admissible_transaction_occurs_checks_prop:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and f: "f ∈ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "is_Val f ⟹ ¬public f"
    and "¬is_Abs f"
proof -
  obtain x where x: "x ∈ fv_lsst 𝒜" "f ∈ funs_term (ℐ x)" using f by moura
  obtain T where T: "Fun f T ⊑ ℐ x" using funs_term_Fun_subterm[OF x(2)] by moura

  have ℐ_interp: "interpretation_subst ℐ"
    and ℐ_wt: "wt_subst ℐ"
    and ℐ_wf_trms: "wf_trms (subst_range ℐ)"
    by (metis ℐ welltyped_constraint_model_def constraint_model_def,
        metis ℐ welltyped_constraint_model_def,
        metis ℐ welltyped_constraint_model_def constraint_model_def)

  have 1: "Γ (Var x) = Γ (ℐ x)" using wt_subst_trm''[OF ℐ_wt, of "Var x"] by simp
  hence "∃a. Γ (ℐ x) = Var a"
    using x(1) reachable_constraints_vars_TAtom_typed[OF 𝒜_reach P, of x]
          vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"]
    by force
  hence "∃f. ℐ x = Fun f []"
    using x(1) wf_trm_subst[OF ℐ_wf_trms, of "Var x"] wf_trm_Var[of x] const_type_inv_wf
          empty_fv_exists_fun[OF interpretation_grounds[OF ℐ_interp], of "Var x"]
    by (metis subst_apply_term.simps(1)[of x ℐ])
  hence 2: "ℐ x = Fun f []" using x(2) by force

  have "(is_Val f ⟶ ¬public f) ∧ ¬is_Abs f"
  proof (cases "Γ_v x = TAtom Value")
    case True
    then obtain B where B: "prefix B 𝒜" "x ∉ fv_lsst B" "ℐ x ∈ subterms_set (trms_lsst B)"
```

```
    using constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach ℐ P] x(1)
    by fast

  have "ℐ x ∈ subterms_set (trms_lsst 𝒜)"
    using B(1,3) trms_sst_append[of "unlabel B"] unlabel_append[of B]
    unfolding prefix_def by auto
  hence "f ∈ ⋃(funs_term ‘ trms_lsst 𝒜)"
    using x(2) funs_term_subterms_eq(2)[of "trms_lsst 𝒜"] by blast
  thus ?thesis
    using reachable_constraints_val_funs_private[OF 𝒜_reach P]
    by blast+
 next
  case False thus ?thesis using x 1 2 by (cases f) auto
 qed
 thus "is_Val f ⟹ ¬public f" "¬is_Abs f" by metis+
qed

lemma admissible_transaction_occurs_checks_prop':
 assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
  and ℐ: "welltyped_constraint_model ℐ 𝒜"
  and P: "∀T ∈ set P. admissible_transaction T"
  and f: "f ∈ ⋃(funs_term ‘ (ℐ ‘ fv_lsst 𝒜))"
 shows "∄n. f = Val (n,True)"
  and "∄n. f = Abs n"
using admissible_transaction_occurs_checks_prop[OF 𝒜_reach ℐ P f] by auto

lemma transaction_var_becomes_Val:
 assumes 𝒜_reach: "𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α) ∈ reachable_constraints P"
  and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
  and σ: "transaction_fresh_subst σ T 𝒜"
  and α: "transaction_renaming_subst α P 𝒜"
  and P: "∀T ∈ set P. admissible_transaction T"
  and T: "T ∈ set P"
  and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
 shows "∃n. Fun (Val (n,False)) [] = (σ ∘_s α) x · ℐ"
proof -
 obtain m where m: "x = (TAtom Value, m)" by (metis x(2) eq_fst_iff)

 have x_not_bvar: "x ∉ bvars_transaction T" "fv ((σ ∘_s α) x) ∩ bvars_transaction T = {}"
  using x(1) transactions_fv_bvars_disj[OF P] T
      transaction_fresh_subst_transaction_renaming_subst_vars_disj(2)[OF σ α, of x]
      vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_strand T)"]
  by blast+

 show ?thesis
 proof (cases "x ∈ subst_domain σ")
  case True
  then obtain n where "σ x = Fun (Val (n, False)) []"
   using σ unfolding transaction_fresh_subst_def by fastforce
  thus ?thesis using subst_compose[of σ α x] by simp
 next
  case False
  hence "σ x = Var x" by auto
  then obtain n where n: "(σ ∘_s α) x = Var (TAtom Value, n)"
   using m transaction_renaming_subst_is_renaming[OF α] subst_compose[of σ α x]
   by force
  hence "(TAtom Value, n) ∈ fv_lsst (transaction_strand T ·_lsst σ ∘_s α)"
   using x_not_bvar fv_sst_subst_fv_subset[OF x(1), of "σ ∘_s α"]
       unlabel_subst[of "transaction_strand T" "σ ∘_s α"]
   by force
  hence "∃n'. ℐ (TAtom Value, n) = Fun (Val (n',False)) []"
   using constraint_model_Value_term_is_Val'[OF 𝒜_reach ℐ P, of n] x
       fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·_lsst σ ∘_s α"]
```

```
        fv_sst_append[of "unlabel A"] unlabel_append[of A]
    by fastforce
  thus ?thesis using n by simp
 qed
qed

lemma reachable_constraints_SMP_subset:
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
 shows "SMP (trms_lsst A) ⊆ SMP (⋃T ∈ set P. trms_transaction T)" (is "?A A")
   and "SMP (pair`setops_sst (unlabel A)) ⊆ SMP (⋃T∈set P. pair`setops_transaction T)" (is "?B A")
proof -
 have "?A A ∧ ?B A" using A
 proof (induction A rule: reachable_constraints.induct)
   case (step A T σ α)
   define T' where "T' ≡ transaction_strand T ·_lsst σ ∘_s α"
   define M where "M ≡ ⋃T ∈ set P. trms_transaction T"
   define N where "N ≡ ⋃T ∈ set P. pair ` setops_transaction T"

   let ?P = "λt. ∃s δ. s ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"
   let ?Q = "λt. ∃s δ. s ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ t = s · δ"

   have IH: "SMP (trms_lsst A) ⊆ SMP M" "SMP (pair ` setops_sst (unlabel A)) ⊆ SMP N"
     using step.IH by (metis M_def, metis N_def)

   have σα_wt: "wt_subst (σ ∘_s α)"
     using P(1) step.hyps(2)
         transaction_fresh_subst_transaction_renaming_wt[OF step.hyps(3,4)]
     by fast

   have σα_wf: "wf_trms (subst_range (σ ∘_s α))"
     using transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
         transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
     by (metis wf_trms_subst_compose)

   have 0: "SMP (trms_lsst (A@dual_lsst T')) = SMP (trms_lsst A) ∪ SMP (trms_lsst T')"
         "SMP (pair ` setops_sst (unlabel (A@dual_lsst T'))) =
           SMP (pair ` setops_sst (unlabel A)) ∪ SMP (pair ` setops_sst (unlabel T'))"
     using trms_sst_unlabel_dual_lsst_eq[of T']
         setops_sst_unlabel_dual_lsst_eq[of T']
         trms_sst_append[of "unlabel A" "unlabel (dual_lsst T')"]
         setops_sst_append[of "unlabel A" "unlabel (dual_lsst T')"]
         unlabel_append[of A "dual_lsst T'"]
         image_Un[of pair "setops_sst (unlabel A)" "setops_sst (unlabel T')"]
         SMP_union[of "trms_lsst A" "trms_lsst T'"]
         SMP_union[of "pair ` setops_sst (unlabel A)" "pair ` setops_sst (unlabel T')"]
     by argo+

   have 1: "SMP (trms_lsst T') ⊆ SMP M"
   proof (intro SMP_subset_I ballI)
     fix t show "t ∈ trms_lsst T' ⟹ ?P t"
       using trms_sst_wt_subst_ex[OF σα_wt σα_wf, of t "unlabel (transaction_strand T)"]
           unlabel_subst[of "transaction_strand T" "σ ∘_s α"] step.hyps(2)
       unfolding T'_def M_def by auto
   qed

   have 2: "SMP (pair ` setops_sst (unlabel T')) ⊆ SMP N"
   proof (intro SMP_subset_I ballI)
     fix t show "t ∈ pair ` setops_sst (unlabel T') ⟹ ?Q t"
       using setops_sst_wt_subst_ex[OF σα_wt σα_wf, of t "unlabel (transaction_strand T)"]
           unlabel_subst[of "transaction_strand T" "σ ∘_s α"] step.hyps(2)
       unfolding T'_def N_def by auto
   qed
```

```
    have "SMP (trms_lsst (A@dual_lsst T')) ⊆ SMP M"
         "SMP (pair ‘ setops_sst (unlabel (A@dual_lsst T'))) ⊆ SMP N"
      using 0 1 2 IH by blast+
    thus ?case unfolding T'_def M_def N_def by blast
  qed (simp add: setops_sst_def)
  thus "?A 𝒜" "?B 𝒜" by metis+
qed

lemma reachable_constraints_no_Pair_fun:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "Pair ∉ ⋃(funs_term ‘ SMP (trms_lsst A))"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

  have T_adm: "admissible_transaction T" using step.hyps(2) P unfolding list_all_iff by blast

  have σα_wt: "wt_subst (σ ∘_s α)"
    using protocol_transaction_vars_TAtom_typed(3) P(1) step.hyps(2)
          transaction_fresh_subst_transaction_renaming_wt[OF step.hyps(3,4)]
    by fast

  have σα_wf: "wf_trms (subst_range (σ ∘_s α))"
    using transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
          transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
    by (metis wf_trms_subst_compose)

  have 0: "SMP (trms_lsst (A@T')) = SMP (trms_lsst A) ∪ SMP (trms_lsst T')"
    using SMP_union[of "trms_lsst A" "trms_lsst T'"]
          unlabel_append[of A T'] trms_sst_append[of "unlabel A" "unlabel T'"]
    by simp

  have 1: "wf_trms (trms_lsst T')"
    using reachable_constraints_wf_trms[OF _ reachable_constraints.step[OF step.hyps]]
          admissible_transactions_wf_trms P
          trms_sst_append[of "unlabel A"] unlabel_append[of A]
    unfolding T'_def by force

  have 2: "Pair ∉ ⋃(funs_term ‘ (subst_range (σ ∘_s α)))"
    using transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)] by force

  have "Pair ∉ ⋃(funs_term ‘ (trms_transaction T))"
    using T_adm
    unfolding admissible_transaction_def admissible_transaction_terms_def
    by blast
  hence "Pair ∉ funs_term t"
    when t: "t ∈ trms_sst (unlabel (transaction_strand T) ·_sst σ ∘_s α)" for t
    using 2 trms_sst_funs_term_cases[OF t]
    by force
  hence 3: "Pair ∉ funs_term t" when t: "t ∈ trms_lsst T'" for t
    using t unlabel_subst[of "transaction_strand T" "σ ∘_s α"]
          trms_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·_lsst σ ∘_s α"]
    unfolding T'_def by metis

  have "∃a. Γ_v x = TAtom a" when "x ∈ vars_transaction T" for x
    using that protocol_transaction_vars_TAtom_typed(1) P step.hyps(2)
    by fast
  hence "∃a. Γ_v x = TAtom a" when "x ∈ vars_sst (unlabel (transaction_strand T) ·_sst σ ∘_s α)" for x
    using wt_subst_fv_set_termtype_subterm[OF _ σα_wt σα_wf, of x "vars_transaction T"]
          vars_sst_subst_cases[OF that]
```

```
    by fastforce
  hence "∃a. Γv x = TAtom a" when "x ∈ vars_lsst T'" for x
    using that unlabel_subst[of "transaction_strand T" "σ ∘s α"]
          vars_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
    unfolding T'_def
    by simp
  hence "∃a. Γv x = TAtom a" when "x ∈ fv_set (trms_lsst T')" for x
    using that fv_trms_sst_subset(1) by fast
  hence "Pair ∉ funs_term (Γ (Var x))" when "x ∈ fv_set (trms_lsst T')" for x
    using that by fastforce
  moreover have "Pair ∈ funs_term s"
    when s: "Ana s = (K, M)" "Pair ∈ ⋃(funs_term ‘ set K)"
    for s::"('fun,'atom,'sets) prot_term" and K M
  proof (cases s)
    case (Fun f S) thus ?thesis using s Ana_Fu_keys_not_pairs[of _ S K M] by (cases f) force+
  qed (use s in simp)
  ultimately have "Pair ∉ funs_term t" when t: "t ∈ SMP (trms_lsst T')" for t
    using t 3 SMP_funs_term[OF t _ _ 1, of Pair] funs_term_type_iff by fastforce
  thus ?case using 0 step.IH(1) unfolding T'_def by blast
qed simp

lemma reachable_constraints_setops_form:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ pair ‘ setops_sst (unlabel A)"
  shows "∃c s. t = pair (c, Fun (Set s) []) ∧ Γ c = TAtom Value"
using A t
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)

  have T_adm: "admissible_transaction T" when "T ∈ set P" for T
    using P that unfolding list_all_iff by simp

  have T_adm':
      "admissible_transaction_selects T"
      "admissible_transaction_checks T"
      "admissible_transaction_updates T"
    when "T ∈ set P" for T
    using T_adm[OF that] unfolding admissible_transaction_def by simp_all

  have T_valid: "wellformed_transaction T" when "T ∈ set P" for T
    using T_adm[OF that] unfolding admissible_transaction_def by blast

  have σα_wt: "wt_subst (σ ∘s α)"
    using protocol_transaction_vars_TAtom_typed(3) P(1) step.hyps(2)
          transaction_fresh_subst_transaction_renaming_wt[OF step.hyps(3,4)]
    by fast

  have σα_wf: "wf_trms (subst_range (σ ∘s α))"
    using transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
          transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
    by (metis wf_trms_subst_compose)

  show ?case using step.IH
  proof (cases "t ∈ pair ‘ setops_sst (unlabel A)")
    case False
    hence "t ∈ pair ‘ setops_sst (unlabel (transaction_strand T) ·sst σ ∘s α)"
      using step.prems setops_sst_append unlabel_append
            setops_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
            unlabel_subst[of "transaction_strand T" "σ ∘s α"]
      by fastforce
    then obtain t' δ where t':
        "t' ∈ pair ‘ setops_sst (unlabel (transaction_strand T))"
```

```
      "wt_subst δ" "wf_trms (subst_range δ)" "t = t' · δ"
    using setops_sst_wt_subst_ex[OF σα_wt σα_wf] by blast
  then obtain s s' where s: "t' = pair (s,s')"
    using setops_sst_are_pairs by fastforce
  moreover have "InSet ac s s' = InSet Assign s s' ∨ InSet ac s s' = InSet Check s s'" for ac
    by (cases ac) simp_all
  ultimately have "∃n. s = Var (Var Value, n)" "∃u. s' = Fun (Set u) []"
    using t'(1) setops_sst_member_iff[of s s' "unlabel (transaction_strand T)"]
          pair_in_pair_image_iff[of s s']
          transaction_inserts_are_Value_vars[
            OF T_valid[OF step.hyps(2)] T_adm'(3)[OF step.hyps(2)], of s s']
          transaction_deletes_are_Value_vars[
            OF T_valid[OF step.hyps(2)] T_adm'(3)[OF step.hyps(2)], of s s']
          transaction_selects_are_Value_vars[
            OF T_valid[OF step.hyps(2)] T_adm'(1)[OF step.hyps(2)], of s s']
          transaction_inset_checks_are_Value_vars[
            OF T_valid[OF step.hyps(2)] T_adm'(2)[OF step.hyps(2)], of s s']
          transaction_notinset_checks_are_Value_vars[
            OF T_valid[OF step.hyps(2)] T_adm'(2)[OF step.hyps(2)], of _ _ _ s s']
    by metis+
  then obtain ss n where ss: "t = pair (δ (Var Value, n), Fun (Set ss) [])"
    using t'(4) s unfolding pair_def by force

  have "Γ (δ (Var Value, n)) = TAtom Value" "wf_trm (δ (Var Value, n))"
    using t'(2) wt_subst_trm''[OF t'(2), of "Var (Var Value, n)"] apply simp
    using t'(3) by (cases "(Var Value, n) ∈ subst_domain δ") auto
  thus ?thesis using ss by blast
 qed simp
qed (simp add: setops_sst_def)

lemma reachable_constraints_setops_type:
 fixes t::"('fun,'atom,'sets) prot_term"
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. admissible_transaction T"
   and t: "t ∈ pair ` setops_sst (unlabel A)"
 shows "Γ t = TComp Pair [TAtom Value, TAtom SetType]"
proof -
 obtain s c where s: "t = pair (c, Fun (Set s) [])" "Γ c = TAtom Value"
   using reachable_constraints_setops_form[OF A P t] by moura
 hence "(Fun (Set s) []::('fun,'atom,'sets) prot_term) ∈ trms_lsst A"
   using t setops_sst_member_iff[of c "Fun (Set s) []" "unlabel A"]
   by force
 hence "wf_trm (Fun (Set s) []::('fun,'atom,'sets) prot_term)"
   using reachable_constraints_wf(2) P A
   unfolding admissible_transaction_def admissible_transaction_terms_def by blast
 hence "arity (Set s) = 0" unfolding wf_trm_def by simp
 thus ?thesis using s unfolding pair_def by fastforce
qed

lemma reachable_constraints_setops_same_type_if_unifiable:
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. admissible_transaction T"
 shows "∀s ∈ pair ` setops_sst (unlabel A). ∀t ∈ pair ` setops_sst (unlabel A).
        (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
   (is "?P A")
using reachable_constraints_setops_type[OF A P] by simp

lemma reachable_constraints_setops_unfiable_if_wt_instance_unifiable:
 assumes A: "A ∈ reachable_constraints P"
   and P: "∀T ∈ set P. admissible_transaction T"
 shows "∀s ∈ pair ` setops_sst (unlabel A). ∀t ∈ pair ` setops_sst (unlabel A).
        (∃σ ϑ ϱ. wt_subst σ ∧ wt_subst ϑ ∧ wf_trms (subst_range σ) ∧ wf_trms (subst_range ϑ) ∧
                 Unifier ϱ (s · σ) (t · ϑ))
```

```
        ⟶ (∃δ. Unifier δ s t)"
proof (intro ballI impI)
  fix s t assume st: "s ∈ pair ‘ setops_sst (unlabel A)" "t ∈ pair ‘ setops_sst (unlabel A)" and
    "∃σ ϑ ϱ. wt_subst σ ∧ wt_subst ϑ ∧ wf_trms (subst_range σ) ∧ wf_trms (subst_range ϑ) ∧
          Unifier ϱ (s · σ) (t · ϑ)"
  then obtain σ ϑ ϱ where σ:
      "wt_subst σ" "wt_subst ϑ" "wf_trms (subst_range σ)" "wf_trms (subst_range ϑ)"
      "Unifier ϱ (s · σ) (t · ϑ)"
    by moura

  obtain fs ft cs ct where c:
      "s = pair (cs, Fun (Set fs) [])" "t = pair (ct, Fun (Set ft) [])"
      "Γ cs = TAtom Value" "Γ ct = TAtom Value"
    using reachable_constraints_setops_form[OF A P st(1)]
          reachable_constraints_setops_form[OF A P st(2)]
    by moura

  have "cs ∈ subterms_set (trms_lsst A)" "ct ∈ subterms_set (trms_lsst A)"
    using c(1,2) setops_subterm_trms[OF st(1), of cs] setops_subterm_trms[OF st(2), of ct]
          Fun_param_is_subterm[of cs "args s"] Fun_param_is_subterm[of ct "args t"]
    unfolding pair_def by simp_all
  moreover have
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    using P unfolding admissible_transaction_def admissible_transaction_terms_def by fast+
  ultimately have *: "wf_trm cs" "wf_trm ct"
    using reachable_constraints_wf(2)[OF _ _ A] wf_trms_subterms by blast+

  have "(∃x. cs = Var x) ∨ (∃c d. cs = Fun c [])"
    using const_type_inv_wf c(3) *(1) by (cases cs) auto
  moreover have "(∃x. ct = Var x) ∨ (∃c d. ct = Fun c [])"
    using const_type_inv_wf c(4) *(2) by (cases ct) auto
  ultimately show "∃δ. Unifier δ s t"
    using reachable_constraints_setops_form[OF A P] reachable_constraints_setops_type[OF A P] st σ c
    unfolding pair_def by auto
qed

lemma reachable_constraints_tfr:
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and A: "A ∈ reachable_constraints P"
  shows "tfr_sst (unlabel A)"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

  have P':
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. wf_trms (trms_transaction T)"
    using P(1) protocol_transaction_vars_TAtom_typed(3) admissible_transactions_wf_trms
    by blast+

  have AT'_reach: "A@T' ∈ reachable_constraints P"
    using reachable_constraints.step[OF step.hyps] unfolding T'_def by metis
```

```
  have σα_wt: "wt_subst (σ ∘_s α)"
    using P'(1) step.hyps(2) transaction_fresh_subst_transaction_renaming_wt[OF step.hyps(3,4)]
    by fast

  have σα_wf: "wf_trms (subst_range (σ ∘_s α))"
    using transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
          transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
    by (metis wf_trms_subst_compose)

  have σα_bvars_disj: "bvars_lsst (transaction_strand T) ∩ range_vars (σ ∘_s α) = {}"
    by (rule transaction_fresh_subst_transaction_renaming_subst_vars_disj(4)[OF step.hyps(3,4,2)])

  have wf_trms_M: "wf_trms M"
    using admissible_transactions_wf_trms P(1)
    unfolding M(1) by blast

  have "tfr_set (trms_lsst (A@T'))"
    using reachable_constraints_SMP_subset(1)[OF AT'_reach P'(1)]
          tfr_subset(3)[OF M(4), of "trms_lsst (A@T')"]
          SMP_SMP_subset[of M N] SMP_I'[OF wf_trms_M M(5,2)]
    unfolding M(1) by blast
  moreover have "∀p. Ana (pair p) = ([],[])" unfolding pair_def by auto
  ultimately have 1: "tfr_set (trms_lsst (A@T') ∪ pair ‘ setops_sst (unlabel (A@T')))"
    using tfr_setops_if_tfr_trms[of "unlabel (A@T')"]
          reachable_constraints_no_Pair_fun[OF AT'_reach P(1)]
          reachable_constraints_setops_same_type_if_unifiable[OF AT'_reach P(1)]
          reachable_constraints_setops_unfiable_if_wt_instance_unifiable[OF AT'_reach P(1)]
    by blast

  have "list_all tfr_sstp (unlabel (transaction_strand T))"
    using step.hyps(2) P(2) tfr_sstp_is_comp_tfr_sstp
    unfolding comp_tfr_sst_def tfr_sst_def by fastforce
  hence "list_all tfr_sstp (unlabel T')"
    using tfr_sstp_all_wt_subst_apply[OF _ σα_wt σα_wf σα_bvars_disj]
          dual_lsst_tfr_sstp[of "transaction_strand T ·_lsst σ ∘_s α"]
          unlabel_subst[of "transaction_strand T" "σ ∘_s α"]
    unfolding T'_def by argo
  hence 2: "list_all tfr_sstp (unlabel (A@T'))"
    using step.IH unlabel_append
    unfolding tfr_sst_def by auto

  have "tfr_sst (unlabel (A@T'))" using 1 2 by (metis tfr_sst_def)
  thus ?case by (metis T'_def)
qed simp

lemma reachable_constraints_tfr':
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ‘ setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and A: "A ∈ reachable_constraints P"
  shows "tfr_sst (unlabel A)"
using A
proof (induction A rule: reachable_constraints.induct)
  case (step A T σ α)
  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"
```

```
  have AT'_reach: "A@T' ∈ reachable_constraints P"
    using reachable_constraints.step[OF step.hyps] unfolding T'_def by metis

  have σα_wt: "wt_subst (σ ∘_s α)"
    using P(1) step.hyps(2) transaction_fresh_subst_transaction_renaming_wt[OF step.hyps(3,4)]
    by fast

  have σα_wf: "wf_trms (subst_range (σ ∘_s α))"
    using transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
          transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
    by (metis wf_trms_subst_compose)

  have σα_bvars_disj: "bvars_lsst (transaction_strand T) ∩ range_vars (σ ∘_s α) = {}"
    by (rule transaction_fresh_subst_transaction_renaming_subst_vars_disj(4)[OF step.hyps(3,4,2)])

  have wf_trms_M: "wf_trms M"
    using P(2) setops_sst_wf_trms(2) unfolding M(1) pair_code wf_trms_code[symmetric] by fast

  have "SMP (trms_lsst (A@T')) ⊆ SMP M" "SMP (pair ` setops_sst (unlabel (A@T'))) ⊆ SMP M"
    using reachable_constraints_SMP_subset[OF AT'_reach P(1)]
          SMP_mono[of "⋃T ∈ set P. trms_transaction T" M]
          SMP_mono[of "⋃T ∈ set P. pair ` setops_transaction T" M]
    unfolding M(1) pair_code[symmetric] by blast+
  hence 1: "tfr_set (trms_lsst (A@T') ∪ pair ` setops_sst (unlabel (A@T')))"
    using tfr_subset(3)[OF M(4), of "trms_lsst (A@T') ∪ pair ` setops_sst (unlabel (A@T'))"]
          SMP_union[of "trms_lsst (A@T')" "pair ` setops_sst (unlabel (A@T'))"]
          SMP_SMP_subset[of M N] SMP_I'[OF wf_trms_M M(5,2)]
    by blast

  have "list_all tfr_sstp (unlabel (transaction_strand T))"
    using step.hyps(2) P(3) tfr_sstp_is_comp_tfr_sstp
    unfolding comp_tfr_sst_def tfr_sst_def by fastforce
  hence "list_all tfr_sstp (unlabel T')"
    using tfr_sstp_all_wt_subst_apply[OF _ σα_wt σα_wf σα_bvars_disj]
          dual_lsst_tfr_sstp[of "transaction_strand T ·_lsst σ ∘_s α"]
          unlabel_subst[of "transaction_strand T" "σ ∘_s α"]
    unfolding T'_def by argo
  hence 2: "list_all tfr_sstp (unlabel (A@T'))"
    using step.IH unlabel_append
    unfolding tfr_sst_def by auto

  have "tfr_sst (unlabel (A@T'))" using 1 2 by (metis tfr_sst_def)
  thus ?case by (metis T'_def)
qed simp

lemma reachable_constraints_typing_cond_sst:
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and A: "A ∈ reachable_constraints P"
  shows "typing_cond_sst (unlabel A)"
using reachable_constraints_wf[OF P(1,2) A] reachable_constraints_tfr'[OF M P(3,2,4) A]
unfolding typing_cond_sst_def by blast

context
```

```
begin
private lemma reachable_constraints_par_comp_lsst_aux:
  fixes P
  defines "Ts ≡ concat (map transaction_strand P)"
  assumes P_fresh_wf: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    (is "∀T ∈ set P. ?fresh_wf T")
    and A: "A ∈ reachable_constraints P"
  shows "∀b ∈ set (dual_lsst A). ∃a ∈ set Ts. ∃δ. b = a ·_lsstp δ ∧
      wt_subst δ ∧ wf_trms (subst_range δ) ∧
      (∀t ∈ subst_range δ. (∃x. t = Var x) ∨ (∃c. t = Fun c []))"
    (is "∀b ∈ set (dual_lsst A). ∃a ∈ set Ts. ?P b a")
using A
proof (induction A rule: reachable_constraints.induct)
  case (step 𝒜 T σ α)
  define Q where "Q ≡ ?P"
  define ϑ where "ϑ ≡ σ ∘_s α"

  let ?R = "λA Ts. ∀b ∈ set A. ∃a ∈ set Ts. Q b a"

  have T_fresh_wf: "?fresh_wf T" using step.hyps(2) P_fresh_wf by blast

  have "wt_subst ϑ" "wf_trms (subst_range ϑ)"
       "∀t ∈ subst_range ϑ. (∃x. t = Var x) ∨ (∃c. t = Fun c [])"
    using wt_subst_compose[
            OF transaction_fresh_subst_wt[OF step.hyps(3) T_fresh_wf]
               transaction_renaming_subst_wt[OF step.hyps(4)]]
          wf_trms_subst_compose[
            OF transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
               transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]]
          transaction_fresh_subst_transaction_renaming_subst_range'[OF step.hyps(3,4)]
    unfolding ϑ_def by metis+
  hence "?R (dual_lsst (dual_lsst (transaction_strand T)) ·_lsst ϑ) (transaction_strand T)"
    using dual_lsst_self_inverse[of "transaction_strand T"]
    by (auto simp add: Q_def subst_apply_labeled_stateful_strand_def)
  hence "?R (dual_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))) (transaction_strand T)"
    by (metis dual_lsst_subst)
  hence "?R (dual_lsst (dual_lsst (transaction_strand T ·_lsst ϑ))) Ts"
    using step.hyps(2) unfolding Ts_def dual_lsst_def by fastforce
  thus ?case using step.IH unfolding Q_def ϑ_def by auto
qed simp

lemma reachable_constraints_par_comp_lsst:
  fixes P
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Ts ≡ concat (map transaction_strand P)"
  assumes P_pc: "comp_par_comp_lsst public arity Ana Γ Pair Ts M S"
    and P_wf: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    and A: "A ∈ reachable_constraints P"
  shows "par_comp_lsst A ((f (set S)) - {m. intruder_synth {} m})"
using par_comp_lsst_if_comp_par_comp_lsst'[OF P_pc, of "dual_lsst A", THEN par_comp_lsst_dual_lsst]
      reachable_constraints_par_comp_lsst_aux[OF P_wf A, unfolded Ts_def[symmetric]]
unfolding f_def dual_lsst_self_inverse by fast
end

lemma reachable_constraints_par_comp_constr:
  fixes P f S
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Ts ≡ concat (map transaction_strand P)"
    and "Sec ≡ (f (set S)) - {m. intruder_synth {} m}"
    and "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ‘ setops_transaction T"
  assumes M:
      "has_all_wt_instances_of Γ M N"
      "finite N"
```

```
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
      "comp_par_comp_lsst public arity Ana Γ Pair Ts M_fun S"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
    and ℐ: "constraint_model ℐ 𝒜"
  shows "∃ℐ_τ. welltyped_constraint_model ℐ_τ 𝒜 ∧
              ((∀n. welltyped_constraint_model ℐ_τ (proj n 𝒜)) ∨
               (∃𝒜'. prefix 𝒜' 𝒜 ∧ strand_leaks_lsst 𝒜' Sec ℐ_τ))"
proof -
  have ℐ': "constr_sem_stateful ℐ (unlabel 𝒜)" "interpretation_subst ℐ"
    using ℐ unfolding constraint_model_def by blast+

  show ?thesis
    using reachable_constraints_par_comp_lsst[OF P(5,3)[unfolded Ts_def] 𝒜]
          reachable_constraints_typing_cond_sst[OF M_def M P(1,2,3,4) 𝒜]
          par_comp_constr_stateful[OF _ _ ℐ', of Sec]
    unfolding f_def Sec_def welltyped_constraint_model_def constraint_model_def by blast
qed

end

end
```

2.4 Term Variants (Term_Variants)

```
theory Term_Variants
  imports Stateful_Protocol_Composition_and_Typing.Intruder_Deduction
begin

fun term_variants where
  "term_variants P (Var x) = [Var x]"
| "term_variants P (Fun f T) = (
  let S = product_lists (map (term_variants P) T)
  in map (Fun f) S@concat (map (λg. map (Fun g) S) (P f)))"

inductive term_variants_pred where
  term_variants_Var:
  "term_variants_pred P (Var x) (Var x)"
| term_variants_P:
  "⟦length T = length S; ⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i); g ∈ set (P f)⟧
   ⟹ term_variants_pred P (Fun f T) (Fun g S)"
| term_variants_Fun:
  "⟦length T = length S; ⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)⟧
   ⟹ term_variants_pred P (Fun f T) (Fun f S)"

lemma term_variants_pred_inv:
  assumes "term_variants_pred P (Fun f T) (Fun h S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)"
    and "f ≠ h ⟹ h ∈ set (P f)"
using assms by (auto elim: term_variants_pred.cases)

lemma term_variants_pred_inv':
  assumes "term_variants_pred P (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (args t ! i)"
```

```
    and "f ≠ the_Fun t ⟹ the_Fun t ∈ set (P f)"
    and "P ≡ (λ_. [])(g := [h]) ⟹ f ≠ the_Fun t ⟹ f = g ∧ the_Fun t = h"
using assms by (auto elim: term_variants_pred.cases)

lemma term_variants_pred_inv'':
  assumes "term_variants_pred P t (Fun f T)"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ term_variants_pred P (args t ! i) (T ! i)"
    and "f ≠ the_Fun t ⟹ f ∈ set (P (the_Fun t))"
    and "P ≡ (λ_. [])(g := [h]) ⟹ f ≠ the_Fun t ⟹ f = h ∧ the_Fun t = g"
using assms by (auto elim: term_variants_pred.cases)

lemma term_variants_pred_inv_Var:
  "term_variants_pred P (Var x) t ⟷ t = Var x"
  "term_variants_pred P t (Var x) ⟷ t = Var x"
by (auto intro: term_variants_Var elim: term_variants_pred.cases)

lemma term_variants_pred_inv_const:
  "term_variants_pred P (Fun c []) t ⟷ ((∃g ∈ set (P c). t = Fun g []) ∨ (t = Fun c []))"
by (auto intro: term_variants_P term_variants_Fun elim: term_variants_pred.cases)

lemma term_variants_pred_refl: "term_variants_pred P t t"
by (induct t) (auto intro: term_variants_pred.intros)

lemma term_variants_pred_refl_inv:
  assumes st: "term_variants_pred P s t"
    and P: "∀f. ∀g ∈ set (P f). f = g"
  shows "s = t"
  using st P
proof (induction s t rule: term_variants_pred.induct)
case (term_variants_Var P x) thus ?case by blast
next
  case (term_variants_P T S P g f)
  hence "T ! i = S ! i" when i: "i < length T" for i using i by blast
  hence "T = S" using term_variants_P.hyps(1) by (simp add: nth_equalityI)
  thus ?case using term_variants_P.prems term_variants_P.hyps(3) by fast
next
  case (term_variants_Fun T S P f)
  hence "T ! i = S ! i" when i: "i < length T" for i using i by blast
  hence "T = S" using term_variants_Fun.hyps(1) by (simp add: nth_equalityI)
  thus ?case by fast
qed

lemma term_variants_pred_const:
  assumes "b ∈ set (P a)"
  shows "term_variants_pred P (Fun a []) (Fun b [])"
using term_variants_P[of "[]" "[]"] assms by simp

lemma term_variants_pred_const_cases:
  "P a ≠ [] ⟹ term_variants_pred P (Fun a []) t ⟷
                (t = Fun a [] ∨ (∃b ∈ set (P a). t = Fun b []))"
  "P a = [] ⟹ term_variants_pred P (Fun a []) t ⟷ t = Fun a []"
using term_variants_pred_inv_const[of P] by auto

lemma term_variants_pred_param:
  assumes "term_variants_pred P t s"
    and fg: "f = g ∨ g ∈ set (P f)"
  shows "term_variants_pred P (Fun f (S@t#T)) (Fun g (S@s#T))"
proof -
  have 1: "length (S@t#T) = length (S@s#T)" by simp

  have "term_variants_pred P (T ! i) (T ! i)" "term_variants_pred P (S ! i) (S ! i)" for i
```

```
    by (metis term_variants_pred_refl)+
  hence 2: "term_variants_pred P ((S@t#T) ! i) ((S@s#T) ! i)" for i
    by (simp add: assms nth_Cons' nth_append)

  show ?thesis by (metis term_variants_Fun[OF 1 2] term_variants_P[OF 1 2] fg)
qed

lemma term_variants_pred_Cons:
  assumes t: "term_variants_pred P t s"
    and T: "term_variants_pred P (Fun f T) (Fun f S)"
    and fg: "f = g ∨ g ∈ set (P f)"
  shows "term_variants_pred P (Fun f (t#T)) (Fun g (s#S))"
proof -
  have 1: "length (t#T) = length (s#S)"
      and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)"
    using term_variants_pred_inv[OF T] by simp_all
  hence 2: "⋀i. i < length (t#T) ⟹ term_variants_pred P ((t#T) ! i) ((s#S) ! i)"
    by (metis t One_nat_def diff_less length_Cons less_Suc_eq less_imp_diff_less nth_Cons'
          zero_less_Suc)

  show ?thesis using 1 2 fg by (auto intro: term_variants_pred.intros)
qed

lemma term_variants_pred_dense:
  fixes P Q::"'a set" and fs gs::"'a list"
  defines "P_fs x ≡ if x ∈ P then fs else []"
    and "P_gs x ≡ if x ∈ P then gs else []"
    and "Q_fs x ≡ if x ∈ Q then fs else []"
  assumes ut: "term_variants_pred P_fs u t"
    and g: "g ∈ Q" "g ∈ set gs"
  shows "∃s. term_variants_pred P_gs u s ∧ term_variants_pred Q_fs s t"
proof -
  define F where "F ≡ λ(P::'a set) (fs::'a list) x. if x ∈ P then fs else []"

  show ?thesis using ut g P_fs_def unfolding P_gs_def Q_fs_def
  proof (induction P_fs u t arbitrary: g gs rule: term_variants_pred.induct)
    case (term_variants_Var P h x) thus ?case
      by (auto intro: term_variants_pred.term_variants_Var)
  next
    case (term_variants_P T S P' h' h g gs)
    note hyps = term_variants_P.hyps(1,2,4,5,6,7)
    note IH = term_variants_P.hyps(3)

    have "∃s. term_variants_pred (F P gs) (T ! i) s ∧ term_variants_pred (F Q fs) s (S ! i)"
      when i: "i < length T" for i
      using IH[OF i hyps(4,5,6)] unfolding F_def by presburger
    then obtain U where U:
        "length T = length U" "⋀i. i < length T ⟹ term_variants_pred (F P gs) (T ! i) (U ! i)"
        "length U = length S" "⋀i. i < length U ⟹ term_variants_pred (F Q fs) (U ! i) (S ! i)"
      using hyps(1) Skolem_list_nth[of _ "λi s. term_variants_pred (F P gs) (T ! i) s ∧
                                          term_variants_pred (F Q fs) s (S ! i)"]
      by moura

    show ?case
      using term_variants_pred.term_variants_P[OF U(1,2), of g h]
            term_variants_pred.term_variants_P[OF U(3,4), of h' g]
            hyps(3)[unfolded hyps(6)] hyps(4,5)
      unfolding F_def by force
  next
    case (term_variants_Fun T S P' h' g gs)
    note hyps = term_variants_Fun.hyps(1,2,4,5,6)
    note IH = term_variants_Fun.hyps(3)
```

```
    have "∃s. term_variants_pred (F P gs) (T ! i) s ∧ term_variants_pred (F Q fs) s (S ! i)"
      when i: "i < length T" for i
      using IH[OF i hyps(3,4,5)] unfolding F_def by presburger
    then obtain U where U:
        "length T = length U" "⋀i. i < length T ⟹ term_variants_pred (F P gs) (T ! i) (U ! i)"
        "length U = length S" "⋀i. i < length U ⟹ term_variants_pred (F Q fs) (U ! i) (S ! i)"
      using hyps(1) Skolem_list_nth[of _ "λi s. term_variants_pred (F P gs) (T ! i) s ∧
                                          term_variants_pred (F Q fs) s (S ! i)"]
      by moura

    thus ?case
      using term_variants_pred.term_variants_Fun[OF U(1,2)]
            term_variants_pred.term_variants_Fun[OF U(3,4)]
      unfolding F_def by meson
  qed
qed

lemma term_variants_pred_dense':
  assumes ut: "term_variants_pred ((λ_. [])(a := [b])) u t"
  shows "∃s. term_variants_pred ((λ_. [])(a := [c])) u s ∧
             term_variants_pred ((λ_. [])(c := [b])) s t"
using ut term_variants_pred_dense[of "{a}" "[b]" u t c "{c}" "[c]"]
unfolding fun_upd_def by simp

lemma term_variants_pred_eq_case:
  fixes t s::"('a,'b) term"
  assumes "term_variants_pred P t s" "∀f ∈ funs_term t. P f = []"
  shows "t = s"
using assms
proof (induction P t s rule: term_variants_pred.induct)
  case (term_variants_Fun T S P f) thus ?case
    using subtermeq_imp_funs_term_subset[OF Fun_param_in_subterms[OF nth_mem], of _ T f]
          nth_equalityI[of T S]
    by blast
qed (simp_all add: term_variants_pred_refl)

lemma term_variants_pred_subst:
  assumes "term_variants_pred P t s"
  shows "term_variants_pred P (t · δ) (s · δ)"
using assms
proof (induction P t s rule: term_variants_pred.induct)
  case (term_variants_P T S P f g)
  have 1: "length (map (λt. t · δ) T) = length (map (λt. t · δ) S)"
    using term_variants_P.hyps
    by simp

  have 2: "term_variants_pred P ((map (λt. t · δ) T) ! i) ((map (λt. t · δ) S) ! i)"
    when "i < length (map (λt. t · δ) T)" for i
    using term_variants_P that
    by fastforce

  show ?case
    using term_variants_pred.term_variants_P[OF 1 2 term_variants_P.hyps(3)]
    by fastforce
next
  case (term_variants_Fun T S P f)
  have 1: "length (map (λt. t · δ) T) = length (map (λt. t · δ) S)"
    using term_variants_Fun.hyps
    by simp

  have 2: "term_variants_pred P ((map (λt. t · δ) T) ! i) ((map (λt. t · δ) S) ! i)"
    when "i < length (map (λt. t · δ) T)" for i
    using term_variants_Fun that
```

```
    by fastforce

  show ?case
    using term_variants_pred.term_variants_Fun[OF 1 2]
    by fastforce
qed (simp add: term_variants_pred_refl)

lemma term_variants_pred_subst':
  fixes t s::"('a,'b) term" and δ::"('a,'b) subst"
  assumes "term_variants_pred P (t · δ) s"
    and "∀x ∈ fv t ∪ fv s. (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
  shows "∃u. term_variants_pred P t u ∧ s = u · δ"
using assms
proof (induction P "t · δ" s arbitrary: t rule: term_variants_pred.induct)
  case (term_variants_Var P x g) thus ?case using term_variants_pred_refl by fast
next
  case (term_variants_P T S P g f) show ?case
  proof (cases t)
    case (Var x) thus ?thesis
      using term_variants_P.hyps(4,5) term_variants_P.prems
      by fastforce
  next
    case (Fun h U)
    hence 1: "h = f" "T = map (λs. s · δ) U" "length U = length T"
      using term_variants_P.hyps(5) by simp_all
    hence 2: "T ! i = U ! i · δ" when "i < length T" for i
      using that by simp

    have "∀x ∈ fv (U ! i) ∪ fv (S ! i). (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
      when "i < length U" for i
      using that Fun term_variants_P.prems term_variants_P.hyps(1) 1(3)
      by force
    hence IH: "∀i < length U. ∃u. term_variants_pred P (U ! i) u ∧ S ! i = u · δ"
      by (metis 1(3) term_variants_P.hyps(3)[OF _ 2])

    have "∃V. length U = length V ∧ S = map (λv. v · δ) V ∧
              (∀i < length U. term_variants_pred P (U ! i) (V ! i))"
      using term_variants_P.hyps(1) 1(3) subst_term_list_obtain[OF IH] by metis
    then obtain V where V: "length U = length V" "S = map (λv. v · δ) V"
                           "⋀i. i < length U ⟹ term_variants_pred P (U ! i) (V ! i)"
      by moura

    have "term_variants_pred P (Fun f U) (Fun g V)"
      by (metis term_variants_pred.term_variants_P[OF V(1,3) term_variants_P.hyps(4)])
    moreover have "Fun g S = Fun g V · δ" using V(2) by simp
    ultimately show ?thesis using term_variants_P.hyps(1,4) Fun 1 by blast
  qed
next
  case (term_variants_Fun T S P f t) show ?case
  proof (cases t)
    case (Var x)
    hence "T = []" "P f = []" using term_variants_Fun.hyps(4) term_variants_Fun.prems by fastforce+
    thus ?thesis using term_variants_pred_refl Var term_variants_Fun.hyps(1,4) by fastforce
  next
    case (Fun h U)
    hence 1: "h = f" "T = map (λs. s · δ) U" "length U = length T"
      using term_variants_Fun.hyps(4) by simp_all
    hence 2: "T ! i = U ! i · δ" when "i < length T" for i
      using that by simp

    have "∀x ∈ fv (U ! i) ∪ fv (S ! i). (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
      when "i < length U" for i
      using that Fun term_variants_Fun.prems term_variants_Fun.hyps(1) 1(3)
```

```
    by force
  hence IH: "∀i < length U. ∃u. term_variants_pred P (U ! i) u ∧ S ! i = u · δ"
    by (metis 1(3) term_variants_Fun.hyps(3)[OF _ 2 ])

  have "∃V. length U = length V ∧ S = map (λv. v · δ) V ∧
            (∀i < length U. term_variants_pred P (U ! i) (V ! i))"
    using term_variants_Fun.hyps(1) 1(3) subst_term_list_obtain[OF IH] by metis
  then obtain V where V: "length U = length V" "S = map (λv. v · δ) V"
                       "⋀i. i < length U ⟹ term_variants_pred P (U ! i) (V ! i)"
    by moura

  have "term_variants_pred P (Fun f U) (Fun f V)"
    by (metis term_variants_pred.term_variants_Fun[OF V(1,3)])
  moreover have "Fun f S = Fun f V · δ" using V(2) by simp
  ultimately show ?thesis using term_variants_Fun.hyps(1) Fun 1 by blast
 qed
qed

lemma term_variants_pred_iff_in_term_variants:
 fixes t::"('a,'b) term"
 shows "term_variants_pred P t s ⟷ s ∈ set (term_variants P t)"
   (is "?A t s ⟷ ?B t s")
proof
 define U where "U ≡ λP (T::('a,'b) term list). product_lists (map (term_variants P) T)"

 have a:
     "g ∈ set (P f) ⟹ set (map (Fun g) (U P T)) ⊆ set (term_variants P (Fun f T))"
     "set (map (Fun f) (U P T)) ⊆ set (term_variants P (Fun f T))"
   for f P g and T::"('a,'b) term list"
   using term_variants.simps(2)[of P f T]
   unfolding U_def Let_def by auto

 have b: "∃S ∈ set (U P T). s = Fun f S ∨ (∃g ∈ set (P f). s = Fun g S)"
   when "s ∈ set (term_variants P (Fun f T))" for P T f s
   using that by (cases "P f") (auto simp add: U_def Let_def)

 have c: "length T = length S" when "S ∈ set (U P T)" for S P T
   using that unfolding U_def
   by (simp add: in_set_product_lists_length)

 show "?A t s ⟹ ?B t s"
 proof (induction P t s rule: term_variants_pred.induct)
   case (term_variants_P T S P g f)
   note hyps = term_variants_P.hyps
   note IH = term_variants_P.IH

   have "S ∈ set (U P T)"
     using IH hyps(1) product_lists_in_set_nth'[of _ S]
     unfolding U_def by simp
   thus ?case using a(1)[of _ P, OF hyps(3)] by auto
 next
   case (term_variants_Fun T S P f)
   note hyps = term_variants_Fun.hyps
   note IH = term_variants_Fun.IH

   have "S ∈ set (U P T)"
     using IH hyps(1) product_lists_in_set_nth'[of _ S]
     unfolding U_def by simp
   thus ?case using a(2)[of f P T] by (cases "P f") auto
 qed (simp add: term_variants_Var)

 show "?B t s ⟹ ?A t s"
 proof (induction P t arbitrary: s rule: term_variants.induct)
```

```
    case (2 P f T)
    obtain S where S:
        "s = Fun f S ∨ (∃g ∈ set (P f). s = Fun g S)"
        "S ∈ set (U P T)" "length T = length S"
      using c b[OF "2.prems"] by moura

    have "∀i < length T. term_variants_pred P (T ! i) (S ! i)"
      using "2.IH" S product_lists_in_set_nth by (fastforce simp add: U_def)
    thus ?case using S by (auto intro: term_variants_pred.intros)
  qed (simp add: term_variants_Var)
qed

lemma term_variants_pred_finite:
  "finite {s. term_variants_pred P t s}"
using term_variants_pred_iff_in_term_variants[of P t]
by simp

lemma term_variants_pred_fv_eq:
  assumes "term_variants_pred P s t"
  shows "fv s = fv t"
using assms
by (induct rule: term_variants_pred.induct)
   (metis, metis fv_eq_FunI, metis fv_eq_FunI)

lemma (in intruder_model) term_variants_pred_wf_trms:
  assumes "term_variants_pred P s t"
    and "⋀f g. g ∈ set (P f) ⟹ arity f = arity g"
    and "wf_trm s"
  shows "wf_trm t"
using assms
apply (induction rule: term_variants_pred.induct, simp)
by (metis (no_types) wf_trmI wf_trm_arity in_set_conv_nth wf_trm_param_idx)+

lemma term_variants_pred_funs_term:
  assumes "term_variants_pred P s t"
    and "f ∈ funs_term t"
  shows "f ∈ funs_term s ∨ (∃g ∈ funs_term s. f ∈ set (P g))"
  using assms
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S P g h) thus ?case
  proof (cases "f = g")
    case False
    then obtain s where "s ∈ set S" "f ∈ funs_term s"
      using funs_term_subterms_eq(1)[of "Fun g S"] term_variants_P.prems by auto
    thus ?thesis
      using term_variants_P.IH term_variants_P.hyps(1) in_set_conv_nth[of s S] by force
  qed simp
next
  case (term_variants_Fun T S P h) thus ?case
  proof (cases "f = h")
    case False
    then obtain s where "s ∈ set S" "f ∈ funs_term s"
      using funs_term_subterms_eq(1)[of "Fun h S"] term_variants_Fun.prems by auto
    thus ?thesis
      using term_variants_Fun.IH term_variants_Fun.hyps(1) in_set_conv_nth[of s S] by force
  qed simp
qed fast

end
```

2.5 Term Implication (Term_Implication)

```
theory Term_Implication
  imports Stateful_Protocol_Model Term_Variants
begin
```

2.5.1 Single Term Implications

```
definition timpl_apply_term ("⟨_ --≫ _⟩⟨_⟩") where
  "⟨a --≫ b⟩⟨t⟩ ≡ term_variants ((λ_. [])(Abs a := [Abs b])) t"

definition timpl_apply_terms ("⟨_ --≫ _⟩⟨_⟩set") where
  "⟨a --≫ b⟩⟨M⟩set ≡ ⋃((set ∘ timpl_apply_term a b) ` M)"

lemma timpl_apply_Fun:
  assumes "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --≫ b⟩⟨T ! i⟩"
    and "length T = length S"
  shows "Fun f S ∈ set ⟨a --≫ b⟩⟨Fun f T⟩"
using assms term_variants_Fun term_variants_pred_iff_in_term_variants
by (metis timpl_apply_term_def)

lemma timpl_apply_Abs:
  assumes "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --≫ b⟩⟨T ! i⟩"
    and "length T = length S"
  shows "Fun (Abs b) S ∈ set ⟨a --≫ b⟩⟨Fun (Abs a) T⟩"
using assms(1) term_variants_P[OF assms(2), of "(λ_. [])(Abs a := [Abs b])" "Abs b" "Abs a"]
unfolding timpl_apply_term_def term_variants_pred_iff_in_term_variants[symmetric]
by fastforce

lemma timpl_apply_refl: "t ∈ set ⟨a --≫ b⟩⟨t⟩"
unfolding timpl_apply_term_def
by (metis term_variants_pred_refl term_variants_pred_iff_in_term_variants)

lemma timpl_apply_const: "Fun (Abs b) [] ∈ set ⟨a --≫ b⟩⟨Fun (Abs a) []⟩"
using term_variants_pred_iff_in_term_variants term_variants_pred_const
unfolding timpl_apply_term_def by auto

lemma timpl_apply_const':
  "c = a ⟹ set ⟨a --≫ b⟩⟨Fun (Abs c) []⟩ = {Fun (Abs b) [], Fun (Abs c) []}"
  "c ≠ a ⟹ set ⟨a --≫ b⟩⟨Fun (Abs c) []⟩ = {Fun (Abs c) []}"
using term_variants_pred_const_cases[of "(λ_. [])(Abs a := [Abs b])" "Abs c"]
      term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
unfolding timpl_apply_term_def by auto

lemma timpl_apply_term_subst:
  "s ∈ set ⟨a --≫ b⟩⟨t⟩ ⟹ s · δ ∈ set ⟨a --≫ b⟩⟨t · δ⟩"
by (metis term_variants_pred_iff_in_term_variants term_variants_pred_subst timpl_apply_term_def)

lemma timpl_apply_inv:
  assumes "Fun h S ∈ set ⟨a --≫ b⟩⟨Fun f T⟩"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --≫ b⟩⟨T ! i⟩"
    and "f ≠ h ⟹ f = Abs a ∧ h = Abs b"
using assms term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
unfolding timpl_apply_term_def
by (metis (full_types) term_variants_pred_inv(1),
    metis (full_types) term_variants_pred_inv(2),
    fastforce dest: term_variants_pred_inv(3))

lemma timpl_apply_inv':
  assumes "s ∈ set ⟨a --≫ b⟩⟨Fun f T⟩"
  shows "∃g S. s = Fun g S"
proof -
```

```
  have *: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) (Fun f T) s"
    using assms term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
    unfolding timpl_apply_term_def by force
  show ?thesis using term_variants_pred.cases[OF *, of ?thesis] by fastforce
qed

lemma timpl_apply_term_Var_iff:
  "Var x ∈ set ⟨a --≫ b⟩⟨t⟩ ⟷ t = Var x"
using term_variants_pred_inv_Var term_variants_pred_iff_in_term_variants
unfolding timpl_apply_term_def by metis
```

2.5.2 Term Implication Closure

```
inductive_set timpl_closure for t TI where
  FP: "t ∈ timpl_closure t TI"
| TI: "⟦u ∈ timpl_closure t TI; (a,b) ∈ TI; term_variants_pred ((λ_. [])(Abs a := [Abs b])) u s⟧
      ⟹ s ∈ timpl_closure t TI"

definition "timpl_closure_set M TI ≡ (⋃t ∈ M. timpl_closure t TI)"

inductive_set timpl_closure'_step for TI where
  "⟦(a,b) ∈ TI; term_variants_pred ((λ_. [])(Abs a := [Abs b])) t s⟧
    ⟹ (t,s) ∈ timpl_closure'_step TI"

definition "timpl_closure' TI ≡ (timpl_closure'_step TI)*"

definition comp_timpl_closure where
  "comp_timpl_closure FP TI ≡
    let f = λX. FP ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --≫ b⟩⟨x⟩)
    in while (λX. f X ≠ X) f {}"

definition comp_timpl_closure_list where
  "comp_timpl_closure_list FP TI ≡
    let f = λX. remdups (concat (map (λx. concat (map (λ(a,b). ⟨a --≫ b⟩⟨x⟩) TI)) X))
    in while (λX. set (f X) ≠ set X) f FP"

lemma timpl_closure_setI:
  "t ∈ M ⟹ t ∈ timpl_closure_set M TI"
unfolding timpl_closure_set_def by (auto intro: timpl_closure.FP)

lemma timpl_closure_set_empty_timpls:
  "timpl_closure t {} = {t}" (is "?A = ?B")
proof (intro subset_antisym subsetI)
  fix s show "s ∈ ?A ⟹ s ∈ ?B"
    by (induct s rule: timpl_closure.induct) auto
qed (simp add: timpl_closure.FP)

lemmas timpl_closure_set_is_timpl_closure_union = meta_eq_to_obj_eq[OF timpl_closure_set_def]

lemma term_variants_pred_eq_case_Abs:
  fixes a b
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes "term_variants_pred P t s" "∀f ∈ funs_term s. ¬is_Abs f"
  shows "t = s"
using assms(2,3) P_def
proof (induction P t s rule: term_variants_pred.induct)
  case (term_variants_Fun T S f)
  have "¬is_Abs h" when i: "i < length S" and h: "h ∈ funs_term (S ! i)" for i h
    using i h term_variants_Fun.hyps(4) by auto
  hence "T ! i = S ! i" when i: "i < length T" for i using i term_variants_Fun.hyps(1,3) by auto
  hence "T = S" using term_variants_Fun.hyps(1) nth_equalityI[of T S] by fast
  thus ?case using term_variants_Fun.hyps(1) by blast
qed (simp_all add: term_variants_pred_refl)
```

```
lemma timpl_closure'_step_inv:
  assumes "(t,s) ∈ timpl_closure'_step TI"
  obtains a b where "(a,b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t s"
using assms by (auto elim: timpl_closure'_step.cases)

lemma timpl_closure_mono:
  assumes "TI ⊆ TI'"
  shows "timpl_closure t TI ⊆ timpl_closure t TI'"
proof
  fix s show "s ∈ timpl_closure t TI ⟹ s ∈ timpl_closure t TI'"
    apply (induct rule: timpl_closure.induct)
    using assms by (auto intro: timpl_closure.intros)
qed

lemma timpl_closure_set_mono:
  assumes "M ⊆ M'" "TI ⊆ TI'"
  shows "timpl_closure_set M TI ⊆ timpl_closure_set M' TI'"
using assms(1) timpl_closure_mono[OF assms(2)] unfolding timpl_closure_set_def by fast

lemma timpl_closure_idem:
  "timpl_closure_set (timpl_closure t TI) TI = timpl_closure t TI" (is "?A = ?B")
proof
  have "s ∈ timpl_closure t TI"
    when "s ∈ timpl_closure u TI" "u ∈ timpl_closure t TI"
    for s u
    using that
    by (induction rule: timpl_closure.induct)
       (auto intro: timpl_closure.intros)
  thus "?A ⊆ ?B" unfolding timpl_closure_set_def by blast

  show "?B ⊆ ?A"
    unfolding timpl_closure_set_def
    by (blast intro: timpl_closure.FP)
qed

lemma timpl_closure_set_idem:
  "timpl_closure_set (timpl_closure_set M TI) TI = timpl_closure_set M TI"
using timpl_closure_idem[of _ TI] unfolding timpl_closure_set_def by auto

lemma timpl_closure_set_mono_timpl_closure_set:
  assumes N: "N ⊆ timpl_closure_set M TI"
  shows "timpl_closure_set N TI ⊆ timpl_closure_set M TI"
using timpl_closure_set_mono[OF N, of TI TI] timpl_closure_set_idem[of M TI]
by simp

lemma timpl_closure_is_timpl_closure':
  "s ∈ timpl_closure t TI ⟷ (t,s) ∈ timpl_closure' TI"
proof
  show "s ∈ timpl_closure t TI ⟹ (t,s) ∈ timpl_closure' TI"
    unfolding timpl_closure'_def
    by (induct rule: timpl_closure.induct)
       (auto intro: rtrancl_into_rtrancl timpl_closure'_step.intros)

  show "(t,s) ∈ timpl_closure' TI ⟹ s ∈ timpl_closure t TI"
    unfolding timpl_closure'_def
    by (induct rule: rtrancl_induct)
       (auto dest: timpl_closure'_step_inv
             intro: timpl_closure.FP timpl_closure.TI)
qed

lemma timpl_closure'_mono:
  assumes "TI ⊆ TI'"
```

```
  shows "timpl_closure' TI ⊆ timpl_closure' TI'"
using timpl_closure_mono[OF assms]
      timpl_closure_is_timpl_closure'[of _ _ TI]
      timpl_closure_is_timpl_closure'[of _ _ TI']
by fast

lemma timpl_closureton_is_timpl_closure:
  "timpl_closure_set {t} TI = timpl_closure t TI"
by (simp add: timpl_closure_set_is_timpl_closure_union)

lemma timpl_closure'_timpls_trancl_subset:
  "timpl_closure' (c⁺) ⊆ timpl_closure' c"
unfolding timpl_closure'_def
proof
  fix s t::"(('a,'b,'c) prot_fun,'d) term"
  show "(s,t) ∈ (timpl_closure'_step (c⁺))* ⟹ (s,t) ∈ (timpl_closure'_step c)*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c⁺" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(u,t) ∈ (timpl_closure'_step c)*"
    proof (induction arbitrary: t rule: trancl_induct)
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      have "(u,s) ∈ (timpl_closure'_step c)*"
           "(s,t) ∈ timpl_closure'_step c"
        using step.hyps(2) s(2) step.IH[OF s(1)]
        by (auto intro: timpl_closure'_step.intros)
      thus ?case by simp
    qed (auto intro: timpl_closure'_step.intros)
    thus ?case using step.IH by simp
  qed simp
qed

lemma timpl_closure'_timpls_trancl_subset':
  "timpl_closure' {(a,b) ∈ c⁺. a ≠ b} ⊆ timpl_closure' c"
using timpl_closure'_timpls_trancl_subset
      timpl_closure'_mono[of "{(a,b) ∈ c⁺. a ≠ b}" "c⁺"]
by fast

lemma timpl_closure_set_timpls_trancl_subset:
  "timpl_closure_set M (c⁺) ⊆ timpl_closure_set M c"
using timpl_closure'_timpls_trancl_subset[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "c⁺"]
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "c⁺"]
by fastforce

lemma timpl_closure_set_timpls_trancl_subset':
  "timpl_closure_set M {(a,b) ∈ c⁺. a ≠ b} ⊆ timpl_closure_set M c"
using timpl_closure'_timpls_trancl_subset'[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "{(a,b) ∈ c⁺. a ≠ b}"]
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "{(a,b) ∈ c⁺. a ≠ b}"]
by fastforce
```

```
lemma timpl_closure'_timpls_trancl_supset':
  "timpl_closure' c ⊆ timpl_closure' {(a,b) ∈ c⁺. a ≠ b}"
unfolding timpl_closure'_def
proof
  let ?cl = "{(a,b) ∈ c⁺. a ≠ b}"

  fix s t::"(('e,'f,'c) prot_fun,'g) term"
  show "(s,t) ∈ (timpl_closure'_step c)* ⟹ (s,t) ∈ (timpl_closure'_step ?cl)*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(a,b) ∈ c⁺" by simp
    hence "(u,t) ∈ (timpl_closure'_step ?cl)*" using ab(2)
    proof (induction arbitrary: t rule: trancl_induct)
      case (base d) show ?case
      proof (cases "a = d")
        case True thus ?thesis
          using base term_variants_pred_refl_inv[of _ u t]
          by force
      next
        case False thus ?thesis
          using base timpl_closure'_step.intros[of a d ?cl]
          by fast
      qed
    next
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      show ?case
      proof (cases "d = e")
        case True
        thus ?thesis
          using step.prems step.IH[of t]
          by blast
      next
        case False
        hence "(u,s) ∈ (timpl_closure'_step ?cl)*"
              "(s,t) ∈ timpl_closure'_step ?cl"
          using step.hyps(2) s(2) step.IH[OF s(1)]
          by (auto intro: timpl_closure'_step.intros)
        thus ?thesis by simp
      qed
    qed
    thus ?case using step.IH by simp
  qed simp
qed

lemma timpl_closure'_timpls_trancl_supset:
  "timpl_closure' c ⊆ timpl_closure' (c⁺)"
using timpl_closure'_timpls_trancl_supset'[of c]
      timpl_closure'_mono[of "{(a,b) ∈ c⁺. a ≠ b}" "c⁺"]
by fast

lemma timpl_closure'_timpls_trancl_eq:
  "timpl_closure' (c⁺) = timpl_closure' c"
using timpl_closure'_timpls_trancl_subset timpl_closure'_timpls_trancl_supset
by blast
```

```
lemma timpl_closure'_timpls_trancl_eq':
  "timpl_closure' {(a,b) ∈ c⁺. a ≠ b} = timpl_closure' c"
using timpl_closure'_timpls_trancl_subset' timpl_closure'_timpls_trancl_supset'
by blast

lemma timpl_closure'_timpls_rtrancl_subset:
  "timpl_closure' (c*) ⊆ timpl_closure' c"
unfolding timpl_closure'_def
proof
  fix s t::"(('a,'b,'c) prot_fun,'d) term"
  show "(s,t) ∈ (timpl_closure'_step (c*))* ⟹ (s,t) ∈ (timpl_closure'_step c)*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c*" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(u,t) ∈ (timpl_closure'_step c)*"
    proof (induction arbitrary: t rule: rtrancl_induct)
      case base
      hence "u = t" using term_variants_pred_refl_inv by fastforce
      thus ?case by simp
    next
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      have "(u,s) ∈ (timpl_closure'_step c)*"
           "(s,t) ∈ timpl_closure'_step c"
        using step.hyps(2) s(2) step.IH[OF s(1)]
        by (auto intro: timpl_closure'_step.intros)
      thus ?case by simp
    qed
    thus ?case using step.IH by simp
  qed simp
qed

lemma timpl_closure'_timpls_rtrancl_supset:
  "timpl_closure' c ⊆ timpl_closure' (c*)"
unfolding timpl_closure'_def
proof
  fix s t::"(('e,'f,'c) prot_fun,'g) term"
  show "(s,t) ∈ (timpl_closure'_step c)* ⟹ (s,t) ∈ (timpl_closure'_step (c*))*"
  proof (induction rule: rtrancl_induct)
    case (step u t)
    obtain a b where ab:
        "(a,b) ∈ c" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u t"
      using step.hyps(2) timpl_closure'_step_inv by blast
    hence "(a,b) ∈ c*" by simp
    hence "(u,t) ∈ (timpl_closure'_step (c*))*" using ab(2)
    proof (induction arbitrary: t rule: rtrancl_induct)
      case (base t) thus ?case using term_variants_pred_refl_inv[of _ u t] by fastforce
    next
      case (step d e)
      obtain s where s:
          "term_variants_pred ((λ_. [])(Abs a := [Abs d])) u s"
          "term_variants_pred ((λ_. [])(Abs d := [Abs e])) s t"
        using term_variants_pred_dense'[OF step.prems, of "Abs d"] by blast

      show ?case
      proof (cases "d = e")
        case True
```

```
        thus ?thesis
          using step.prems step.IH[of t]
          by blast
      next
        case False
        hence "(u,s) ∈ (timpl_closure'_step (c*))*"
              "(s,t) ∈ timpl_closure'_step (c*)"
          using step.hyps(2) s(2) step.IH[OF s(1)]
          by (auto intro: timpl_closure'_step.intros)
        thus ?thesis by simp
      qed
    qed
    thus ?case using step.IH by simp
  qed simp
qed

lemma timpl_closure'_timpls_rtrancl_eq:
  "timpl_closure' (c*) = timpl_closure' c"
using timpl_closure'_timpls_rtrancl_subset timpl_closure'_timpls_rtrancl_supset
by blast

lemma timpl_closure_timpls_trancl_eq:
  "timpl_closure t (c+) = timpl_closure t c"
using timpl_closure'_timpls_trancl_eq[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "c+"]
by fastforce

lemma timpl_closure_set_timpls_trancl_eq:
  "timpl_closure_set M (c+) = timpl_closure_set M c"
using timpl_closure_timpls_trancl_eq
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "c+"]
by fastforce

lemma timpl_closure_set_timpls_trancl_eq':
  "timpl_closure_set M {(a,b) ∈ c+. a ≠ b} = timpl_closure_set M c"
using timpl_closure'_timpls_trancl_eq'[of c]
      timpl_closure_is_timpl_closure'[of _ _ c]
      timpl_closure_is_timpl_closure'[of _ _ "{(a,b) ∈ c+. a ≠ b}"]
      timpl_closure_set_is_timpl_closure_union[of M c]
      timpl_closure_set_is_timpl_closure_union[of M "{(a,b) ∈ c+. a ≠ b}"]
by fastforce

lemma timpl_closure_Var_in_iff:
  "Var x ∈ timpl_closure t TI ⟷ t = Var x" (is "?A ⟷ ?B")
proof
  have "s ∈ timpl_closure t TI ⟹ s = Var x ⟹ s = t" for s
    apply (induction rule: timpl_closure.induct)
    by (simp, metis term_variants_pred_inv_Var(2))
  thus "?A ⟹ ?B" by blast
qed (blast intro: timpl_closure.FP)

lemma timpl_closure_set_Var_in_iff:
  "Var x ∈ timpl_closure_set M TI ⟷ Var x ∈ M"
unfolding timpl_closure_set_def by (simp add: timpl_closure_Var_in_iff[of x _ TI])

lemma timpl_closure_Var_inv:
  assumes "t ∈ timpl_closure (Var x) TI"
  shows "t = Var x"
using assms
proof (induction rule: timpl_closure.induct)
  case (TI u a b s) thus ?case using term_variants_pred_inv_Var by fast
```

```
qed simp

lemma timpls_Un_mono: "mono (λX. FP ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --≫ b⟩⟨x⟩))"
by (auto intro!: monoI)

lemma timpl_closure_set_lfp:
  fixes M TI
  defines "f ≡ λX. M ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --≫ b⟩⟨x⟩)"
  shows "lfp f = timpl_closure_set M TI"
proof
  note 0 = timpls_Un_mono[of M TI, unfolded f_def[symmetric]]

  let ?N = "timpl_closure_set M TI"

  show "lfp f ⊆ ?N"
  proof (induction rule: lfp_induct)
    case 2 thus ?case
    proof
      fix t assume "t ∈ f (lfp f ∩ ?N)"
      hence "t ∈ M ∨ t ∈ (⋃x ∈ ?N. ⋃(a,b) ∈ TI. set ⟨a --≫ b⟩⟨x⟩)" (is "?A ∨ ?B")
        unfolding f_def by blast
      thus "t ∈ ?N"
      proof
        assume ?B
        then obtain s a b where s: "s ∈ ?N" "(a,b) ∈ TI" "t ∈ set ⟨a --≫ b⟩⟨s⟩" by moura
        thus ?thesis
          using term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])" s]
          unfolding timpl_closure_set_def timpl_apply_term_def
          by (auto intro: timpl_closure.intros)
      qed (auto simp add: timpl_closure_set_def intro: timpl_closure.intros)
    qed
  qed (rule 0)

  have "t ∈ lfp f" when t: "t ∈ timpl_closure s TI" and s: "s ∈ M" for t s
    using t
  proof (induction t rule: timpl_closure.induct)
    case (TI u a b v) thus ?case
      using term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
            lfp_fixpoint[OF 0]
      unfolding timpl_apply_term_def f_def by fastforce
  qed (use s lfp_fixpoint[OF 0] f_def in blast)
  thus "?N ⊆ lfp f" unfolding timpl_closure_set_def by blast
qed

lemma timpl_closure_set_supset:
  assumes "∀t ∈ FP. t ∈ closure"
  and "∀t ∈ closure. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --≫ b⟩⟨t⟩. s ∈ closure"
  shows "timpl_closure_set FP TI ⊆ closure"
proof -
  have "t ∈ closure" when t: "t ∈ timpl_closure s TI" and s: "s ∈ FP" for t s
    using t
  proof (induction rule: timpl_closure.induct)
    case FP thus ?case using s assms(1) by blast
  next
    case (TI u a b s') thus ?case
      using assms(2) term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs a := [Abs b])"]
      unfolding timpl_apply_term_def by fastforce
  qed
  thus ?thesis unfolding timpl_closure_set_def by blast
qed

lemma timpl_closure_set_supset':
  assumes "∀t ∈ FP. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --≫ b⟩⟨t⟩. s ∈ FP"
```

```
  shows "timpl_closure_set FP TI ⊆ FP"
using timpl_closure_set_supset[OF _ assms] by blast

lemma timpl_closure'_param:
  assumes "(t,s) ∈ timpl_closure' c"
    and fg: "f = g ∨ (∃a b. (a,b) ∈ c ∧ f = Abs a ∧ g = Abs b)"
  shows "(Fun f (S@t#T), Fun g (S@s#T)) ∈ timpl_closure' c"
using assms(1) unfolding timpl_closure'_def
proof (induction rule: rtrancl_induct)
  case base thus ?case
  proof (cases "f = g")
    case False
    then obtain a b where ab: "(a,b) ∈ c" "f = Abs a" "g = Abs b"
      using fg by moura
    show ?thesis
      using term_variants_pred_param[OF term_variants_pred_refl[of "(λ_. [])(Abs a := [Abs b])" t]]
            timpl_closure'_step.intros[OF ab(1)] ab(2,3)
      by fastforce
  qed simp
next
  case (step u s)
  obtain a b where ab: "(a,b) ∈ c" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u s"
    using timpl_closure'_step_inv[OF step.hyps(2)] by blast
  have "(Fun g (S@u#T), Fun g (S@s#T)) ∈ timpl_closure'_step c"
      using ab(1) term_variants_pred_param[OF ab(2), of g g S T]
      by (auto simp add: timpl_closure'_def intro: timpl_closure'_step.intros)
  thus ?case using rtrancl_into_rtrancl[OF step.IH] fg by blast
qed

lemma timpl_closure'_param':
  assumes "(t,s) ∈ timpl_closure' c"
  shows "(Fun f (S@t#T), Fun f (S@s#T)) ∈ timpl_closure' c"
using timpl_closure'_param[OF assms] by simp

lemma timpl_closure_FunI:
  assumes IH: "⋀i. i < length T ⟹ (T ! i, S ! i) ∈ timpl_closure' c"
    and len: "length T = length S"
    and fg: "f = g ∨ (∃a b. (a,b) ∈ c⁺ ∧ f = Abs a ∧ g = Abs b)"
  shows "(Fun f T, Fun g S) ∈ timpl_closure' c"
proof -
  have aux: "(Fun f T, Fun g (take n S@drop n T)) ∈ timpl_closure' c"
    when "n ≤ length T" for n
    using that
  proof (induction n)
    case 0
    have "(T ! n, T ! n) ∈ timpl_closure' c" when n: "n < length T" for n
      using n unfolding timpl_closure'_def by simp
    hence "(Fun f T, Fun g T) ∈ timpl_closure' c"
    proof (cases "f = g")
      case False
      then obtain a b where ab: "(a, b) ∈ c⁺" "f = Abs a" "g = Abs b"
        using fg by moura
      show ?thesis
        using timpl_closure'_step.intros[OF ab(1), of "Fun f T" "Fun g T"] ab(2,3)
              term_variants_P[OF _ term_variants_pred_refl[of "(λ_. [])(Abs a := [Abs b])"],
                              of T g f]
              timpl_closure'_timpls_trancl_eq
        unfolding timpl_closure'_def
        by (metis fun_upd_same list.set_intros(1) r_into_rtrancl)
    qed (simp add: timpl_closure'_def)
    thus ?case by simp
  next
    case (Suc n)
```

```
    hence IH': "(Fun f T, Fun g (take n S@drop n T)) ∈ timpl_closure' c"
      and n: "n < length T" "n < length S"
      by (simp_all add: len)

    obtain T1 T2 where T: "T = T1@T ! n#T2" "length T1 = n"
      using length_prefix_ex'[OF n(1)] by auto

    obtain S1 S2 where S: "S = S1@S ! n#S2" "length S1 = n"
      using length_prefix_ex'[OF n(2)] by auto

    have "take n S@drop n T = S1@T ! n#T2" "take (Suc n) S@drop (Suc n) T = S1@S ! n#T2"
      using n T S append_eq_conv_conj
      by (metis, metis (no_types, hide_lams) Cons_nth_drop_Suc append.assoc append_Cons
                                             append_Nil take_Suc_conv_app_nth)
    moreover have "(T ! n, S ! n) ∈ timpl_closure' c" using IH Suc.prems by simp
    ultimately show ?case
      using timpl_closure'_param IH'(1)
      by (metis (no_types, lifting) timpl_closure'_def rtrancl_trans)
  qed

  show ?thesis using aux[of "length T"] len by simp
qed

lemma timpl_closure_FunI':
  assumes IH: "⋀i. i < length T ⟹ (T ! i, S ! i) ∈ timpl_closure' c"
    and len: "length T = length S"
  shows "(Fun f T, Fun f S) ∈ timpl_closure' c"
using timpl_closure_FunI[OF IH len] by simp

lemma timpl_closure_FunI2:
  fixes f g::"('a, 'b, 'c) prot_fun"
  assumes IH: "⋀i. i < length T ⟹ ∃u. (T!i, u) ∈ timpl_closure' c ∧ (S!i, u) ∈ timpl_closure' c"
    and len: "length T = length S"
    and fg: "f = g ∨ (∃a b d. (a, d) ∈ c⁺ ∧ (b, d) ∈ c⁺ ∧ f = Abs a ∧ g = Abs b)"
  shows "∃h U. (Fun f T, Fun h U) ∈ timpl_closure' c ∧ (Fun g S, Fun h U) ∈ timpl_closure' c"
proof -
  let ?P = "λi u. (T ! i, u) ∈ timpl_closure' c ∧ (S ! i, u) ∈ timpl_closure' c"

  define U where "U ≡ map (λi. SOME u. ?P i u) [0..<length T]"

  have U1: "length U = length T"
    unfolding U_def by auto

  have U2: "(T ! i, U ! i) ∈ timpl_closure' c ∧ (S ! i, U ! i) ∈ timpl_closure' c"
    when i: "i < length U" for i
    using i someI_ex[of "?P i"] IH[of i] U1 len
    unfolding U_def by simp

  show ?thesis
  proof (cases "f = g")
    case False
    then obtain a b d where abd: "(a, d) ∈ c⁺" "(b, d) ∈ c⁺" "f = Abs a" "g = Abs b"
      using fg by moura

    define h::"('a, 'b, 'c) prot_fun" where "h = Abs d"

    have "f = h ∨ (∃a b. (a, b) ∈ c⁺ ∧ f = Abs a ∧ h = Abs b)"
         "g = h ∨ (∃a b. (a, b) ∈ c⁺ ∧ g = Abs a ∧ h = Abs b)"
      using abd unfolding h_def by blast+
    thus ?thesis by (metis timpl_closure_FunI len U1 U2)
  qed (metis timpl_closure_FunI' len U1 U2)
qed
```

```
lemma timpl_closure_FunI3:
  fixes f g::"('a, 'b, 'c) prot_fun"
  assumes IH: "⋀i. i < length T ⟹ ∃u. (T!i, u) ∈ timpl_closure' c ∧ (S!i, u) ∈ timpl_closure' c"
    and len: "length T = length S"
    and fg: "f = g ∨ (∃a b d. (a, d) ∈ c ∧ (b, d) ∈ c ∧ f = Abs a ∧ g = Abs b)"
  shows "∃h U. (Fun f T, Fun h U) ∈ timpl_closure' c ∧ (Fun g S, Fun h U) ∈ timpl_closure' c"
using timpl_closure_FunI2[OF IH len] fg unfolding timpl_closure'_timpls_trancl_eq by blast

lemma timpl_closure_fv_eq:
  assumes "s ∈ timpl_closure t T"
  shows "fv s = fv t"
using assms
by (induct rule: timpl_closure.induct)
   (metis, metis term_variants_pred_fv_eq)

lemma (in stateful_protocol_model) timpl_closure_subst:
  assumes t: "wf_trm t" "∀x ∈ fv t. ∃a. Γ_v x = TAtom (Atom a)"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)"
  shows "timpl_closure (t · δ) T = timpl_closure t T ·_set δ"
proof
  have "s ∈ timpl_closure t T ·_set δ"
    when "s ∈ timpl_closure (t · δ) T" for s
    using that
  proof (induction s rule: timpl_closure.induct)
    case FP thus ?case using timpl_closure.FP[of t T] by simp
  next
    case (TI u a b s)
    then obtain u' where u': "u' ∈ timpl_closure t T" "u = u' · δ" by moura

    have u'_fv: "∀x ∈ fv u'. ∃a. Γ_v x = TAtom (Atom a)"
      using timpl_closure_fv_eq[OF u'(1)] t(2) by simp
    hence u_fv: "∀x ∈ fv u. ∃a. Γ_v x = TAtom (Atom a)"
      using u'(2) wt_subst_trm''[OF δ(1)] wt_subst_const_fv_type_eq[OF _ δ(1,2), of u']
      by fastforce

    have "∀x ∈ fv u' ∪ fv s. (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ Abs a ≠ f)"
    proof (intro ballI)
      fix x assume x: "x ∈ fv u' ∪ fv s"
      then obtain c where c: "Γ_v x = TAtom (Atom c)"
        using u'_fv u_fv term_variants_pred_fv_eq[OF TI.hyps(3)]
        by blast

      show "(∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ Abs a ≠ f)"
      proof (cases "δ x")
        case (Fun f T)
        hence **: "Γ (Fun f T) = TAtom (Atom c)" and "wf_trm (Fun f T)"
          using c wt_subst_trm''[OF δ(1), of "Var x"] δ(2)
          by fastforce+
        hence "δ x = Fun f []" using Fun const_type_inv_wf by metis
        thus ?thesis using ** by force
      qed metis
    qed
    hence *: "∀x ∈ fv u' ∪ fv s.
              (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ ((λ_. [])(Abs a := [Abs b])) f = [])"
      by fastforce

    obtain s' where s': "term_variants_pred ((λ_. [])(Abs a := [Abs b])) u' s'" "s = s' · δ"
      using term_variants_pred_subst'[OF _ *] u'(2) TI.hyps(3)
      by blast

    show ?case using timpl_closure.TI[OF u'(1) TI.hyps(2) s'(1)] s'(2) by blast
  qed
  thus "timpl_closure (t · δ) T ⊆ timpl_closure t T ·_set δ" by fast
```

```
  have "s ∈ timpl_closure (t · δ) T"
    when s: "s ∈ timpl_closure t T ·set δ" for s
  proof -
    obtain s' where s': "s' ∈ timpl_closure t T" "s = s' · δ" using s by moura
    have "s' · δ ∈ timpl_closure (t · δ) T" using s'(1)
    proof (induction s' rule: timpl_closure.induct)
      case FP thus ?case using timpl_closure.FP[of "t · δ" T] by simp
    next
      case (TI u' a b s') show ?case
        using timpl_closure.TI[OF TI.IH TI.hyps(2)]
              term_variants_pred_subst[OF TI.hyps(3)]
        by blast
    qed
    thus ?thesis using s'(2) by metis
  qed
  thus "timpl_closure t T ·set δ ⊆ timpl_closure (t · δ) T" by fast
qed

lemma (in stateful_protocol_model) timpl_closure_subst_subset:
  assumes t: "t ∈ M"
    and M: "wftrms M" "∀x ∈ fvset M. ∃a. Γv x = TAtom (Atom a)"
    and δ: "wtsubst δ" "wftrms (subst_range δ)" "ground (subst_range δ)" "subst_domain δ ⊆ fvset M"
    and M_supset: "timpl_closure t T ⊆ M"
  shows "timpl_closure (t · δ) T ⊆ M ·set δ"
proof -
  have t': "wftrm t" "∀x ∈ fv t. ∃a. Γv x = TAtom (Atom a)" using t M by auto
  show ?thesis using timpl_closure_subst[OF t' δ(1,2), of T] M_supset by blast
qed

lemma (in stateful_protocol_model) timpl_closure_set_subst_subset:
  assumes M: "wftrms M" "∀x ∈ fvset M. ∃a. Γv x = TAtom (Atom a)"
    and δ: "wtsubst δ" "wftrms (subst_range δ)" "ground (subst_range δ)" "subst_domain δ ⊆ fvset M"
    and M_supset: "timpl_closure_set M T ⊆ M"
  shows "timpl_closure_set (M ·set δ) T ⊆ M ·set δ"
using timpl_closure_subst_subset[OF _ M δ, of _ T] M_supset
      timpl_closure_set_is_timpl_closure_union[of "M ·set δ" T]
      timpl_closure_set_is_timpl_closure_union[of M T]
by auto

lemma timpl_closure_set_Union:
  "timpl_closure_set (⋃Ms) T = (⋃M ∈ Ms. timpl_closure_set M T)"
using timpl_closure_set_is_timpl_closure_union[of "⋃Ms" T]
      timpl_closure_set_is_timpl_closure_union[of _ T]
by force

lemma timpl_closure_set_Union_subst_set:
  assumes "s ∈ timpl_closure_set (⋃{M ·set δ | δ. P δ}) T"
  shows "∃δ. P δ ∧ s ∈ timpl_closure_set (M ·set δ) T"
using assms timpl_closure_set_is_timpl_closure_union[of "(⋃{M ·set δ | δ. P δ})" T]
      timpl_closure_set_is_timpl_closure_union[of _ T]
by blast

lemma timpl_closure_set_Union_subst_singleton:
  assumes "s ∈ timpl_closure_set {t · δ | δ. P δ} T"
  shows "∃δ. P δ ∧ s ∈ timpl_closure_set {t · δ} T"
using assms timpl_closure_set_is_timpl_closure_union[of "{t · δ |δ. P δ}" T]
      timpl_closureton_is_timpl_closure[of _ T]
by fast

lemma timpl_closure'_inv:
  assumes "(s, t) ∈ timpl_closure' TI"
  shows "(∃x. s = Var x ∧ t = Var x) ∨ (∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T)"
```

```
using assms unfolding timpl_closure'_def
proof (induction rule: rtrancl_induct)
  case base thus ?case by (cases s) auto
next
  case (step t u)
  obtain a b where ab: "(a, b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t u"
    using timpl_closure'_step_inv[OF step.hyps(2)] by blast
  show ?case using step.IH
  proof
    assume "∃x. s = Var x ∧ t = Var x"
    thus ?case using step.hyps(2) term_variants_pred_inv_Var ab by fastforce
  next
    assume "∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T"
    then obtain f g S T where st: "s = Fun f S" "t = Fun g T" "length S = length T" by moura
    thus ?case
      using ab step.hyps(2) term_variants_pred_inv'[of "(λ_. [])(Abs a := [Abs b])" g T u]
      by auto
  qed
qed

lemma timpl_closure'_inv':
  assumes "(s, t) ∈ timpl_closure' TI"
  shows "(∃x. s = Var x ∧ t = Var x) ∨
         (∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T ∧
                    (∀i < length T. (S ! i, T ! i) ∈ timpl_closure' TI) ∧
                    (f ≠ g ⟶ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI⁺))"
    (is "?A s t ∨ ?B s t (timpl_closure' TI)")
using assms unfolding timpl_closure'_def
proof (induction rule: rtrancl_induct)
  case base thus ?case by (cases s) auto
next
  case (step t u)
  obtain a b where ab: "(a, b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t u"
    using timpl_closure'_step_inv[OF step.hyps(2)] by blast
  show ?case using step.IH
  proof
    assume "?A s t"
    thus ?case using step.hyps(2) term_variants_pred_inv_Var ab by fastforce
  next
    assume "?B s t ((timpl_closure'_step TI)*)"
    then obtain f g S T where st:
        "s = Fun f S" "t = Fun g T" "length S = length T"
        "⋀i. i < length T ⟹ (S ! i, T ! i) ∈ (timpl_closure'_step TI)*"
        "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI⁺"
      by moura
    obtain h U where u:
        "u = Fun h U" "length T = length U"
        "⋀i. i < length T ⟹ term_variants_pred ((λ_. [])(Abs a := [Abs b])) (T ! i) (U ! i)"
        "g ≠ h ⟹ is_Abs g ∧ is_Abs h ∧ (the_Abs g, the_Abs h) ∈ TI⁺"
      using ab(2) st(2) r_into_trancl[OF ab(1)]
            term_variants_pred_inv'(1,2,3,4)[of "(λ_. [])(Abs a := [Abs b])" g T u]
            term_variants_pred_inv'(5)[of "(λ_. [])(Abs a := [Abs b])" g T u "Abs a" "Abs b"]
      unfolding is_Abs_def the_Abs_def by force

    have "(S ! i, U ! i) ∈ (timpl_closure'_step TI)*" when i: "i < length U" for i
      using u(2) i rtrancl.rtrancl_into_rtrancl[OF
              st(4)[of i] timpl_closure'_step.intros[OF ab(1) u(3)[of i]]]
      by argo
    moreover have "length S = length U" using st u by argo
    moreover have "is_Abs f ∧ is_Abs h ∧ (the_Abs f, the_Abs h) ∈ TI⁺" when fh: "f ≠ h"
      using fh st u by fastforce
    ultimately show ?case using st(1) u(1) by blast
  qed
```

qed

lemma *timpl_closure'_inv'':*
  assumes *"(Fun f S, Fun g T) ∈ timpl_closure' TI"*
  shows *"length S = length T"*
    and *"⋀i. i < length T ⟹ (S ! i, T ! i) ∈ timpl_closure' TI"*
    and *"f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI⁺"*
using *assms timpl_closure'_inv'* **by** *auto*

lemma *timpl_closure_Fun_inv:*
  assumes *"s ∈ timpl_closure (Fun f T) TI"*
  shows *"∃g S. s = Fun g S"*
using *assms timpl_closure_is_timpl_closure' timpl_closure'_inv*
by *fastforce*

lemma *timpl_closure_Fun_inv':*
  assumes *"Fun g S ∈ timpl_closure (Fun f T) TI"*
  shows *"length S = length T"*
    and *"⋀i. i < length S ⟹ S ! i ∈ timpl_closure (T ! i) TI"*
    and *"f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI⁺"*
using *assms timpl_closure_is_timpl_closure'*
by *(metis timpl_closure'_inv''(1), metis timpl_closure'_inv''(2), metis timpl_closure'_inv''(3))*

lemma *timpl_closure_Fun_not_Var[simp]:*
  "Fun f T ∉ timpl_closure (Var x) TI"
using *timpl_closure_Var_inv* **by** *fast*

lemma *timpl_closure_Var_not_Fun[simp]:*
  "Var x ∉ timpl_closure (Fun f T) TI"
using *timpl_closure_Fun_inv* **by** *fast*

lemma (**in** *stateful_protocol_model*) *timpl_closure_wf_trms:*
  assumes *m: "wf$_{trm}$ m"*
  shows *"wf$_{trms}$ (timpl_closure m TI)"*
proof
  fix *t* **assume** *"t ∈ timpl_closure m TI"*
  thus *"wf$_{trm}$ t"*
  proof *(induction t rule: timpl_closure.induct)*
    case *TI* **thus** *?case* **using** *term_variants_pred_wf_trms* **by** *force*
  qed *(rule m)*
qed

lemma (**in** *stateful_protocol_model*) *timpl_closure_set_wf_trms:*
  assumes *M: "wf$_{trms}$ M"*
  shows *"wf$_{trms}$ (timpl_closure_set M TI)"*
proof
  fix *t* **assume** *"t ∈ timpl_closure_set M TI"*
  then obtain *m* **where** *"t ∈ timpl_closure m TI" "m ∈ M" "wf$_{trm}$ m"*
    using *M timpl_closure_set_is_timpl_closure_union* **by** *blast*
  thus *"wf$_{trm}$ t"* **using** *timpl_closure_wf_trms* **by** *blast*
qed

lemma *timpl_closure_Fu_inv:*
  assumes *"t ∈ timpl_closure (Fun (Fu f) T) TI"*
  shows *"∃S. length S = length T ∧ t = Fun (Fu f) S"*
using *assms*
proof *(induction t rule: timpl_closure.induct)*
  case *(TI u a b s)*
  then obtain *U* **where** *U: "length U = length T" "u = Fun (Fu f) U"*
    by *moura*
  hence *∗: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) (Fun (Fu f) U) s"*
    using *TI.hyps(3)* **by** *meson*

```
  show ?case
    using term_variants_pred_inv'(1,2,4)[OF *] U
    by force
qed simp

lemma timpl_closure_Fu_inv':
  assumes "Fun (Fu f) T ∈ timpl_closure t TI"
  shows "∃S. length S = length T ∧ t = Fun (Fu f) S"
using assms
proof (induction "Fun (Fu f) T" arbitrary: T rule: timpl_closure.induct)
  case (TI u a b)
  obtain g U where U:
      "u = Fun g U" "length U = length T"
      "Fu f ≠ g ⟹ Abs a = g ∧ Fu f = Abs b"
    using term_variants_pred_inv''[OF TI.hyps(4)] by fastforce

  have g: "g = Fu f" using U(3) by blast

  show ?case using TI.hyps(2)[OF U(1)[unfolded g]] U(2) by auto
qed simp

lemma timpl_closure_no_Abs_eq:
  assumes "t ∈ timpl_closure s TI"
    and "∀f ∈ funs_term t. ¬is_Abs f"
  shows "t = s"
using assms
proof (induction t rule: timpl_closure.induct)
  case (TI t a b s) thus ?case
    using term_variants_pred_eq_case_Abs[of a b t s]
    unfolding timpl_apply_term_def term_variants_pred_iff_in_term_variants[symmetric]
    by metis
qed simp

lemma timpl_closure_set_no_Abs_in_set:
  assumes "t ∈ timpl_closure_set FP TI"
    and "∀f ∈ funs_term t. ¬is_Abs f"
  shows "t ∈ FP"
using assms timpl_closure_no_Abs_eq unfolding timpl_closure_set_def by blast

lemma timpl_closure_funs_term_subset:
  "⋃(funs_term ` (timpl_closure t TI)) ⊆ funs_term t ∪ Abs ` snd ` TI"
  (is "?A ⊆ ?B ∪ ?C")
proof
  fix f assume "f ∈ ?A"
  then obtain s where "s ∈ timpl_closure t TI" "f ∈ funs_term s" by moura
  thus "f ∈ ?B ∪ ?C"
  proof (induction s rule: timpl_closure.induct)
    case (TI u a b s)
    have "Abs b ∈ Abs ` snd ` TI" using TI.hyps(2) by force
    thus ?case using term_variants_pred_funs_term[OF TI.hyps(3) TI.prems] TI.IH by force
  qed blast
qed

lemma timpl_closure_set_funs_term_subset:
  "⋃(funs_term ` (timpl_closure_set FP TI)) ⊆ ⋃(funs_term ` FP) ∪ Abs ` snd ` TI"
using timpl_closure_funs_term_subset[of _ TI]
      timpl_closure_set_is_timpl_closure_union[of FP TI]
by auto

lemma funs_term_OCC_TI_subset:
  defines "absc ≡ λa. Fun (Abs a) []"
  assumes OCC1: "∀t ∈ FP. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` OCC"
    and OCC2: "snd ` TI ⊆ OCC"
```

```
  shows "∀t ∈ timpl_closure_set FP TI. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ OCC" (is ?A)
    and "∀t ∈ absc ‘ OCC. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --≫ b⟩⟨t⟩. s ∈ absc ‘ OCC" (is ?B)
proof -
  let ?F = "⋃(funs_term ‘ FP)"
  let ?G = "Abs ‘ snd ‘ TI"

  show ?A
  proof (intro ballI impI)
    fix t f assume t: "t ∈ timpl_closure_set FP TI" and f: "f ∈ funs_term t" "is_Abs f"
    hence "f ∈ ?F ∨ f ∈ ?G" using timpl_closure_set_funs_term_subset[of FP TI] by auto
    thus "f ∈ Abs ‘ OCC"
    proof
      assume "f ∈ ?F" thus ?thesis using OCC1 f(2) by fast
    next
      assume "f ∈ ?G" thus ?thesis using OCC2 by auto
    qed
  qed

  { fix s t a b
    assume t: "t ∈ absc ‘ OCC"
      and ab: "(a, b) ∈ TI"
      and s: "s ∈ set ⟨a --≫ b⟩⟨t⟩"
    obtain c where c: "t = absc c" "c ∈ OCC" using t by moura
    hence "s = absc b ∨ s = absc c"
      using ab s timpl_apply_const'[of c a b] unfolding absc_def by auto
    moreover have "b ∈ OCC" using ab OCC2 by auto
    ultimately have "s ∈ absc ‘ OCC" using c(2) by blast
  } thus ?B by blast
qed

lemma (in stateful_protocol_model) intruder_synth_timpl_closure_set:
  fixes M::"('fun,'atom,'sets) prot_terms" and t::"('fun,'atom,'sets) prot_term"
  assumes "M ⊢c t"
    and "s ∈ timpl_closure t TI"
  shows "timpl_closure_set M TI ⊢c s"
using assms
proof (induction t arbitrary: s rule: intruder_synth_induct)
  case (AxiomC t)
  hence "s ∈ timpl_closure_set M TI"
    using timpl_closure_set_is_timpl_closure_union[of M TI]
    by blast
  thus ?case by simp
next
  case (ComposeC T f)
  obtain g S where s: "s = Fun g S"
    using timpl_closure_Fun_inv[OF ComposeC.prems] by moura
  hence s':
      "f = g" "length S = length T"
      "⋀i. i < length S ⟹ S ! i ∈ timpl_closure (T ! i) TI"
    using timpl_closure_Fun_inv'[of g S f T TI] ComposeC.prems ComposeC.hyps(2)
    unfolding is_Abs_def by fastforce+

  have "timpl_closure_set M TI ⊢c u" when u: "u ∈ set S" for u
    using ComposeC.IH u s'(2,3) in_set_conv_nth[of _ T] in_set_conv_nth[of u S] by auto
  thus ?case
    using s s'(1,2) ComposeC.hyps(1,2) intruder_synth.ComposeC[of S g "timpl_closure_set M TI"]
    by argo
qed

lemma (in stateful_protocol_model) intruder_synth_timpl_closure':
  fixes M::"('fun,'atom,'sets) prot_terms" and t::"('fun,'atom,'sets) prot_term"
  assumes "timpl_closure_set M TI ⊢c t"
    and "s ∈ timpl_closure t TI"
```

```
  shows "timpl_closure_set M TI ⊢c s"
by (metis intruder_synth_timpl_closure_set[OF assms] timpl_closure_set_idem)

lemma timpl_closure_set_absc_subset_in:
  defines "absc ≡ λa. Fun (Abs a) []"
  assumes A: "timpl_closure_set (absc ‘ A) TI ⊆ absc ‘ A"
    and a: "a ∈ A" "(a,b) ∈ TI⁺"
  shows "b ∈ A"
proof -
  have "timpl_closure (absc a) (TI⁺) ⊆ absc ‘ A"
    using a(1) A timpl_closure_timpls_trancl_eq
    unfolding timpl_closure_set_def by fast
  thus ?thesis
    using timpl_closure.TI[OF timpl_closure.FP[of "absc a"] a(2), of "absc b"]
          term_variants_P[of "[]" "[]" "(λ_. [])(Abs a := [Abs b])" "Abs b" "Abs a"]
    unfolding absc_def by auto
qed
```

2.5.3 Composition-only Intruder Deduction Modulo Term Implication Closure of the Intruder Knowledge

```
context stateful_protocol_model
begin

fun in_trancl where
  "in_trancl TI a b = (
    if (a,b) ∈ set TI then True
    else list_ex (λ(c,d). c = a ∧ in_trancl (removeAll (c,d) TI) d b) TI)"

definition in_rtrancl where
  "in_rtrancl TI a b ≡ a = b ∨ in_trancl TI a b"

declare in_trancl.simps[simp del]

fun timpls_transformable_to where
  "timpls_transformable_to TI (Var x) (Var y) = (x = y)"
| "timpls_transformable_to TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)) ∧
    list_all2 (timpls_transformable_to TI) T S)"
| "timpls_transformable_to _ _ _ = False"

fun timpls_transformable_to' where
  "timpls_transformable_to' TI (Var x) (Var y) = (x = y)"
| "timpls_transformable_to' TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))) ∧
    list_all2 (timpls_transformable_to' TI) T S)"
| "timpls_transformable_to' _ _ _ = False"

fun equal_mod_timpls where
  "equal_mod_timpls TI (Var x) (Var y) = (x = y)"
| "equal_mod_timpls TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧
                ((the_Abs f, the_Abs g) ∈ set TI ∨
                 (the_Abs g, the_Abs f) ∈ set TI ∨
                 (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧ (the_Abs g, snd ti) ∈ set TI)))) ∧
    list_all2 (equal_mod_timpls TI) T S)"
| "equal_mod_timpls _ _ _ = False"

fun intruder_synth_mod_timpls where
  "intruder_synth_mod_timpls M TI (Var x) = List.member M (Var x)"
| "intruder_synth_mod_timpls M TI (Fun f T) = (
    (list_ex (λt. timpls_transformable_to TI t (Fun f T)) M) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_timpls M TI) T))"
```

```
fun intruder_synth_mod_timpls' where
  "intruder_synth_mod_timpls' M TI (Var x) = List.member M (Var x)"
| "intruder_synth_mod_timpls' M TI (Fun f T) = (
    (list_ex (λt. timpls_transformable_to' TI t (Fun f T)) M) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_timpls' M TI) T))"

fun intruder_synth_mod_eq_timpls where
  "intruder_synth_mod_eq_timpls M TI (Var x) = (Var x ∈ M)"
| "intruder_synth_mod_eq_timpls M TI (Fun f T) = (
    (∃t ∈ M. equal_mod_timpls TI t (Fun f T)) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_eq_timpls M TI) T))"

definition analyzed_closed_mod_timpls where
  "analyzed_closed_mod_timpls M TI ≡
    let f = list_all (intruder_synth_mod_timpls M TI);
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

definition analyzed_closed_mod_timpls' where
  "analyzed_closed_mod_timpls' M TI ≡
    let f = list_all (intruder_synth_mod_timpls' M TI);
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

definition analyzed_closed_mod_timpls_alt where
  "analyzed_closed_mod_timpls_alt M TI timpl_cl_witness ≡
    let f = λR. ∀r ∈ set R. intruder_synth_mod_timpls M TI r;
        N = {t ∈ set M. f (fst (Ana t))};
        N' = set M - N
    in (∀t ∈ N. f (snd (Ana t))) ∧
       (N' ≠ {} ⟶ (N' ∪ (⋃x∈timpl_cl_witness. ⋃(a,b)∈set TI. set ⟨a --≫ b⟩⟨x⟩) ⊆ timpl_cl_witness))
∧
       (∀s ∈ timpl_cl_witness. case Ana s of (K,R) ⇒ f K ⟶ f R)"

lemma in_trancl_closure_iff_in_trancl_fun:
  "(a,b) ∈ (set TI)⁺ ⟷ in_trancl TI a b" (is "?A TI a b ⟷ ?B TI a b")
proof
  show "?A TI a b ⟹ ?B TI a b"
  proof (induction rule: trancl_induct)
    case (step c d)
    show ?case using step.IH step.hyps(2)
    proof (induction TI a c rule: in_trancl.induct)
      case (1 TI a b) thus ?case using in_trancl.simps
        by (smt Bex_set case_prodE case_prodI member_remove prod.sel(2) remove_code(1))
    qed
  qed (metis in_trancl.simps)

  show "?B TI a b ⟹ ?A TI a b"
  proof (induction TI a b rule: in_trancl.induct)
    case (1 TI a b)
    let ?P = "λTI a b c d. in_trancl (List.removeAll (c,d) TI) d b"
    have *: "∃(c,d) ∈ set TI. c = a ∧ ?P TI a b c d" when "(a,b) ∉ set TI"
      using that "1.prems" list_ex_iff[of _ TI] in_trancl.simps[of TI a b]
      by auto
    show ?case
    proof (cases "(a,b) ∈ set TI")
      case False
      hence "∃(c,d) ∈ set TI. c = a ∧ ?P TI a b c d" using * by blast
      then obtain d where d: "(a,d) ∈ set TI" "?P TI a b a d" by blast
      have "(d,b) ∈ (set (removeAll (a,d) TI))⁺" using "1.IH"[OF False d(1)] d(2) by blast
```

```
      moreover have "set (removeAll (a,d) TI) ⊆ set TI" by simp
      ultimately have "(d,b) ∈ (set TI)⁺" using trancl_mono by blast
      thus ?thesis using d(1) by fastforce
    qed simp
  qed
qed

lemma in_rtrancl_closure_iff_in_rtrancl_fun:
  "(a,b) ∈ (set TI)* ⟷ in_rtrancl TI a b"
by (metis rtrancl_eq_or_trancl in_trancl_closure_iff_in_trancl_fun in_rtrancl_def)

lemma in_trancl_mono:
  assumes "set TI ⊆ set TI'"
    and "in_trancl TI a b"
  shows "in_trancl TI' a b"
by (metis assms in_trancl_closure_iff_in_trancl_fun trancl_mono)

lemma equal_mod_timpls_refl:
  "equal_mod_timpls TI t t"
proof (induction t)
  case (Fun f T) thus ?case
    using list_all2_conv_all_nth[of "equal_mod_timpls TI" T T] by force
qed simp

lemma equal_mod_timpls_inv_Var:
  "equal_mod_timpls TI (Var x) t ⟹ t = Var x" (is "?A ⟹ ?C")
  "equal_mod_timpls TI t (Var x) ⟹ t = Var x" (is "?B ⟹ ?C")
proof -
  show "?A ⟹ ?C" by (cases t) auto
  show "?B ⟹ ?C" by (cases t) auto
qed

lemma equal_mod_timpls_inv:
  assumes "equal_mod_timpls TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ equal_mod_timpls TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ (
                    (the_Abs f, the_Abs g) ∈ set TI ∨ (the_Abs g, the_Abs f) ∈ set TI ∨
                    (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                                   (the_Abs g, snd ti) ∈ set TI)))"
using assms list_all2_conv_all_nth[of "equal_mod_timpls TI" T S]
by (auto elim: equal_mod_timpls.cases)

lemma equal_mod_timpls_inv':
  assumes "equal_mod_timpls TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ equal_mod_timpls TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (is_Abs f ∧ is_Abs (the_Fun t) ∧ (
                    (the_Abs f, the_Abs (the_Fun t)) ∈ set TI ∨
                    (the_Abs (the_Fun t), the_Abs f) ∈ set TI ∨
                    (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                                   (the_Abs (the_Fun t), snd ti) ∈ set TI)))"
    and "¬is_Abs f ⟹ f = the_Fun t"
using assms list_all2_conv_all_nth[of "equal_mod_timpls TI" T]
by (cases t; auto)+

lemma equal_mod_timpls_if_term_variants:
  fixes s t::"('a, 'b, 'c) prot_fun, 'd) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ set TI"
  shows "equal_mod_timpls TI s t"
```

```
using st P_def
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S f) thus ?case
    using ab list_all2_conv_all_nth[of "equal_mod_timpls TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
next
  case (term_variants_Fun T S f) thus ?case
    using ab list_all2_conv_all_nth[of "equal_mod_timpls TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
qed simp

lemma equal_mod_timpls_mono:
  assumes "set TI ⊆ set TI'"
    and "equal_mod_timpls TI s t"
  shows "equal_mod_timpls TI' s t"
  using assms
proof (induction TI s t rule: equal_mod_timpls.induct)
  case (2 TI f T g S)
  have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ ((the_Abs f, the_Abs g) ∈ set TI ∨
                (the_Abs g, the_Abs f) ∈ set TI ∨
                (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                               (the_Abs g, snd ti) ∈ set TI)))"
         "list_all2 (equal_mod_timpls TI) T S"
    using "2.prems" by simp_all

  show ?case
    using "2.IH" "2.prems"(1) list.rel_mono_strong[OF *(2)] *(1) in_trancl_mono[of TI TI']
    by (metis (no_types, lifting) equal_mod_timpls.simps(2) set_rev_mp)
qed auto

lemma equal_mod_timpls_refl_minus_eq:
  "equal_mod_timpls TI s t ⟷ equal_mod_timpls (filter (λ(a,b). a ≠ b) TI) s t"
  (is "?A ⟷ ?B")
proof
  show ?A when ?B using that equal_mod_timpls_mono[of "filter (λ(a,b). a ≠ b) TI" TI] by auto

  show ?B when ?A using that
  proof (induction TI s t rule: equal_mod_timpls.induct)
    case (2 TI f T g S)
    define TI' where "TI' ≡ filter (λ(a,b). a ≠ b) TI"

    let ?P = "λX Y. f = g ∨ (is_Abs f ∧ is_Abs g ∧ ((the_Abs f, the_Abs g) ∈ set X ∨
                (the_Abs g, the_Abs f) ∈ set X ∨ (∃ti ∈ set Y.
                (the_Abs f, snd ti) ∈ set X ∧ (the_Abs g, snd ti) ∈ set X)))"

    have *: "?P TI TI" "list_all2 (equal_mod_timpls TI) T S"
      using "2.prems" by simp_all

    have "?P TI' TI"
      using *(1) unfolding TI'_def is_Abs_def by auto
    hence "?P TI' TI'"
      by (metis (no_types, lifting) snd_conv)
    moreover have "list_all2 (equal_mod_timpls TI') T S"
      using *(2) "2.IH" list.rel_mono_strong unfolding TI'_def by blast
    ultimately show ?case unfolding TI'_def by force
  qed auto
qed

lemma timpls_transformable_to_refl:
  "timpls_transformable_to TI t t" (is ?A)
  "timpls_transformable_to' TI t t" (is ?B)
```

```
by (induct t) (auto simp add: list_all2_conv_all_nth)

lemma timpls_transformable_to_inv_Var:
  "timpls_transformable_to TI (Var x) t ⟹ t = Var x" (is "?A ⟹ ?C")
  "timpls_transformable_to TI t (Var x) ⟹ t = Var x" (is "?B ⟹ ?C")
  "timpls_transformable_to' TI (Var x) t ⟹ t = Var x" (is "?A' ⟹ ?C")
  "timpls_transformable_to' TI t (Var x) ⟹ t = Var x" (is "?B' ⟹ ?C")
by (cases t; auto)+

lemma timpls_transformable_to_inv:
  assumes "timpls_transformable_to TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
using assms list_all2_conv_all_nth[of "timpls_transformable_to TI" T S] by auto

lemma timpls_transformable_to'_inv:
  assumes "timpls_transformable_to' TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))"
using assms list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S] by auto

lemma timpls_transformable_to_inv':
  assumes "timpls_transformable_to TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (
          is_Abs f ∧ is_Abs (the_Fun t) ∧ (the_Abs f, the_Abs (the_Fun t)) ∈ set TI)"
    and "¬is_Abs f ⟹ f = the_Fun t"
using assms list_all2_conv_all_nth[of "timpls_transformable_to TI" T]
by (cases t; auto)+

lemma timpls_transformable_to'_inv':
  assumes "timpls_transformable_to' TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (
          is_Abs f ∧ is_Abs (the_Fun t) ∧ in_trancl TI (the_Abs f) (the_Abs (the_Fun t)))"
    and "¬is_Abs f ⟹ f = the_Fun t"
using assms list_all2_conv_all_nth[of "timpls_transformable_to' TI" T]
by (cases t; auto)+

lemma timpls_transformable_to_size_eq:
  fixes s t::"(('b, 'c, 'a) prot_fun, 'd) term"
  shows "timpls_transformable_to TI s t ⟹ size s = size t" (is "?A ⟹ ?C")
    and "timpls_transformable_to' TI s t ⟹ size s = size t" (is "?B ⟹ ?C")
proof -
  have *: "size_list size T = size_list size S"
    when "length T = length S" "⋀i. i < length T ⟹ size (T ! i) = size (S ! i)"
    for S T::"(('b, 'c, 'a) prot_fun, 'd) term list"
    using that
  proof (induction T arbitrary: S)
    case (Cons x T')
    then obtain y S' where y: "S = y#S'" by (cases S) auto
    hence "size_list size T' = size_list size S'" "size x = size y"
      using Cons.prems Cons.IH[of S'] by force+
    thus ?case using y by simp
  qed simp

  show ?C when ?A using that
```

```
  proof (induction rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    hence "length T = length S" "⋀i. i < length T ⟹ size (T ! i) = size (S ! i)"
      using timpls_transformable_to_inv(1,2)[of TI f T g S] by auto
    thus ?case using *[of S T] by simp
  qed simp_all

  show ?C when ?B using that
  proof (induction rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    hence "length T = length S" "⋀i. i < length T ⟹ size (T ! i) = size (S ! i)"
      using timpls_transformable_to'_inv(1,2)[of TI f T g S] by auto
    thus ?case using *[of S T] by simp
  qed simp_all
qed

lemma timpls_transformable_to_if_term_variants:
  fixes s t::"(('a, 'b, 'c) prot_fun, 'd) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ set TI"
  shows "timpls_transformable_to TI s t"
using st P_def
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to TI" T S]
    by auto
next
  case (term_variants_Fun T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to TI" T S]
    by auto
qed simp

lemma timpls_transformable_to'_if_term_variants:
  fixes s t::"(('a, 'b, 'c) prot_fun, 'd) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ (set TI)⁺"
  shows "timpls_transformable_to' TI s t"
using st P_def
proof (induction rule: term_variants_pred.induct)
  case (term_variants_P T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
next
  case (term_variants_Fun T S f) thus ?case
    using ab list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S]
          in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    by auto
qed simp

lemma timpls_transformable_to_trans:
  assumes TI_trancl: "∀(a,b) ∈ (set TI)⁺. a ≠ b ⟶ (a,b) ∈ set TI"
    and st: "timpls_transformable_to TI s t"
    and tu: "timpls_transformable_to TI t u"
  shows "timpls_transformable_to TI s u"
using st tu
proof (induction s arbitrary: t u)
  case (Var x) thus ?case using tu timpls_transformable_to_inv_Var(1) by fast
next
  case (Fun f T)
  obtain g S where t:
```

```
      "t = Fun g S" "length T = length S"
      "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (S ! i)"
      "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI"
    using timpls_transformable_to_inv'[OF Fun.prems(1)] TI_trancl by moura

  obtain h U where u:
      "u = Fun h U" "length S = length U"
      "⋀i. i < length S ⟹ timpls_transformable_to TI (S ! i) (U ! i)"
      "g ≠ h ⟹ is_Abs g ∧ is_Abs h ∧ (the_Abs g, the_Abs h) ∈ set TI"
    using timpls_transformable_to_inv'[OF Fun.prems(2)[unfolded t(1)]] TI_trancl by moura

  have "list_all2 (timpls_transformable_to TI) T U"
    using t(1,2,3) u(1,2,3) Fun.IH
          list_all2_conv_all_nth[of "timpls_transformable_to TI" T S]
          list_all2_conv_all_nth[of "timpls_transformable_to TI" S U]
          list_all2_conv_all_nth[of "timpls_transformable_to TI" T U]
    by force
  moreover have "(the_Abs f, the_Abs h) ∈ set TI"
    when "(the_Abs f, the_Abs g) ∈ set TI" "(the_Abs g, the_Abs h) ∈ set TI"
         "f ≠ h" "is_Abs f" "is_Abs h"
    using that(3,4,5) TI_trancl trancl_into_trancl[OF r_into_trancl[OF that(1)] that(2)]
    unfolding is_Abs_def the_Abs_def
    by force
  hence "is_Abs f ∧ is_Abs h ∧ (the_Abs f, the_Abs h) ∈ set TI"
    when "f ≠ h"
    using that TI_trancl t(4) u(4) by fast
  ultimately show ?case using t(1) u(1) by force
qed

lemma timpls_transformable_to'_trans:
  assumes st: "timpls_transformable_to' TI s t"
    and tu: "timpls_transformable_to' TI t u"
  shows "timpls_transformable_to' TI s u"
using st tu
proof (induction s arbitrary: t u)
  case (Var x) thus ?case using tu timpls_transformable_to_inv_Var(3) by fast
next
  case (Fun f T)
  note 0 = in_trancl_closure_iff_in_trancl_fun[of _ _ TI]

  obtain g S where t:
      "t = Fun g S" "length T = length S"
      "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (S ! i)"
      "f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ (set TI)⁺"
    using timpls_transformable_to'_inv'[OF Fun.prems(1)] 0 by moura

  obtain h U where u:
      "u = Fun h U" "length S = length U"
      "⋀i. i < length S ⟹ timpls_transformable_to' TI (S ! i) (U ! i)"
      "g ≠ h ⟹ is_Abs g ∧ is_Abs h ∧ (the_Abs g, the_Abs h) ∈ (set TI)⁺"
    using timpls_transformable_to'_inv'[OF Fun.prems(2)[unfolded t(1)]] 0 by moura

  have "list_all2 (timpls_transformable_to' TI) T U"
    using t(1,2,3) u(1,2,3) Fun.IH
          list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S]
          list_all2_conv_all_nth[of "timpls_transformable_to' TI" S U]
          list_all2_conv_all_nth[of "timpls_transformable_to' TI" T U]
    by force
  moreover have "(the_Abs f, the_Abs h) ∈ (set TI)⁺"
    when "(the_Abs f, the_Abs g) ∈ (set TI)⁺" "(the_Abs g, the_Abs h) ∈ (set TI)⁺"
    using that by simp
  hence "is_Abs f ∧ is_Abs h ∧ (the_Abs f, the_Abs h) ∈ (set TI)⁺"
    when "f ≠ h"
```

```
    by (metis that t(4) u(4))
  ultimately show ?case using t(1) u(1) 0 by force
qed

lemma timpls_transformable_to_mono:
  assumes "set TI ⊆ set TI'"
    and "timpls_transformable_to TI s t"
  shows "timpls_transformable_to TI' s t"
  using assms
proof (induction TI s t rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
          "list_all2 (timpls_transformable_to TI) T S"
    using "2.prems" by simp_all

  show ?case
    using "2.IH" "2.prems"(1) list.rel_mono_strong[OF *(2)] *(1) in_trancl_mono[of TI TI']
    by (metis (no_types, lifting) timpls_transformable_to.simps(2) set_rev_mp)
qed auto

lemma timpls_transformable_to'_mono:
  assumes "set TI ⊆ set TI'"
    and "timpls_transformable_to' TI s t"
  shows "timpls_transformable_to' TI' s t"
  using assms
proof (induction TI s t rule: timpls_transformable_to'.induct)
  case (2 TI f T g S)
  have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))"
          "list_all2 (timpls_transformable_to' TI) T S"
    using "2.prems" by simp_all

  show ?case
    using "2.IH" "2.prems"(1) list.rel_mono_strong[OF *(2)] *(1) in_trancl_mono[of TI TI']
    by (metis (no_types, lifting) timpls_transformable_to'.simps(2))
qed auto

lemma timpls_transformable_to_refl_minus_eq:
  "timpls_transformable_to TI s t ⟷ timpls_transformable_to (filter (λ(a,b). a ≠ b) TI) s t"
  (is "?A ⟷ ?B")
proof
  let ?TI' = "λTI. filter (λ(a,b). a ≠ b) TI"

  show ?A when ?B using that timpls_transformable_to_mono[of "?TI' TI" TI] by auto

  show ?B when ?A using that
  proof (induction TI s t rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    have *: "f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
            "list_all2 (timpls_transformable_to TI) T S"
      using "2.prems" by simp_all

    have "f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set (?TI' TI))"
      using *(1) unfolding is_Abs_def by auto
    moreover have "list_all2 (timpls_transformable_to (?TI' TI)) T S"
      using *(2) "2.IH" list.rel_mono_strong by blast
    ultimately show ?case by force
  qed auto
qed

lemma timpls_transformable_to_iff_in_timpl_closure:
  assumes "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "timpls_transformable_to TI' s t ⟷ t ∈ timpl_closure s (set TI)" (is "?A s t ⟷ ?B s t")
proof
```

```
  show "?A s t ⟹ ?B s t" using assms
  proof (induction s t rule: timpls_transformable_to.induct)
    case (2 TI f T g S)
    note prems = "2.prems"
    note IH = "2.IH"

    have 1: "length T = length S" "∀i<length T. timpls_transformable_to TI' (T ! i) (S ! i)"
      using prems(1) list_all2_conv_all_nth[of "timpls_transformable_to TI'" T S] by simp_all

    note 2 = timpl_closure_is_timpl_closure'
    note 3 = in_set_conv_nth[of _ T] in_set_conv_nth[of _ S]

    have 4: "timpl_closure' (set TI') = timpl_closure' (set TI)"
      using timpl_closure'_timpls_trancl_eq'[of "set TI"] prems(2) by simp

    have IH': "(T ! i, S ! i) ∈ timpl_closure' (set TI')" when i: "i < length S" for i
    proof -
      have "timpls_transformable_to TI' (T ! i) (S ! i)" using i 1 by presburger
      hence "S ! i ∈ timpl_closure (T ! i) (set TI)"
        using IH[of "T ! i" "S ! i"] i 1(1) prems(2) by force
      thus ?thesis using 2[of "S ! i" "T ! i" "set TI"] 4 by blast
    qed

    have 5: "f = g ∨ (∃a b. (a, b) ∈ (set TI')⁺ ∧ f = Abs a ∧ g = Abs b)"
      using prems(1) the_Abs_def[of f] the_Abs_def[of g] is_Abs_def[of f] is_Abs_def[of g]
      by fastforce

    show ?case using 2 4 timpl_closure_FunI[OF IH' 1(1) 5] 1(1) by auto
  qed (simp_all add: timpl_closure.FP)

  show "?B s t ⟹ ?A s t"
  proof (induction t rule: timpl_closure.induct)
    case (TI u a b v) show ?case
    proof (cases "a = b")
      case True thus ?thesis using TI.hyps(3) TI.IH term_variants_pred_refl_inv by fastforce
    next
      case False
      hence 1: "timpls_transformable_to TI' u v"
        using TI.hyps(2) assms timpls_transformable_to_if_term_variants[OF TI.hyps(3), of TI']
        by blast
      have 2: "(c,d) ∈ set TI'" when cd: "(c,d) ∈ (set TI')⁺" "c ≠ d" for c d
      proof -
        let ?cl = "λX. {(a,b) ∈ X⁺. a ≠ b}"
        have "?cl (set TI') = ?cl (?cl (set TI))" using assms by presburger
        hence "set TI' = ?cl (set TI')" using assms trancl_minus_refl_idem[of "set TI"] by argo
        thus ?thesis using cd by blast
      qed
      show ?thesis using timpls_transformable_to_trans[OF _ TI.IH 1] 2 by blast
    qed
  qed (use timpls_transformable_to_refl in fast)
qed

lemma timpls_transformable_to'_iff_in_timpl_closure:
  "timpls_transformable_to' TI s t ⟷ t ∈ timpl_closure s (set TI)" (is "?A s t ⟷ ?B s t")
proof
  show "?A s t ⟹ ?B s t"
  proof (induction s t rule: timpls_transformable_to'.induct)
    case (2 TI f T g S)
    note prems = "2.prems"
    note IH = "2.IH"

    have 1: "length T = length S" "∀i<length T. timpls_transformable_to' TI (T ! i) (S ! i)"
      using prems list_all2_conv_all_nth[of "timpls_transformable_to' TI" T S] by simp_all
```

```
    note 2 = timpl_closure_is_timpl_closure'
    note 3 = in_set_conv_nth[of _ T] in_set_conv_nth[of _ S]

    have IH': "(T ! i, S ! i) ∈ timpl_closure' (set TI)" when i: "i < length S" for i
    proof -
      have "timpls_transformable_to' TI (T ! i) (S ! i)" using i 1 by presburger
      hence "S ! i ∈ timpl_closure (T ! i) (set TI)" using IH[of "T ! i" "S ! i"] i 1(1) by force
      thus ?thesis using 2[of "S ! i" "T ! i" "set TI"] by blast
    qed

    have 4: "f = g ∨ (∃a b. (a, b) ∈ (set TI)⁺ ∧ f = Abs a ∧ g = Abs b)"
      using prems the_Abs_def[of f] the_Abs_def[of g] is_Abs_def[of f] is_Abs_def[of g]
            in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      by auto

    show ?case using 2 timpl_closure_FunI[OF IH' 1(1) 4] 1(1) by auto
  qed (simp_all add: timpl_closure.FP)

  show "?B s t ⟹ ?A s t"
  proof (induction t rule: timpl_closure.induct)
    case (TI u a b v) thus ?case
      using timpls_transformable_to'_trans
            timpls_transformable_to'_if_term_variants
      by blast
  qed (use timpls_transformable_to_refl(2) in fast)
qed

lemma equal_mod_timpls_iff_ex_in_timpl_closure:
  assumes "set TI' = {(a,b) ∈ TI⁺. a ≠ b}"
  shows "equal_mod_timpls TI' s t ⟷ (∃u. u ∈ timpl_closure s TI ∧ u ∈ timpl_closure t TI)"
    (is "?A s t ⟷ ?B s t")
proof
  show "?A s t ⟹ ?B s t" using assms
  proof (induction s t rule: equal_mod_timpls.induct)
    case (2 TI' f T g S)
    note prems = "2.prems"
    note IH = "2.IH"

    have 1: "length T = length S" "∀i<length T. equal_mod_timpls (TI') (T ! i) (S ! i)"
      using prems list_all2_conv_all_nth[of "equal_mod_timpls TI'" T S] by simp_all

    note 2 = timpl_closure_is_timpl_closure'
    note 3 = in_set_conv_nth[of _ T] in_set_conv_nth[of _ S]

    have 4: "timpl_closure' (set TI') = timpl_closure' TI"
      using timpl_closure'_timpls_trancl_eq'[of TI] prems
      by simp

    have IH: "∃u. (T ! i, u) ∈ timpl_closure' TI ∧ (S ! i, u) ∈ timpl_closure' TI"
      when i: "i < length S" for i
    proof -
      have "equal_mod_timpls TI' (T ! i) (S ! i)" using i 1 by presburger
      hence "∃u. u ∈ timpl_closure (T ! i) TI ∧ u ∈ timpl_closure (S ! i) TI"
        using IH[of "T ! i" "S ! i"] i 1(1) prems by force
      thus ?thesis using 4 unfolding 2 by blast
    qed

    let ?P = "λG. f = g ∨ (∃a b. (a, b) ∈ G ∧ f = Abs a ∧ g = Abs b) ∨
                  (∃a b. (a, b) ∈ G ∧ f = Abs b ∧ g = Abs a) ∨
                  (∃a b c. (a, c) ∈ G ∧ (b, c) ∈ G ∧ f = Abs a ∧ g = Abs b)"

    have "?P (set TI')"
```

```
    using prems the_Abs_def[of f] the_Abs_def[of g] is_Abs_def[of f] is_Abs_def[of g]
    by fastforce
  hence "?P (TI⁺)" unfolding prems by blast
  hence "?P (rtrancl TI)" by (metis (no_types, lifting) trancl_into_rtrancl)
  hence 5: "f = g ∨ (∃a b c. (a, c) ∈ TI* ∧ (b, c) ∈ TI* ∧ f = Abs a ∧ g = Abs b)" by blast

  show ?case
    using timpl_closure_FunI3[OF _ 1(1) 5]  IH 1(1)
    unfolding timpl_closure'_timpls_rtrancl_eq 2
    by auto
qed (use timpl_closure.FP in auto)

show "?A s t" when B: "?B s t"
proof -
  obtain u where u: "u ∈ timpl_closure s TI" "u ∈ timpl_closure t TI"
    using B by moura
  thus ?thesis using assms
  proof (induction u arbitrary: s t rule: term.induct)
    case (Var x s t) thus ?case
      using timpl_closure_Var_in_iff[of x s TI]
            timpl_closure_Var_in_iff[of x t TI]
            equal_mod_timpls.simps(1)[of TI' x x]
      by blast
  next
    case (Fun f U s t)
    obtain g S where s:
        "s = Fun g S" "length U = length S"
        "⋀i. i < length U ⟹ U ! i ∈ timpl_closure (S ! i) TI"
        "g ≠ f ⟹ is_Abs g ∧ is_Abs f ∧ (the_Abs g, the_Abs f) ∈ TI⁺"
      using Fun.prems(1) timpl_closure_Fun_inv'[of f U _ _ TI]
      by (cases s) auto

    obtain h T where t:
        "t = Fun h T" "length U = length T"
        "⋀i. i < length U ⟹ U ! i ∈ timpl_closure (T ! i) TI"
        "h ≠ f ⟹ is_Abs h ∧ is_Abs f ∧ (the_Abs h, the_Abs f) ∈ TI⁺"
      using Fun.prems(2) timpl_closure_Fun_inv'[of f U _ _ TI]
      by (cases t) auto

    have g: "(the_Abs g, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs g" when neq_f: "g ≠ f"
    proof -
      obtain ga fa where a: "g = Abs ga" "f = Abs fa"
        using s(4)[OF neq_f] unfolding is_Abs_def by presburger
      hence "the_Abs g ≠ the_Abs f" using neq_f by simp
      thus "(the_Abs g, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs g"
        using s(4)[OF neq_f] Fun.prems by blast+
    qed

    have h: "(the_Abs h, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs h" when neq_f: "h ≠ f"
    proof -
      obtain ha fa where a: "h = Abs ha" "f = Abs fa"
        using t(4)[OF neq_f] unfolding is_Abs_def by presburger
      hence "the_Abs h ≠ the_Abs f" using neq_f by simp
      thus "(the_Abs h, the_Abs f) ∈ set TI'" "is_Abs f" "is_Abs h"
        using t(4)[OF neq_f] Fun.prems by blast+
    qed

    have "equal_mod_timpls TI' (S ! i) (T ! i)"
      when i: "i < length U" for i
      using i Fun.IH s(1,2,3) t(1,2,3) nth_mem[OF i] Fun.prems by meson
    hence "list_all2 (equal_mod_timpls TI') S T"
      using list_all2_conv_all_nth[of "equal_mod_timpls TI'" S T] s(2) t(2) by presburger
    thus ?case using s(1) t(1) g h by fastforce
```

```
    qed
  qed
qed



context
begin
private inductive timpls_transformable_to_pred where
  Var: "timpls_transformable_to_pred A (Var x) (Var x)"
| Fun: "⟦¬is_Abs f; length T = length S;
        ⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (S ! i)⟧
       ⟹ timpls_transformable_to_pred A (Fun f T) (Fun f S)"
| Abs: "b ∈ A ⟹ timpls_transformable_to_pred A (Fun (Abs a) []) (Fun (Abs b) [])"

private lemma timpls_transformable_to_pred_inv_Var:
  assumes "timpls_transformable_to_pred A (Var x) t"
  shows "t = Var x"
using assms by (auto elim: timpls_transformable_to_pred.cases)

private lemma timpls_transformable_to_pred_inv:
  assumes "timpls_transformable_to_pred A (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (args t ! i)"
    and "¬is_Abs f ⟹ f = the_Fun t"
    and "is_Abs f ⟹ (is_Abs (the_Fun t) ∧ the_Abs (the_Fun t) ∈ A)"
using assms by (auto elim!: timpls_transformable_to_pred.cases[of A])

private lemma timpls_transformable_to_pred_finite_aux1:
  assumes f: "¬is_Abs f"
  shows "{s. timpls_transformable_to_pred A (Fun f T) s} ⊆
         (λS. Fun f S) ` {S. length T = length S ∧
                             (∀s ∈ set S. ∃t ∈ set T. timpls_transformable_to_pred A t s)}"
    (is "?B ⊆ ?C")
proof
  fix s assume s: "s ∈ ?B"
  hence *: "timpls_transformable_to_pred A (Fun f T) s" by blast

  obtain S where S:
      "s = Fun f S" "length T = length S" "⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i)
(S ! i)"
    using f timpls_transformable_to_pred_inv[OF *] unfolding the_Abs_def is_Abs_def by auto

  have "∀s∈set S. ∃t∈set T. timpls_transformable_to_pred A t s" using S(2,3) in_set_conv_nth by metis
  thus "s ∈ ?C" using S(1,2) by blast
qed

private lemma timpls_transformable_to_pred_finite_aux2:
  "{s. timpls_transformable_to_pred A (Fun (Abs a) []) s} ⊆ (λb. Fun (Abs b) []) ` A" (is "?B ⊆ ?C")
proof
  fix s assume s: "s ∈ ?B"
  hence *: "timpls_transformable_to_pred A (Fun (Abs a) []) s" by blast

  obtain b where b: "s = Fun (Abs b) []" "b ∈ A"
    using timpls_transformable_to_pred_inv[OF *] unfolding the_Abs_def is_Abs_def by auto
  thus "s ∈ ?C" by blast
qed

private lemma timpls_transformable_to_pred_finite:
  fixes t::"(('fun,'atom,'sets) prot_fun, 'a) term"
  assumes A: "finite A"
    and t: "wf_trm t"
```

```
  shows "finite {s. timpls_transformable_to_pred A t s}"
using t
proof (induction t)
  case (Var x)
  have "{s::(('fun,'atom,'sets) prot_fun, 'a) term. timpls_transformable_to_pred A (Var x) s} = {Var x}"
    by (auto intro: timpls_transformable_to_pred.Var elim: timpls_transformable_to_pred_inv_Var)
  thus ?case by simp
next
  case (Fun f T)
  have IH: "finite {s. timpls_transformable_to_pred A t s}" when t: "t ∈ set T" for t
    using Fun.IH[OF t] wf_trm_param[OF Fun.prems t] by blast

  show ?case
  proof (cases "is_Abs f")
    case True
    then obtain a where a: "f = Abs a" unfolding is_Abs_def by presburger
    hence "T = []" using wf_trm_arity[OF Fun.prems] by simp_all
    hence "{a. timpls_transformable_to_pred A (Fun f T) a} ⊆ (λb. Fun (Abs b) []) ` A"
      using timpls_transformable_to_pred_finite_aux2[of A a] a by auto
    thus ?thesis using A finite_subset by fast
  next
    case False thus ?thesis
      using IH finite_lists_length_eq' timpls_transformable_to_pred_finite_aux1[of f A T] finite_subset
      by blast
  qed
qed

private lemma timpls_transformable_to_pred_if_timpls_transformable_to:
  assumes s: "timpls_transformable_to TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
using s t
proof (induction rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  let ?A = "A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  note prems = "2.prems"
  note IH = "2.IH"

  note 0 = timpls_transformable_to_inv[OF prems(1)]

  have 1: "T = []" "S = []" when f: "f = Abs a" for a
    using f wf_trm_arity[OF prems(2)] 0(1) by simp_all

  have "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A" when t: "t ∈ set T" for t
    using t prems(3) funs_term_subterms_eq(1)[of "Fun f T"] by blast
  hence 2: "timpls_transformable_to_pred ?A (T ! i) (S ! i)"
    when i: "i < length T" for i
    using i IH 0(1,2) wf_trm_param[OF prems(2)]
    by (metis (no_types) in_set_conv_nth)

  have 3: "the_Abs f ∈ ?A" when f: "is_Abs f" using prems(3) f by force

  show ?case
  proof (cases "f = g")
    case True
    note fg = True
    show ?thesis
    proof (cases "is_Abs f")
      case True
      then obtain a where a: "f = Abs a" unfolding is_Abs_def by moura
      thus ?thesis using fg 1[OF a] timpls_transformable_to_pred.Abs[of a ?A a] 3 by simp
    qed (use fg timpls_transformable_to_pred.Fun[OF _ 0(1) 2, of f] in blast)
```

```
  next
    case False
    then obtain a b where ab: "f = Abs a" "g = Abs b" "(a, b) ∈ (set TI)⁺"
      using 0(3) in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      unfolding is_Abs_def the_Abs_def by fastforce
    hence "a ∈ ?A" "b ∈ ?A" by force+
    thus ?thesis using timpls_transformable_to_pred.Abs ab(1,2) 1[OF ab(1)] by metis
  qed
qed (simp_all add: timpls_transformable_to_pred.Var)

private lemma timpls_transformable_to_pred_if_timpls_transformable_to':
  assumes s: "timpls_transformable_to' TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
using s t
proof (induction rule: timpls_transformable_to.induct)
  case (2 TI f T g S)
  let ?A = "A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  note prems = "2.prems"
  note IH = "2.IH"

  note 0 = timpls_transformable_to'_inv[OF prems(1)]

  have 1: "T = []" "S = []" when f: "f = Abs a" for a
    using f wf_trm_arity[OF prems(2)] 0(1) by simp_all

  have "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A" when t: "t ∈ set T" for t
    using t prems(3) funs_term_subterms_eq(1)[of "Fun f T"] by blast
  hence 2: "timpls_transformable_to_pred ?A (T ! i) (S ! i)"
    when i: "i < length T" for i
    using i IH 0(1,2) wf_trm_param[OF prems(2)]
    by (metis (no_types) in_set_conv_nth)

  have 3: "the_Abs f ∈ ?A" when f: "is_Abs f" using prems(3) f by force

  show ?case
  proof (cases "f = g")
    case True
    note fg = True
    show ?thesis
    proof (cases "is_Abs f")
      case True
      then obtain a where a: "f = Abs a" unfolding is_Abs_def by moura
      thus ?thesis using fg 1[OF a] timpls_transformable_to_pred.Abs[of a ?A a] 3 by simp
    qed (use fg timpls_transformable_to_pred.Fun[OF _ 0(1) 2, of f] in blast)
  next
    case False
    then obtain a b where ab: "f = Abs a" "g = Abs b" "(a, b) ∈ (set TI)⁺"
      using 0(3) in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      unfolding is_Abs_def the_Abs_def by fastforce
    hence "a ∈ ?A" "b ∈ ?A" by force+
    thus ?thesis using timpls_transformable_to_pred.Abs ab(1,2) 1[OF ab(1)] by metis
  qed
qed (simp_all add: timpls_transformable_to_pred.Var)

private lemma timpls_transformable_to_pred_if_equal_mod_timpls:
  assumes s: "equal_mod_timpls TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
using s t
proof (induction rule: equal_mod_timpls.induct)
  case (2 TI f T g S)
```

```
  let ?A = "A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  note prems = "2.prems"
  note IH = "2.IH"

  note 0 = equal_mod_timpls_inv[OF prems(1)]

  have 1: "T = []" "S = []" when f: "f = Abs a" for a
    using f wf_trm_arity[OF prems(2)] 0(1) by simp_all

  have "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A" when t: "t ∈ set T" for t
    using t prems(3) funs_term_subterms_eq(1)[of "Fun f T"] by blast
  hence 2: "timpls_transformable_to_pred ?A (T ! i) (S ! i)"
    when i: "i < length T" for i
    using i IH 0(1,2) wf_trm_param[OF prems(2)]
    by (metis (no_types) in_set_conv_nth)

  have 3: "the_Abs f ∈ ?A" when f: "is_Abs f" using prems(3) f by force

  show ?case
  proof (cases "f = g")
    case True
    note fg = True
    show ?thesis
    proof (cases "is_Abs f")
      case True
      then obtain a where a: "f = Abs a" unfolding is_Abs_def by moura
      thus ?thesis using fg 1[OF a] timpls_transformable_to_pred.Abs[of a ?A a] 3 by simp
    qed (use fg timpls_transformable_to_pred.Fun[OF _ 0(1) 2, of f] in blast)
  next
    case False
    then obtain a b where ab: "f = Abs a" "g = Abs b"
        "(a, b) ∈ (set TI)⁺ ∨ (b, a) ∈ (set TI)⁺ ∨
         (∃ti ∈ set TI. (a, snd ti) ∈ (set TI)⁺ ∧ (b, snd ti) ∈ (set TI)⁺)"
      using 0(3) in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
      unfolding is_Abs_def the_Abs_def by fastforce
    hence "a ∈ ?A" "b ∈ ?A" by force+
    thus ?thesis using timpls_transformable_to_pred.Abs ab(1,2) 1[OF ab(1)] by metis
  qed
qed (simp_all add: timpls_transformable_to_pred.Var)

lemma timpls_transformable_to_finite:
  assumes t: "wf_trm t"
  shows "finite {s. timpls_transformable_to TI t s}" (is ?P)
    and "finite {s. timpls_transformable_to' TI t s}" (is ?Q)
proof -
  let ?A = "the_Abs ` {f ∈ funs_term t. is_Abs f} ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  have 0: "finite ?A" by auto

  have 1: "{s. timpls_transformable_to TI t s} ⊆ {s. timpls_transformable_to_pred ?A t s}"
    using timpls_transformable_to_pred_if_timpls_transformable_to[OF _ t] by auto

  have 2: "{s. timpls_transformable_to' TI t s} ⊆ {s. timpls_transformable_to_pred ?A t s}"
    using timpls_transformable_to_pred_if_timpls_transformable_to'[OF _ t] by auto

  show ?P using timpls_transformable_to_pred_finite[OF 0 t] finite_subset[OF 1] by blast
  show ?Q using timpls_transformable_to_pred_finite[OF 0 t] finite_subset[OF 2] by blast
qed

lemma equal_mod_timpls_finite:
  assumes t: "wf_trm t"
  shows "finite {s. equal_mod_timpls TI t s}"
```

```
proof -
  let ?A = "the_Abs ` {f ∈ funs_term t. is_Abs f} ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺"

  have 0: "finite ?A" by auto

  have 1: "{s. equal_mod_timpls TI t s} ⊆ {s. timpls_transformable_to_pred ?A t s}"
    using timpls_transformable_to_pred_if_equal_mod_timpls[OF _ t] by auto

  show ?thesis using timpls_transformable_to_pred_finite[OF 0 t] finite_subset[OF 1] by blast
qed

end

lemma intruder_synth_mod_timpls_is_synth_timpl_closure_set:
  fixes t::"(('fun, 'atom, 'sets) prot_fun, 'a) term" and TI TI'
  assumes "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "intruder_synth_mod_timpls M TI' t ⟷ timpl_closure_set (set M) (set TI) ⊢c t"
      (is "?C t ⟷ ?D t")
proof -
  have *: "(∃m ∈ M. timpls_transformable_to TI' m t) ⟷ t ∈ timpl_closure_set M (set TI)"
    when "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    for M TI TI' and t::"(('fun, 'atom, 'sets) prot_fun, 'a) term"
    using timpls_transformable_to_iff_in_timpl_closure[OF that]
          timpl_closure_set_is_timpl_closure_union[of M "set TI"]
          timpl_closure_set_timpls_trancl_eq[of M "set TI"]
          timpl_closure_set_timpls_trancl_eq'[of M "set TI"]
    by auto

  show "?C t ⟷ ?D t"
  proof
    show "?C t ⟹ ?D t" using assms
    proof (induction t arbitrary: M TI TI' rule: intruder_synth_mod_timpls.induct)
      case (1 M TI' x)
      hence "Var x ∈ timpl_closure_set (set M) (set TI)"
        using timpl_closure.FP member_def unfolding timpl_closure_set_def by force
      thus ?case by simp
    next
      case (2 M TI f T)
      show ?case
      proof (cases "∃m ∈ set M. timpls_transformable_to TI' m (Fun f T)")
        case True thus ?thesis
          using "2.prems" *[of TI' TI "set M" "Fun f T"]
                intruder_synth.AxiomC[of "Fun f T" "timpl_closure_set (set M) (set TI)"]
          by blast
      next
        case False
        hence "¬(list_ex (λt. timpls_transformable_to TI' t (Fun f T)) M)"
          unfolding list_ex_iff by blast
        hence "public f" "length T = arity f" "list_all (intruder_synth_mod_timpls M TI') T"
          using "2.prems"(1) by force+
        thus ?thesis using "2.IH"[OF _ _ "2.prems"(2)] unfolding list_all_iff by force
      qed
    qed

    show "?D t ⟹ ?C t"
    proof (induction t rule: intruder_synth_induct)
      case (AxiomC t) thus ?case
        using timpl_closure_set_Var_in_iff[of _ "set M" "set TI"] *[OF assms, of "set M" t]
        by (cases t rule: term.exhaust) (force simp add: member_def list_ex_iff)+
    next
      case (ComposeC T f) thus ?case
        using list_all_iff[of "intruder_synth_mod_timpls M TI'" T]
              intruder_synth_mod_timpls.simps(2)[of M TI' f T]
```

```
      by blast
    qed
  qed
qed

lemma intruder_synth_mod_timpls'_is_synth_timpl_closure_set:
  fixes t::"('fun, 'atom, 'sets) prot_fun, 'a) term" and TI
  shows "intruder_synth_mod_timpls' M TI t ⟷ timpl_closure_set (set M) (set TI) ⊢c t"
      (is "?A t ⟷ ?B t")
proof -
  have *: "(∃m ∈ M. timpls_transformable_to' TI m t) ⟷ t ∈ timpl_closure_set M (set TI)"
    for M TI and t::"('fun, 'atom, 'sets) prot_fun, 'a) term"
    using timpls_transformable_to'_iff_in_timpl_closure[of TI _ t]
          timpl_closure_set_is_timpl_closure_union[of M "set TI"]
    by blast+

  show "?A t ⟷ ?B t"
  proof
    show "?A t ⟹ ?B t"
    proof (induction t arbitrary: M TI rule: intruder_synth_mod_timpls'.induct)
      case (1 M TI x)
      hence "Var x ∈ timpl_closure_set (set M) (set TI)"
        using timpl_closure.FP List.member_def[of M] unfolding timpl_closure_set_def by auto
      thus ?case by simp
    next
      case (2 M TI f T)
      show ?case
      proof (cases "∃m ∈ set M. timpls_transformable_to' TI m (Fun f T)")
        case True thus ?thesis
          using "2.prems" *[of "set M" TI "Fun f T"]
                intruder_synth.AxiomC[of "Fun f T" "timpl_closure_set (set M) (set TI)"]
          by blast
      next
        case False
        hence "public f" "length T = arity f" "list_all (intruder_synth_mod_timpls' M TI) T"
          using "2.prems" list_ex_iff[of _ M] by force+
        thus ?thesis
          using "2.IH"[of _ M TI] list_all_iff[of "intruder_synth_mod_timpls' M TI" T]
          by force
      qed
    qed

    show "?B t ⟹ ?A t"
    proof (induction t rule: intruder_synth_induct)
      case (AxiomC t) thus ?case
        using AxiomC timpl_closure_set_Var_in_iff[of _ "set M" "set TI"] *[of "set M" TI t]
              list_ex_iff[of _ M] List.member_def[of M]
        by (cases t rule: term.exhaust) force+
    next
      case (ComposeC T f) thus ?case
        using list_all_iff[of "intruder_synth_mod_timpls' M TI" T]
              intruder_synth_mod_timpls'.simps(2)[of M TI f T]
        by blast
    qed
  qed
qed

lemma intruder_synth_mod_eq_timpls_is_synth_timpl_closure_set:
  fixes t::"('fun, 'atom, 'sets) prot_fun, 'a) term" and TI
  defines "cl ≡ λTI. {(a,b) ∈ TI⁺. a ≠ b}"
  shows  "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b} ⟹
        intruder_synth_mod_eq_timpls M TI' t ⟷
        (∃s ∈ timpl_closure t (set TI). timpl_closure_set M (set TI) ⊢c s)"
```

```
      (is "?Q TI TI' ⟹ ?C t ⟷ ?D t")
proof -


  have **: "(∃m ∈ M. equal_mod_timpls TI' m t) ⟷
          (∃s ∈ timpl_closure t (set TI). s ∈ timpl_closure_set M (set TI))"
    when Q: "?Q TI TI'"
    for M TI TI' and t::"(('fun, 'atom, 'sets) prot_fun, 'a) term"
    using equal_mod_timpls_iff_ex_in_timpl_closure[OF Q]
          timpl_closure_set_is_timpl_closure_union[of M "set TI"]
          timpl_closure_set_timpls_trancl_eq'[of M "set TI"]
    by fastforce



  show "?C t ⟷ ?D t" when Q: "?Q TI TI'"
  proof
    show "?C t ⟹ ?D t" using Q
    proof (induction t arbitrary: M TI rule: intruder_synth_mod_eq_timpls.induct)
      case (1 M TI' x M TI)
      hence "Var x ∈ timpl_closure_set M (set TI)" "Var x ∈ timpl_closure (Var x) (set TI)"
        using timpl_closure.FP unfolding timpl_closure_set_def by auto
      thus ?case by force
    next
      case (2 M TI' f T M TI)
      show ?case
      proof (cases "∃m ∈ M. equal_mod_timpls TI' m (Fun f T)")
        case True thus ?thesis
          using **[OF "2.prems"(2), of M "Fun f T"]
                intruder_synth.AxiomC[of _ "timpl_closure_set M (set TI)"]
          by blast
      next
        case False
        hence f: "public f" "length T = arity f" "list_all (intruder_synth_mod_eq_timpls M TI') T"
          using "2.prems" by force+

        let ?sy = "intruder_synth (timpl_closure_set M (set TI))"

        have IH: "∃u ∈ timpl_closure (T ! i) (set TI). ?sy u"
          when i: "i < length T" for i
          using "2.IH"[of _ M TI] f(3) nth_mem[OF i] "2.prems"(2)
          unfolding list_all_iff by blast

        define S where "S ≡ map (λu. SOME v. v ∈ timpl_closure u (set TI) ∧ ?sy v) T"

        have S1: "length T = length S"
          unfolding S_def by simp

        have S2: "S ! i ∈ timpl_closure (T ! i) (set TI)"
                 "timpl_closure_set M (set TI) ⊢c S ! i"
          when i: "i < length S" for i
          using i IH someI_ex[of "λv. v ∈ timpl_closure (T ! i) (set TI) ∧ ?sy v"]
          unfolding S_def by auto

        have "Fun f S ∈ timpl_closure (Fun f T) (set TI)"
          using timpl_closure_FunI[of T S "set TI" f f] S1 S2(1)
          unfolding timpl_closure_is_timpl_closure' by presburger
        thus ?thesis
          by (metis intruder_synth.ComposeC[of S f] f(1,2) S1 S2(2) in_set_conv_nth[of _ S])
      qed
    qed

    show "?C t" when D: "?D t"
```

```
    proof -
      obtain s where "timpl_closure_set M (set TI) ⊢c s" "s ∈ timpl_closure t (set TI)"
        using D by moura
      thus ?thesis
      proof (induction s arbitrary: t rule: intruder_synth_induct)
        case (AxiomC s t)
        note 1 = timpl_closure_set_Var_in_iff[of _ M "set TI"] timpl_closure_Var_inv[of s _ "set TI"]
        note 2 = **[OF Q, of M]
        show ?case
        proof (cases t)
          case Var thus ?thesis using 1 AxiomC by auto
        next
          case Fun thus ?thesis using 2 AxiomC by auto
        qed
      next
        case (ComposeC T f t)
        obtain g S where gS:
            "t = Fun g S" "length S = length T"
            "∀i < length T. T ! i ∈ timpl_closure (S ! i) (set TI)"
            "g ≠ f ⟹ is_Abs g ∧ is_Abs f ∧ (the_Abs g, the_Abs f) ∈ (set TI)+"
          using ComposeC.prems(1) timpl_closure'_inv'[of t "Fun f T" "set TI"]
                timpl_closure_is_timpl_closure'[of _ _ "set TI"]
          by fastforce

        have IH: "intruder_synth_mod_eq_timpls M TI' u" when u: "u ∈ set S" for u
          by (metis u gS(2,3) ComposeC.IH in_set_conv_nth)

        note 0 = list_all_iff[of "intruder_synth_mod_eq_timpls M TI'" S]
                 intruder_synth_mod_eq_timpls.simps(2)[of M TI' g S]

        have "f = g" using ComposeC.hyps gS(4) unfolding is_Abs_def by fastforce
        thus ?case by (metis ComposeC.hyps(1,2) gS(1,2) IH 0)
      qed
    qed
  qed
qed

lemma timpl_closure_finite:
  assumes t: "wftrm t"
  shows "finite (timpl_closure t (set TI))"
using timpls_transformable_to'_iff_in_timpl_closure[of TI t]
      timpls_transformable_to_finite[OF t, of TI]
by auto

lemma timpl_closure_set_finite:
  fixes TI::"('sets set × 'sets set) list"
  assumes M_finite: "finite M"
    and M_wf: "wftrms M"
  shows "finite (timpl_closure_set M (set TI))"
using timpl_closure_set_is_timpl_closure_union[of M "set TI"]
      timpl_closure_finite[of _ TI] M_finite M_wf finite
by auto

lemma comp_timpl_closure_is_timpl_closure_set:
  fixes M and TI::"('sets set × 'sets set) list"
  assumes M_finite: "finite M"
    and M_wf: "wftrms M"
  shows "comp_timpl_closure M (set TI) = timpl_closure_set M (set TI)"
using lfp_while''[OF timpls_Un_mono[of M "set TI"]]
      timpl_closure_set_finite[OF M_finite M_wf]
      timpl_closure_set_lfp[of M "set TI"]
unfolding comp_timpl_closure_def Let_def by presburger
```

```
context
begin

private lemma analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux1:
  fixes M::"('fun,'atom,'sets) prot_terms"
  assumes f: "arity_f f = length T" "arity_f f > 0" "Ana_f f = (K, R)"
    and i: "i < length R"
    and M: "timpl_closure_set M TI ⊢_c T ! (R ! i)"
    and m: "Fun (Fu f) T ∈ M"
    and t: "Fun (Fu f) S ∈ timpl_closure (Fun (Fu f) T) TI"
  shows "timpl_closure_set M TI ⊢_c S ! (R ! i)"
proof -
  have "R ! i < length T" using i Ana_f_assm2_alt[OF f(3)] f(1) by simp
  thus ?thesis
    using timpl_closure_Fun_inv'(1,2)[OF t] intruder_synth_timpl_closure'[OF M]
    by presburger
qed

private lemma analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2:
  fixes M::"('fun,'atom,'sets) prot_terms"
  assumes M: "∀s ∈ set (snd (Ana m)). timpl_closure_set M TI ⊢_c s"
    and m: "m ∈ M"
    and t: "t ∈ timpl_closure m TI"
    and s: "s ∈ set (snd (Ana t))"
  shows "timpl_closure_set M TI ⊢_c s"
proof -
  obtain f S K N where fS: "t = Fun (Fu f) S" "arity_f f = length S" "0 < arity_f f"
      and Ana_f: "Ana_f f = (K, N)"
      and Ana_t: "Ana t = (K ·_list (!) S, map ((!) S) N)"
    using Ana_nonempty_inv[of t] s by fastforce
  then obtain T where T: "m = Fun (Fu f) T" "length T = length S"
    using t timpl_closure_Fu_inv'[of f S m TI]
    by moura
  hence Ana_m: "Ana m = (K ·_list (!) T, map ((!) T) N)"
    using fS(2,3) Ana_f by auto

  obtain i where i: "i < length N" "s = S ! (N ! i)"
    using s[unfolded fS(1)] Ana_t[unfolded fS(1)] T(2)
          in_set_conv_nth[of s "map (λi. S ! i) N"]
    by auto
  hence "timpl_closure_set M TI ⊢_c T ! (N ! i)"
    using M[unfolded T(1)] Ana_m[unfolded T(1)] T(2)
    by simp
  thus ?thesis
    using analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux1[
            OF fS(2)[unfolded T(2)[symmetric]] fS(3) Ana_f
               i(1) _ m[unfolded T(1)] t[unfolded fS(1) T(1)]]
          i(2)
    by argo
qed

lemma analyzed_closed_mod_timpls_is_analyzed_timpl_closure_set:
  fixes M::"('fun,'atom,'sets) prot_term list"
  assumes TI': "set TI' = {(a,b) ∈ (set TI)^+. a ≠ b}"
    and M_wf: "wf_trms (set M)"
  shows "analyzed_closed_mod_timpls M TI' ⟷ analyzed (timpl_closure_set (set M) (set TI))"
    (is "?A ⟷ ?B")
proof
  let ?C = "∀t ∈ timpl_closure_set (set M) (set TI).
              analyzed_in t (timpl_closure_set (set M) (set TI))"

  let ?P = "λT. ∀t ∈ set T. timpl_closure_set (set M) (set TI) ⊢_c t"
  let ?Q = "λt. ∀s ∈ comp_timpl_closure {t} (set TI'). case Ana s of (K, R) ⇒ ?P K ⟶ ?P R"
```

```
note defs = analyzed_closed_mod_timpls_def analyzed_in_code
note 0 = intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI', of M]
note 1 = timpl_closure_set_is_timpl_closure_union[of _ "set TI"]

have 2: "comp_timpl_closure {t} (set TI') = timpl_closure_set {t} (set TI)"
  when t: "t ∈ set M" "wf_trm t" for t
  using t timpl_closure_set_timpls_trancl_eq'[of "{t}" "set TI"]
        comp_timpl_closure_is_timpl_closure_set[of "{t}" TI']
  unfolding TI'[symmetric]
  by blast
hence 3: "comp_timpl_closure {t} (set TI') ⊆ timpl_closure_set (set M) (set TI)"
  when t: "t ∈ set M" "wf_trm t" for t
  using t timpl_closure_set_mono[of "{t}" "set M"]
  by fast

have ?A when C: ?C
  unfolding analyzed_closed_mod_timpls_def
            intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI']
            list_all_iff Let_def
proof (intro ballI)
  fix t assume t: "t ∈ set M"
  show "if ?P (fst (Ana t)) then ?P (snd (Ana t)) else ?Q t" (is ?R)
  proof (cases "?P (fst (Ana t))")
    case True
    hence "?P (snd (Ana t))"
      using C timpl_closure_setI[OF t, of "set TI"] prod.exhaust_sel
      unfolding analyzed_in_def by blast
    thus ?thesis using True by simp
  next
    case False
    have "?Q t" using 3[OF t] C M_wf t unfolding analyzed_in_def by auto
    thus ?thesis using False by argo
  qed
qed
thus ?A when B: ?B using B analyzed_is_all_analyzed_in by metis

have ?C when A: ?A unfolding analyzed_in_def Let_def
proof (intro ballI allI impI; elim conjE)
  fix t K T s
  assume t: "t ∈ timpl_closure_set (set M) (set TI)"
    and s: "s ∈ set T"
    and Ana_t: "Ana t = (K, T)"
    and K: "∀k ∈ set K. timpl_closure_set (set M) (set TI) ⊢_c k"

  obtain m where m: "m ∈ set M" "t ∈ timpl_closure m (set TI)"
    using timpl_closure_set_is_timpl_closure_union t by moura

  show "timpl_closure_set (set M) (set TI) ⊢_c s"
  proof (cases "∀k ∈ set (fst (Ana m)). timpl_closure_set (set M) (set TI) ⊢_c k")
    case True
    hence *: "∀r ∈ set (snd (Ana m)). timpl_closure_set (set M) (set TI) ⊢_c r"
      using m(1) A
      unfolding analyzed_closed_mod_timpls_def
                intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI']
                list_all_iff
      by simp

    show ?thesis
      using K s Ana_t A
            analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2[OF * m]
      by simp
  next
```

```
      case False
      hence "?Q m"
        using m(1) A
        unfolding analyzed_closed_mod_timpls_def
                  intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI']
                  list_all_iff Let_def
        by auto
      moreover have "comp_timpl_closure {m} (set TI') = timpl_closure m (set TI)"
        using 2[OF m(1)] timpl_closureton_is_timpl_closure M_wf m(1)
        by blast
      ultimately show ?thesis
        using m(2) K s Ana_t
        unfolding Let_def by auto
    qed
  qed
  thus ?B when A: ?A using A analyzed_is_all_analyzed_in by metis
qed

lemma analyzed_closed_mod_timpls'_is_analyzed_timpl_closure_set:
  fixes M::"('fun,'atom,'sets) prot_term list"
  assumes M_wf: "wf_trms (set M)"
  shows "analyzed_closed_mod_timpls' M TI ⟷ analyzed (timpl_closure_set (set M) (set TI))"
    (is "?A ⟷ ?B")
proof
  let ?C = "∀t ∈ timpl_closure_set (set M) (set TI). analyzed_in t (timpl_closure_set (set M) (set TI))"

  let ?P = "λT. ∀t ∈ set T. timpl_closure_set (set M) (set TI) ⊢_c t"
  let ?Q = "λt. ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K, R) ⇒ ?P K ⟶ ?P R"

  note defs = analyzed_closed_mod_timpls'_def analyzed_in_code
  note 0 = intruder_synth_mod_timpls'_is_synth_timpl_closure_set[of M TI]
  note 1 = timpl_closure_set_is_timpl_closure_union[of _ "set TI"]

  have 2: "comp_timpl_closure {t} (set TI) = timpl_closure_set {t} (set TI)"
    when t: "t ∈ set M" "wf_trm t" for t
    using t timpl_closure_set_timpls_trancl_eq[of "{t}" "set TI"]
          comp_timpl_closure_is_timpl_closure_set[of "{t}"]
    by blast
  hence 3: "comp_timpl_closure {t} (set TI) ⊆ timpl_closure_set (set M) (set TI)"
    when t: "t ∈ set M" "wf_trm t" for t
    using t timpl_closure_set_mono[of "{t}" "set M"]
    by fast

  have ?A when C: ?C
    unfolding analyzed_closed_mod_timpls'_def
              intruder_synth_mod_timpls'_is_synth_timpl_closure_set
              list_all_iff Let_def
  proof (intro ballI)
    fix t assume t: "t ∈ set M"
    show "if ?P (fst (Ana t)) then ?P (snd (Ana t)) else ?Q t" (is ?R)
    proof (cases "?P (fst (Ana t))")
      case True
      hence "?P (snd (Ana t))"
        using C timpl_closure_setI[OF t, of "set TI"] prod.exhaust_sel
        unfolding analyzed_in_def by blast
      thus ?thesis using True by simp
    next
      case False
      have "?Q t" using 3[OF t] C M_wf t unfolding analyzed_in_def by auto
      thus ?thesis using False by argo
    qed
  qed
  thus ?A when B: ?B using B analyzed_is_all_analyzed_in by metis
```

```
  have ?C when A: ?A unfolding analyzed_in_def Let_def
  proof (intro ballI allI impI; elim conjE)
    fix t K T s
    assume t: "t ∈ timpl_closure_set (set M) (set TI)"
      and s: "s ∈ set T"
      and Ana_t: "Ana t = (K, T)"
      and K: "∀k ∈ set K. timpl_closure_set (set M) (set TI) ⊢c k"

    obtain m where m: "m ∈ set M" "t ∈ timpl_closure m (set TI)"
      using timpl_closure_set_is_timpl_closure_union t by moura

    show "timpl_closure_set (set M) (set TI) ⊢c s"
    proof (cases "∀k ∈ set (fst (Ana m)). timpl_closure_set (set M) (set TI) ⊢c k")
      case True
      hence *: "∀r ∈ set (snd (Ana m)). timpl_closure_set (set M) (set TI) ⊢c r"
        using m(1) A
        unfolding analyzed_closed_mod_timpls'_def
                  intruder_synth_mod_timpls'_is_synth_timpl_closure_set
                  list_all_iff
        by simp

      show ?thesis
        using K s Ana_t A
              analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2[OF * m]
        by simp
    next
      case False
      hence "?Q m"
        using m(1) A
        unfolding analyzed_closed_mod_timpls'_def
                  intruder_synth_mod_timpls'_is_synth_timpl_closure_set
                  list_all_iff Let_def
        by auto
      moreover have "comp_timpl_closure {m} (set TI) = timpl_closure m (set TI)"
        using 2[OF m(1)] timpl_closureton_is_timpl_closure M_wf m(1)
        by blast
      ultimately show ?thesis
        using m(2) K s Ana_t
        unfolding Let_def by auto
    qed
  qed
  thus ?B when A: ?A using A analyzed_is_all_analyzed_in by metis
qed

end

end

end
```

2.6 Stateful Protocol Verification (Stateful_Protocol_Verification)

```
theory Stateful_Protocol_Verification
imports Stateful_Protocol_Model Term_Implication
begin
```

2.6.1 Fixed-Point Intruder Deduction Lemma

```
context stateful_protocol_model
begin
```

```
abbreviation pubval_terms::"('fun,'atom,'sets) prot_terms" where
  "pubval_terms ≡ {t. ∃f ∈ funs_term t. is_Val f ∧ public f}"

abbreviation abs_terms::"('fun,'atom,'sets) prot_terms" where
  "abs_terms ≡ {t. ∃f ∈ funs_term t. is_Abs f}"

definition intruder_deduct_GSMP::
  "[('fun,'atom,'sets) prot_terms,
    ('fun,'atom,'sets) prot_terms,
    ('fun,'atom,'sets) prot_term]
    ⇒ bool" ("⟨_;_⟩ ⊢GSMP _" 50)
where
  "⟨M; T⟩ ⊢GSMP t ≡ intruder_deduct_restricted M (λt. t ∈ GSMP T - (pubval_terms ∪ abs_terms)) t"

lemma intruder_deduct_GSMP_induct[consumes 1, case_names AxiomH ComposeH DecomposeH]:
  assumes "⟨M; T⟩ ⊢GSMP t" "⋀t. t ∈ M ⟹ P M t"
          "⋀S f. ⟦length S = arity f; public f;
                 ⋀s. s ∈ set S ⟹ ⟨M; T⟩ ⊢GSMP s;
                 ⋀s. s ∈ set S ⟹ P M s;
                 Fun f S ∈ GSMP T - (pubval_terms ∪ abs_terms)
                 ⟧ ⟹ P M (Fun f S)"
          "⋀t K T' t_i. ⟦⟨M; T⟩ ⊢GSMP t; P M t; Ana t = (K, T'); ⋀k. k ∈ set K ⟹ ⟨M; T⟩ ⊢GSMP k;
                        ⋀k. k ∈ set K ⟹ P M k; t_i ∈ set T'⟧ ⟹ P M t_i"
  shows "P M t"
proof -
  let ?Q = "λt. t ∈ GSMP T - (pubval_terms ∪ abs_terms)"
  show ?thesis
    using intruder_deduct_restricted_induct[of M ?Q t "λM Q t. P M t"] assms
    unfolding intruder_deduct_GSMP_def
    by blast
qed

lemma pubval_terms_subst:
  assumes "t · ϑ ∈ pubval_terms" "ϑ ‘ fv t ∩ pubval_terms = {}"
  shows "t ∈ pubval_terms"
using assms(1,2)
proof (induction t)
  case (Fun f T)
  let ?P = "λf. is_Val f ∧ public f"
  from Fun show ?case
  proof (cases "?P f")
    case False
    then obtain t where t: "t ∈ set T" "t · ϑ ∈ pubval_terms"
      using Fun.prems by auto
    hence "ϑ ‘ fv t ∩ pubval_terms = {}" using Fun.prems(2) by auto
    thus ?thesis using Fun.IH[OF t] t(1) by auto
  qed force
qed simp

lemma abs_terms_subst:
  assumes "t · ϑ ∈ abs_terms" "ϑ ‘ fv t ∩ abs_terms = {}"
  shows "t ∈ abs_terms"
using assms(1,2)
proof (induction t)
  case (Fun f T)
  let ?P = "λf. is_Abs f"
  from Fun show ?case
  proof (cases "?P f")
    case False
    then obtain t where t: "t ∈ set T" "t · ϑ ∈ abs_terms"
      using Fun.prems by auto
    hence "ϑ ‘ fv t ∩ abs_terms = {}" using Fun.prems(2) by auto
    thus ?thesis using Fun.IH[OF t] t(1) by auto
```

```
  qed force
qed simp

lemma pubval_terms_subst':
  assumes "t · ϑ ∈ pubval_terms" "∀n. Val (n,True) ∉ ⋃(funs_term ` (ϑ ` fv t))"
  shows "t ∈ pubval_terms"
proof -
  have "¬public f"
    when fs: "f ∈ funs_term s" "s ∈ subterms_set (ϑ ` fv t)" "is_Val f"
    for f s
  proof -
    obtain T where T: "Fun f T ∈ subterms s" using funs_term_Fun_subterm[OF fs(1)] by moura
    hence "Fun f T ∈ subterms_set (ϑ ` fv t)" using fs(2) in_subterms_subset_Union by blast
    thus ?thesis using assms(2) funs_term_Fun_subterm'[of f T] fs(3) by (cases f) force+
  qed
  thus ?thesis using pubval_terms_subst[OF assms(1)] by force
qed

lemma abs_terms_subst':
  assumes "t · ϑ ∈ abs_terms" "∀n. Abs n ∉ ⋃(funs_term ` (ϑ ` fv t))"
  shows "t ∈ abs_terms"
proof -
  have "¬is_Abs f" when fs: "f ∈ funs_term s" "s ∈ subterms_set (ϑ ` fv t)" for f s
  proof -
    obtain T where T: "Fun f T ∈ subterms s" using funs_term_Fun_subterm[OF fs(1)] by moura
    hence "Fun f T ∈ subterms_set (ϑ ` fv t)" using fs(2) in_subterms_subset_Union by blast
    thus ?thesis using assms(2) funs_term_Fun_subterm'[of f T] by (cases f) auto
  qed
  thus ?thesis using abs_terms_subst[OF assms(1)] by force
qed

lemma pubval_terms_subst_range_disj:
  "subst_range ϑ ∩ pubval_terms = {} ⟹ ϑ ` fv t ∩ pubval_terms = {}"
proof (induction t)
  case (Var x) thus ?case by (cases "x ∈ subst_domain ϑ") auto
qed auto

lemma abs_terms_subst_range_disj:
  "subst_range ϑ ∩ abs_terms = {} ⟹ ϑ ` fv t ∩ abs_terms = {}"
proof (induction t)
  case (Var x) thus ?case by (cases "x ∈ subst_domain ϑ") auto
qed auto

lemma pubval_terms_subst_range_comp:
  assumes "subst_range ϑ ∩ pubval_terms = {}" "subst_range δ ∩ pubval_terms = {}"
  shows "subst_range (ϑ ∘s δ) ∩ pubval_terms = {}"
proof -
  { fix t f assume t:
      "t ∈ subst_range (ϑ ∘s δ)" "f ∈ funs_term t" "is_Val f" "public f"
    then obtain x where x: "(ϑ ∘s δ) x = t" by auto
    have "ϑ x ∉ pubval_terms" using assms(1) by (cases "ϑ x ∈ subst_range ϑ") force+
    hence "(ϑ ∘s δ) x ∉ pubval_terms"
      using assms(2) pubval_terms_subst[of "ϑ x" δ] pubval_terms_subst_range_disj
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3,4) x by blast
  } thus ?thesis by fast
qed

lemma pubval_terms_subst_range_comp':
  assumes "(ϑ ` X) ∩ pubval_terms = {}" "(δ ` fv_set (ϑ ` X)) ∩ pubval_terms = {}"
  shows "((ϑ ∘s δ) ` X) ∩ pubval_terms = {}"
proof -
  { fix t f assume t:
```

```
      "t ∈ (ϑ ∘s δ) ‘ X" "f ∈ funs_term t" "is_Val f" "public f"
    then obtain x where x: "(ϑ ∘s δ) x = t" "x ∈ X" by auto
    have "ϑ x ∉ pubval_terms" using assms(1) x(2) by force
    moreover have "fv (ϑ x) ⊆ fvset (ϑ ‘ X)" using x(2) by (auto simp add: fv_subset)
    hence "δ ‘ fv (ϑ x) ∩ pubval_terms = {}" using assms(2) by auto
    ultimately have "(ϑ ∘s δ) x ∉ pubval_terms"
      using pubval_terms_subst[of "ϑ x" δ]
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3,4) x by blast
  } thus ?thesis by fast
qed

lemma abs_terms_subst_range_comp:
  assumes "subst_range ϑ ∩ abs_terms = {}" "subst_range δ ∩ abs_terms = {}"
  shows "subst_range (ϑ ∘s δ) ∩ abs_terms = {}"
proof -
  { fix t f assume t: "t ∈ subst_range (ϑ ∘s δ)" "f ∈ funs_term t" "is_Abs f"
    then obtain x where x: "(ϑ ∘s δ) x = t" by auto
    have "ϑ x ∉ abs_terms" using assms(1) by (cases "ϑ x ∈ subst_range ϑ") force+
    hence "(ϑ ∘s δ) x ∉ abs_terms"
      using assms(2) abs_terms_subst[of "ϑ x" δ] abs_terms_subst_range_disj
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3) x by blast
  } thus ?thesis by fast
qed

lemma abs_terms_subst_range_comp':
  assumes "(ϑ ‘ X) ∩ abs_terms = {}" "(δ ‘ fvset (ϑ ‘ X)) ∩ abs_terms = {}"
  shows "((ϑ ∘s δ) ‘ X) ∩ abs_terms = {}"
proof -
  { fix t f assume t:
      "t ∈ (ϑ ∘s δ) ‘ X" "f ∈ funs_term t" "is_Abs f"
    then obtain x where x: "(ϑ ∘s δ) x = t" "x ∈ X" by auto
    have "ϑ x ∉ abs_terms" using assms(1) x(2) by force
    moreover have "fv (ϑ x) ⊆ fvset (ϑ ‘ X)" using x(2) by (auto simp add: fv_subset)
    hence "δ ‘ fv (ϑ x) ∩ abs_terms = {}" using assms(2) by auto
    ultimately have "(ϑ ∘s δ) x ∉ abs_terms"
      using abs_terms_subst[of "ϑ x" δ]
      by (metis (mono_tags, lifting) subst_compose_def)
    hence False using t(2,3) x by blast
  } thus ?thesis by fast
qed

context
begin
private lemma Ana_abs_aux1:
  fixes δ::"(('fun,'atom,'sets) prot_fun, nat, ('fun,'atom,'sets) prot_var) gsubst"
    and α::"nat × bool ⇒ 'sets set"
  assumes "Anaf f = (K,T)"
  shows "(K ·list δ) ·αlist α = K ·list (λn. δ n ·α α)"
proof -
  { fix k assume "k ∈ set K"
    hence "k ∈ subtermsset (set K)" by force
    hence "k · δ ·α α = k · (λn. δ n ·α α)"
    proof (induction k)
      case (Fun g S)
      have "⋀s. s ∈ set S ⟹ s · δ ·α α = s · (λn. δ n ·α α)"
        using Fun.IH in_subterms_subset_Union[OF Fun.prems] Fun_param_in_subterms[of _ S g]
        by (meson contra_subsetD)
      thus ?case using Anaf_assm1_alt[OF assms Fun.prems] by (cases g) auto
    qed simp
  } thus ?thesis unfolding abs_apply_list_def by force
qed
```

```
private lemma Ana_abs_aux2:
  fixes α::"nat × bool ⇒ 'sets set"
    and K::"(('fun,'atom,'sets) prot_fun, nat) term list"
    and M::"nat list"
    and T::"('fun,'atom,'sets) prot_term list"
  assumes "∀i ∈ fv_set (set K) ∪ set M. i < length T"
    and "(K ·_list (!) T) ·_αlist α = K ·_list (λn. T ! n ·_α α)"
  shows "(K ·_list (!) T) ·_αlist α = K ·_list (!) (map (λs. s ·_α α) T)" (is "?A1 = ?A2")
    and "(map ((!) T) M) ·_αlist α = map ((!) (map (λs. s ·_α α) T)) M" (is "?B1 = ?B2")
proof -
  have "T ! i ·_α α = (map (λs. s ·_α α) T) ! i" when "i ∈ fv_set (set K)" for i
    using that assms(1) by auto
  hence "k · (λi. T ! i ·_α α) = k · (λi. (map (λs. s ·_α α) T) ! i)" when "k ∈ set K" for k
    using that term_subst_eq_conv[of k "λi. T ! i ·_α α" "λi. (map (λs. s ·_α α) T) ! i"]
    by auto
  thus "?A1 = ?A2" using assms(2) by (force simp add: abs_apply_terms_def)

  have "T ! i ·_α α = map (λs. s ·_α α) T ! i" when "i ∈ set M" for i
    using that assms(1) by auto
  thus "?B1 = ?B2" by (force simp add: abs_apply_list_def)
qed

private lemma Ana_abs_aux1_set:
  fixes δ::"(('fun,'atom,'sets) prot_fun, nat, ('fun,'atom,'sets) prot_var) gsubst"
    and α::"nat × bool ⇒ 'sets set"
  assumes "Ana_f f = (K,T)"
  shows "(set K ·_set δ) ·_αset α = set K ·_set (λn. δ n ·_α α)"
proof -
  { fix k assume "k ∈ set K"
    hence "k ∈ subterms_set (set K)" by force
    hence "k · δ ·_α α = k · (λn. δ n ·_α α)"
    proof (induction k)
      case (Fun g S)
      have "⋀s. s ∈ set S ⟹ s · δ ·_α α = s · (λn. δ n ·_α α)"
        using Fun.IH in_subterms_subset_Union[OF Fun.prems] Fun_param_in_subterms[of _ S g]
        by (meson contra_subsetD)
      thus ?case using Ana_f_assm1_alt[OF assms Fun.prems] by (cases g) auto
    qed simp
  } thus ?thesis unfolding abs_apply_terms_def by force
qed

private lemma Ana_abs_aux2_set:
  fixes α::"nat × bool ⇒ 'sets set"
    and K::"(('fun,'atom,'sets) prot_fun, nat) terms"
    and M::"nat set"
    and T::"('fun,'atom,'sets) prot_term list"
  assumes "∀i ∈ fv_set K ∪ M. i < length T"
    and "(K ·_set (!) T) ·_αset α = K ·_set (λn. T ! n ·_α α)"
  shows "(K ·_set (!) T) ·_αset α = K ·_set (!) (map (λs. s ·_α α) T)" (is "?A1 = ?A2")
    and "((!) T ‘ M) ·_αset α = (!) (map (λs. s ·_α α) T) ‘ M" (is "?B1 = ?B2")
proof -
  have "T ! i ·_α α = (map (λs. s ·_α α) T) ! i" when "i ∈ fv_set K" for i
    using that assms(1) by auto
  hence "k · (λi. T ! i ·_α α) = k · (λi. (map (λs. s ·_α α) T) ! i)" when "k ∈ K" for k
    using that term_subst_eq_conv[of k "λi. T ! i ·_α α" "λi. (map (λs. s ·_α α) T) ! i"]
    by auto
  thus "?A1 = ?A2" using assms(2) by (force simp add: abs_apply_terms_def)

  have "T ! i ·_α α = map (λs. s ·_α α) T ! i" when "i ∈ M" for i
    using that assms(1) by auto
  thus "?B1 = ?B2" by (force simp add: abs_apply_terms_def)
qed
```

```
lemma Ana_abs:
  fixes t::"('fun,'atom,'sets) prot_term"
  assumes "Ana t = (K, T)"
  shows "Ana (t ·α α) = (K ·αlist α, T ·αlist α)"
  using assms
proof (induction t rule: Ana.induct)
  case (1 f S)
  obtain K' T' where *: "Ana_f f = (K',T')" by moura
  show ?case using 1
  proof (cases "arity_f f = length S ∧ arity_f f > 0")
    case True
    hence "K = K' ·list (!) S" "T = map ((!) S) T'"
        and **: "arity_f f = length (map (λs. s ·α α) S)" "arity_f f > 0"
      using 1 * by auto
    hence "K ·αlist α = K' ·list (!) (map (λs. s ·α α) S)"
          "T ·αlist α = map ((!) (map (λs. s ·α α) S)) T'"
      using Ana_f_assm2_alt[OF *] Ana_abs_aux2[OF _ Ana_abs_aux1[OF *], of T' S α]
      unfolding abs_apply_list_def
      by auto
    moreover have "Fun (Fu f) S ·α α = Fun (Fu f) (map (λs. s ·α α) S)" by simp
    ultimately show ?thesis using Ana_Fu_intro[OF ** *] by metis
  qed (auto simp add: abs_apply_list_def)
qed (simp_all add: abs_apply_list_def)
end

lemma deduct_FP_if_deduct:
  fixes M IK FP::"('fun,'atom,'sets) prot_terms"
  assumes IK: "IK ⊆ GSMP M - (pubval_terms ∪ abs_terms)" "∀t ∈ IK ·αset α. FP ⊢c t"
    and t: "IK ⊢ t" "t ∈ GSMP M - (pubval_terms ∪ abs_terms)"
  shows "FP ⊢ t ·α α"
proof -
  let ?P = "λf. is_Val f ⟶ ¬public f"
  let ?GSMP = "GSMP M - (pubval_terms ∪ abs_terms)"

  have 1: "∀m ∈ IK. m ∈ ?GSMP"
    using IK(1) by blast

  have 2: "∀t t'. t ∈ ?GSMP ⟶ t' ⊑ t ⟶ t' ∈ ?GSMP"
  proof (intro allI impI)
    fix t t' assume t: "t ∈ ?GSMP" "t' ⊑ t"
    hence "t' ∈ GSMP M" using ground_subterm unfolding GSMP_def by auto
    moreover have "¬public f"
      when "f ∈ funs_term t" "is_Val f" for f
      using t(1) that by auto
    hence "¬public f"
      when "f ∈ funs_term t'" "is_Val f" for f
      using that subtermeq_imp_funs_term_subset[OF t(2)] by auto
    moreover have "¬is_Abs f" when "f ∈ funs_term t" for f using t(1) that by auto
    hence "¬is_Abs f" when "f ∈ funs_term t'" for f
      using that subtermeq_imp_funs_term_subset[OF t(2)] by auto
    ultimately show "t' ∈ ?GSMP" by simp
  qed

  have 3: "∀t K T k. t ∈ ?GSMP ⟶ Ana t = (K, T) ⟶ k ∈ set K ⟶ k ∈ ?GSMP"
  proof (intro allI impI)
    fix t K T k assume t: "t ∈ ?GSMP" "Ana t = (K, T)" "k ∈ set K"
    hence "k ∈ GSMP M" using GSMP_Ana_key by blast
    moreover have "∀f ∈ funs_term t. ?P f" using t(1) by auto
    with t(2,3) have "∀f ∈ funs_term k. ?P f"
    proof (induction t arbitrary: k rule: Ana.induct)
      case 1 thus ?case by (metis Ana_Fu_keys_not_pubval_terms surj_pair)
    qed auto
```

```
    moreover have "∀f ∈ funs_term t. ¬is_Abs f" using t(1) by auto
    with t(2,3) have "∀f ∈ funs_term k. ¬is_Abs f"
    proof (induction t arbitrary: k rule: Ana.induct)
      case 1 thus ?case by (metis Ana_Fu_keys_not_abs_terms surj_pair)
    qed auto
    ultimately show "k ∈ ?GSMP" by simp
  qed

  have "⟨IK; M⟩ ⊢_GSMP t"
    unfolding intruder_deduct_GSMP_def
    by (rule restricted_deduct_if_deduct'[OF 1 2 3 t])
  thus ?thesis
  proof (induction t rule: intruder_deduct_GSMP_induct)
    case (AxiomH t)
    show ?case using IK(2) abs_in[OF AxiomH.hyps] by force
  next
    case (ComposeH T f)
    have *: "Fun f T ·_α α = Fun f (map (λt. t ·_α α) T)"
      using ComposeH.hyps(2,4)
      by (cases f) auto

    have **: "length (map (λt. t ·_α α) T) = arity f"
      using ComposeH.hyps(1)
      by auto

    show ?case
      using intruder_deduct.Compose[OF ** ComposeH.hyps(2)] ComposeH.IH(1) *
      by auto
  next
    case (DecomposeH t K T' t_i)
    have *: "Ana (t ·_α α) = (K ·_αlist α, T' ·_αlist α)"
      using Ana_abs[OF DecomposeH.hyps(2)]
      by metis

    have **: "t_i ·_α α ∈ set (T' ·_αlist α)"
      using DecomposeH.hyps(4) abs_in abs_list_set_is_set_abs_set[of T']
      by auto

    have ***: "FP ⊢ k"
      when k: "k ∈ set (K ·_αlist α)" for k
    proof -
      obtain k' where k': "k' ∈ set K" "k = k' ·_α α"
        by (metis (no_types) k abs_apply_terms_def imageE abs_list_set_is_set_abs_set)

      show "FP ⊢ k"
        using DecomposeH.IH k' by blast
    qed

    show ?case
      using intruder_deduct.Decompose[OF _ * _ **]
            DecomposeH.IH(1) ***(1)
      by blast
  qed
qed

end
```

2.6.2 Computing and Checking Term Implications and Messages

```
context stateful_protocol_model
begin

abbreviation (input) "absc s ≡ (Fun (Abs s) []::('fun, 'atom, 'sets) prot_term)"
```

```
fun absdbupd where
  "absdbupd [] _ a = a"
| "absdbupd (insert⟨Var y, Fun (Set s) T⟩#D) x a = (
    if x = y then absdbupd D x (insert s a) else absdbupd D x a)"
| "absdbupd (delete⟨Var y, Fun (Set s) T⟩#D) x a = (
    if x = y then absdbupd D x (a - {s}) else absdbupd D x a)"
| "absdbupd (_#D) x a = absdbupd D x a"

lemma absdbupd_cons_cases:
  "absdbupd (insert⟨Var x, Fun (Set s) T⟩#D) x d = absdbupd D x (insert s d)"
  "absdbupd (delete⟨Var x, Fun (Set s) T⟩#D) x d = absdbupd D x (d - {s})"
  "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T) ⟹ absdbupd (insert⟨t,u⟩#D) x d = absdbupd D x d"
  "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T) ⟹ absdbupd (delete⟨t,u⟩#D) x d = absdbupd D x d"
proof -
  assume *: "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T)"
  let ?P = "absdbupd (insert⟨t,u⟩#D) x d = absdbupd D x d"
  let ?Q = "absdbupd (delete⟨t,u⟩#D) x d = absdbupd D x d"
  { fix y f T assume "t = Fun f T ∨ u = Var y" hence ?P ?Q by auto
  } moreover {
    fix y f T assume "t = Var y" "u = Fun f T" hence ?P using * by (cases f) auto
  } moreover {
    fix y f T assume "t = Var y" "u = Fun f T" hence ?Q using * by (cases f) auto
  } ultimately show ?P ?Q by (metis term.exhaust)+
qed simp_all

lemma absdbupd_filter: "absdbupd S x d = absdbupd (filter is_Update S) x d"
by (induction S x d rule: absdbupd.induct) simp_all

lemma absdbupd_append:
  "absdbupd (A@B) x d = absdbupd B x (absdbupd A x d)"
proof (induction A arbitrary: d)
  case (Cons a A) thus ?case
  proof (cases a)
    case (Insert t u) thus ?thesis
    proof (cases "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T)")
      case False
      then obtain s T where "t = Var x" "u = Fun (Set s) T" by moura
      thus ?thesis by (simp add: Insert Cons.IH absdbupd_cons_cases(1))
    qed (simp_all add: Cons.IH absdbupd_cons_cases(3))
  next
    case (Delete t u) thus ?thesis
    proof (cases "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T)")
      case False
      then obtain s T where "t = Var x" "u = Fun (Set s) T" by moura
      thus ?thesis by (simp add: Delete Cons.IH absdbupd_cons_cases(2))
    qed (simp_all add: Cons.IH absdbupd_cons_cases(4))
  qed simp_all
qed simp

lemma absdbupd_wellformed_transaction:
  assumes T: "wellformed_transaction T"
  shows "absdbupd (unlabel (transaction_strand T)) = absdbupd (unlabel (transaction_updates T))"
proof -
  define S0 where "S0 ≡ unlabel (transaction_strand T)"
  define S1 where "S1 ≡ unlabel (transaction_receive T)"
  define S2 where "S2 ≡ unlabel (transaction_selects T)"
  define S3 where "S3 ≡ unlabel (transaction_checks T)"
  define S4 where "S4 ≡ unlabel (transaction_updates T)"
  define S5 where "S5 ≡ unlabel (transaction_send T)"

  note S_defs = S0_def S1_def S2_def S3_def S4_def S5_def
```

```
  have 0: "list_all is_Receive S1"
          "list_all is_Assignment S2"
          "list_all is_Check S3"
          "list_all is_Update S4"
          "list_all is_Send S5"
    using T unfolding wellformed_transaction_def S_defs by metis+

  have "filter is_Update S1 = []"
       "filter is_Update S2 = []"
       "filter is_Update S3 = []"
       "filter is_Update S4 = S4"
       "filter is_Update S5 = []"
    using list_all_filter_nil[OF 0(1), of is_Update]
          list_all_filter_nil[OF 0(2), of is_Update]
          list_all_filter_nil[OF 0(3), of is_Update]
          list_all_filter_eq[OF 0(4)]
          list_all_filter_nil[OF 0(5), of is_Update]
    by blast+
  moreover have "S0 = S1@S2@S3@S4@S5"
    unfolding S_defs transaction_strand_def unlabel_def by auto
  ultimately have "filter is_Update S0 = S4"
    using filter_append[of is_Update] list_all_append[of is_Update]
    by simp
  thus ?thesis
    using absdbupd_filter[of S0]
    unfolding S_defs by presburger
qed

fun abs_substs_set::
  "[('fun,'atom,'sets) prot_var list,
    'sets set list,
    ('fun,'atom,'sets) prot_var ⇒ 'sets set,
    ('fun,'atom,'sets) prot_var ⇒ 'sets set]
  ⇒ (((('fun,'atom,'sets) prot_var × 'sets set) list) list"
where
  "abs_substs_set [] _ _ _ = [[]]"
| "abs_substs_set (x#xs) as posconstrs negconstrs = (
    let bs = filter (λa. posconstrs x ⊆ a ∧ a ∩ negconstrs x = {}) as
    in concat (map (λb. map (λδ. (x, b)#δ) (abs_substs_set xs as posconstrs negconstrs)) bs))"

definition abs_substs_fun::
  "[((('fun,'atom,'sets) prot_var × 'sets set) list,
    ('fun,'atom,'sets) prot_var]
  ⇒ 'sets set"
where
  "abs_substs_fun δ x = (case find (λb. fst b = x) δ of Some (_,a) ⇒ a | None ⇒ {})"

lemmas abs_substs_set_induct = abs_substs_set.induct[case_names Nil Cons]

fun transaction_poschecks_comp::
  "(('fun,'atom,'sets) prot_fun, ('fun,'atom,'sets) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets) prot_var ⇒ 'sets set)"
where
  "transaction_poschecks_comp [] = (λ_. {})"
| "transaction_poschecks_comp (⟨_: Var x ∈ Fun (Set s) []⟩#T) = (
    let f = transaction_poschecks_comp T in f(x := insert s (f x)))"
| "transaction_poschecks_comp (_#T) = transaction_poschecks_comp T"

fun transaction_negchecks_comp::
  "(('fun,'atom,'sets) prot_fun, ('fun,'atom,'sets) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets) prot_var ⇒ 'sets set)"
where
  "transaction_negchecks_comp [] = (λ_. {})"
```

```
| "transaction_negchecks_comp (⟨Var x not in Fun (Set s) []⟩#T) = (
    let f = transaction_negchecks_comp T in f(x := insert s (f x)))"
| "transaction_negchecks_comp (_#T) = transaction_negchecks_comp T"

definition transaction_check_pre where
  "transaction_check_pre FP TI T δ ≡
    let C = set (unlabel (transaction_checks T));
        S = set (unlabel (transaction_selects T));
        xs = fv_list_sst (unlabel (transaction_strand T));
        ϑ = λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x
    in (∀x ∈ set (transaction_fresh T). δ x = {}) ∧
       (∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)) ∧
       (∀u ∈ S ∪ C.
          (is_InSet u ⟶ (
            let x = the_elem_term u; s = the_set_term u
            in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∈ δ (the_Var x))) ∧
          ((is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1) ⟶ (
            let x = fst (hd (the_ins u)); s = snd (hd (the_ins u))
            in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∉ δ (the_Var x))))"

definition transaction_check_post where
  "transaction_check_post FP TI T δ ≡
    let xs = fv_list_sst (unlabel (transaction_strand T));
        ϑ = λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x;
        u = λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)
    in (∀x ∈ set xs - set (transaction_fresh T). δ x ≠ u δ x ⟶ List.member TI (δ x, u δ x)) ∧
       (∀t ∈ trms_lsst (transaction_send T). intruder_synth_mod_timpls FP TI (t · ϑ (u δ)))"

definition transaction_check_comp::
  "[('fun,'atom,'sets) prot_term list,
    'sets set list,
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ ((('fun,'atom,'sets) prot_var × 'sets set) list) list"
where
  "transaction_check_comp FP OCC TI T ≡
    let S = unlabel (transaction_strand T);
        C = unlabel (transaction_selects T@transaction_checks T);
        xs = filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S);
        posconstrs = transaction_poschecks_comp C;
        negconstrs = transaction_negchecks_comp C;
        pre_check = transaction_check_pre FP TI T
    in filter (λδ. pre_check (abs_substs_fun δ)) (abs_substs_set xs OCC posconstrs negconstrs)"

definition transaction_check::
  "[('fun,'atom,'sets) prot_term list,
    'sets set list,
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check FP OCC TI T ≡
    list_all (λδ. transaction_check_post FP TI T (abs_substs_fun δ)) (transaction_check_comp FP OCC TI T)"

lemma abs_subst_fun_cons:
  "abs_substs_fun ((x,b)#δ) = (abs_substs_fun δ)(x := b)"
unfolding abs_substs_fun_def by fastforce

lemma abs_substs_cons:
  assumes "δ ∈ set (abs_substs_set xs as poss negs)" "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}"
  shows "(x,b)#δ ∈ set (abs_substs_set (x#xs) as poss negs)"
using assms by auto
```

```
lemma abs_substs_cons':
  assumes δ: "δ ∈ abs_substs_fun ‘ set (abs_substs_set xs as poss negs)"
    and b: "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}"
  shows "δ(x := b) ∈ abs_substs_fun ‘ set (abs_substs_set (x#xs) as poss negs)"
proof -
  obtain ϑ where ϑ: "δ = abs_substs_fun ϑ" "ϑ ∈ set (abs_substs_set xs as poss negs)"
    using δ by moura
  have "abs_substs_fun ((x, b)#ϑ) ∈ abs_substs_fun ‘ set (abs_substs_set (x#xs) as poss negs)"
    using abs_substs_cons[OF ϑ(2) b] by blast
  thus ?thesis
    using ϑ(1) abs_subst_fun_cons[of x b ϑ] by argo
qed

lemma abs_substs_has_all_abs:
  assumes "∀x. x ∈ set xs ⟶ δ x ∈ set as"
    and "∀x. x ∈ set xs ⟶ poss x ⊆ δ x"
    and "∀x. x ∈ set xs ⟶ δ x ∩ negs x = {}"
    and "∀x. x ∉ set xs ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ‘ set (abs_substs_set xs as poss negs)"
using assms
proof (induction xs arbitrary: δ)
  case (Cons x xs)
  define ϑ where "ϑ ≡ λy. if y ∈ set xs then δ y else {}"

  have "ϑ ∈ abs_substs_fun ‘ set (abs_substs_set xs as poss negs)"
    using Cons.prems Cons.IH by (simp add: ϑ_def)
  moreover have "δ x ∈ set as" "poss x ⊆ δ x" "δ x ∩ negs x = {}"
    using Cons.prems(1,2,3) by fastforce+
  ultimately have 0: "ϑ(x := δ x) ∈ abs_substs_fun ‘ set (abs_substs_set (x#xs) as poss negs)"
    by (metis abs_substs_cons')

  have "δ = ϑ(x := δ x)"
  proof
    fix y show "δ y = (ϑ(x := δ x)) y"
    proof (cases "y ∈ set (x#xs)")
      case False thus ?thesis using Cons.prems(4) by (fastforce simp add: ϑ_def)
    qed (auto simp add: ϑ_def)
  qed
  thus ?case by (metis 0)
qed (auto simp add: abs_substs_fun_def)

lemma abs_substs_abss_bounded:
  assumes "δ ∈ abs_substs_fun ‘ set (abs_substs_set xs as poss negs)"
    and "x ∈ set xs"
  shows "δ x ∈ set as"
    and "poss x ⊆ δ x"
    and "δ x ∩ negs x = {}"
using assms
proof (induct xs as poss negs arbitrary: δ rule: abs_substs_set_induct)
  case (Cons y xs as poss negs)
  { case 1 thus ?case using Cons.hyps(1) unfolding abs_substs_fun_def by fastforce }

  { case 2 thus ?case
    proof (cases "x = y")
      case False
      then obtain δ' where δ':
          "δ' ∈ abs_substs_fun ‘ set (abs_substs_set xs as poss negs)" "δ' x = δ x"
        using 2 unfolding abs_substs_fun_def by force
      moreover have "x ∈ set xs" using 2(2) False by simp
      moreover have "∃b. b ∈ set as ∧ poss y ⊆ b ∧ b ∩ negs y = {}"
        using 2 False by auto
      ultimately show ?thesis using Cons.hyps(2) by fastforce
    qed (auto simp add: abs_substs_fun_def)
```

```
  }

  { case 3 thus ?case
    proof (cases "x = y")
      case False
      then obtain δ' where δ':
          "δ' ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs)" "δ' x = δ x"
        using 3 unfolding abs_substs_fun_def by force
      moreover have "x ∈ set xs" using 3(2) False by simp
      moreover have "∃b. b ∈ set as ∧ poss y ⊆ b ∧ b ∩ negs y = {}"
        using 3 False by auto
      ultimately show ?thesis using Cons.hyps(3) by fastforce
    qed (auto simp add: abs_substs_fun_def)
  }
qed (simp_all add: abs_substs_fun_def)

lemma transaction_poschecks_comp_unfold:
  "transaction_poschecks_comp C x = {s. ∃a. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C}"
proof (induction C)
  case (Cons c C) thus ?case
  proof (cases "∃a y s. c = ⟨a: Var y ∈ Fun (Set s) []⟩")
    case True
    then obtain a y s where c: "c = ⟨a: Var y ∈ Fun (Set s) []⟩" by moura

    define f where "f ≡ transaction_poschecks_comp C"

    have "transaction_poschecks_comp (c#C) = f(y := insert s (f y))"
      using c by (simp add: f_def Let_def)
    moreover have "f x = {s. ∃a. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C}"
      using Cons.IH unfolding f_def by blast
    ultimately show ?thesis using c by auto
  next
    case False
    hence "transaction_poschecks_comp (c#C) = transaction_poschecks_comp C" (is ?P)
      using transaction_poschecks_comp.cases[of "c#C" ?P] by force
    thus ?thesis using False Cons.IH by auto
  qed
qed simp

lemma transaction_poschecks_comp_notin_fv_empty:
  assumes "x ∉ fv_sst C"
  shows "transaction_poschecks_comp C x = {}"
using assms transaction_poschecks_comp_unfold[of C x] by fastforce

lemma transaction_negchecks_comp_unfold:
  "transaction_negchecks_comp C x = {s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C}"
proof (induction C)
  case (Cons c C) thus ?case
  proof (cases "∃y s. c = ⟨Var y not in Fun (Set s) []⟩")
    case True
    then obtain y s where c: "c = ⟨Var y not in Fun (Set s) []⟩" by moura

    define f where "f ≡ transaction_negchecks_comp C"

    have "transaction_negchecks_comp (c#C) = f(y := insert s (f y))"
      using c by (simp add: f_def Let_def)
    moreover have "f x = {s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C}"
      using Cons.IH unfolding f_def by blast
    ultimately show ?thesis using c by auto
  next
    case False
    hence "transaction_negchecks_comp (c#C) = transaction_negchecks_comp C" (is ?P)
      using transaction_negchecks_comp.cases[of "c#C" ?P]
```

```
      by force
    thus ?thesis using False Cons.IH by fastforce
  qed
qed simp

lemma transaction_negchecks_comp_notin_fv_empty:
  assumes "x ∉ fv_sst C"
  shows "transaction_negchecks_comp C x = {}"
using assms transaction_negchecks_comp_unfold[of C x] by fastforce

lemma transaction_check_preI[intro]:
  fixes T
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "S ≡ set (unlabel (transaction_selects T))"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes a0: "∀x ∈ set (transaction_fresh T). δ x = {}"
    and a1: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ δ x ∈ set OCC"
    and a2: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
    and a3: "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∈ δ x"
    and a4: "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∉ δ x"
  shows "transaction_check_pre FP TI T δ"
proof -
  let ?P = "λu. is_InSet u ⟶ (
    let x = the_elem_term u; s = the_set_term u
    in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∈ δ (the_Var x))"

  let ?Q = "λu. (is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1) ⟶ (
    let x = fst (hd (the_ins u)); s = snd (hd (the_ins u))
    in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∉ δ (the_Var x))"

  have 1: "?P u" when u: "u ∈ S ∪ C" for u
    apply (unfold Let_def, intro impI, elim conjE)
    using u a3 Fun_Set_InSet_iff[of u] by metis

  have 2: "?Q u" when u: "u ∈ S ∪ C" for u
    apply (unfold Let_def, intro impI, elim conjE)
    using u a4 Fun_Set_NotInSet_iff[of u] by metis

  show ?thesis
    using a0 a1 a2 1 2 fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
    unfolding transaction_check_pre_def ϑ_def S_def C_def Let_def
    by blast
qed

lemma transaction_check_pre_InSetE:
  assumes T: "transaction_check_pre FP TI T δ"
    and u: "u = ⟨a: Var x ∈ Fun (Set s) []⟩"
           "u ∈ set (unlabel (transaction_selects T)) ∪ set (unlabel (transaction_checks T))"
  shows "s ∈ δ x"
proof -
  have "is_InSet u ⟶ is_Var (the_elem_term u) ∧ is_Fun_Set (the_set_term u) ⟶
        the_Set (the_Fun (the_set_term u)) ∈ δ (the_Var (the_elem_term u))"
    using T u unfolding transaction_check_pre_def Let_def by blast
  thus ?thesis using Fun_Set_InSet_iff[of u a x s] u by argo
qed

lemma transaction_check_pre_NotInSetE:
  assumes T: "transaction_check_pre FP TI T δ"
    and u: "u = ⟨Var x not in Fun (Set s) []⟩"
           "u ∈ set (unlabel (transaction_selects T)) ∪ set (unlabel (transaction_checks T))"
  shows "s ∉ δ x"
proof -
  have "is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1 ⟶
```

```
        is_Var (fst (hd (the_ins u))) ∧ is_Fun_Set (snd (hd (the_ins u))) ⟶
        the_Set (the_Fun (snd (hd (the_ins u)))) ∉ δ (the_Var (fst (hd (the_ins u))))"
    using T u unfolding transaction_check_pre_def Let_def by blast
  thus ?thesis using Fun_Set_NotInSet_iff[of u  x s] u by argo
qed

lemma transaction_check_compI[intro]:
  assumes T: "transaction_check_pre FP TI T δ"
    and T_adm: "admissible_transaction T"
    and x1: "∀x. (x ∈ fv_transaction T - set (transaction_fresh T) ∧ fst x = TAtom Value)
                ⟶ δ x ∈ set OCC"
    and x2: "∀x. (x ∉ fv_transaction T - set (transaction_fresh T) ∨ fst x ≠ TAtom Value)
                ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ‘ set (transaction_check_comp FP OCC TI T)"
proof -
  define S where "S ≡ unlabel (transaction_strand T)"
  define C where "C ≡ unlabel (transaction_selects T@transaction_checks T)"
  define C' where "C' ≡ set (unlabel (transaction_selects T)) ∪
                       set (unlabel (transaction_checks T))"

  let ?xs = "fv_list_sst S"

  define poss where "poss ≡ transaction_poschecks_comp C"
  define negs where "negs ≡ transaction_negchecks_comp C"
  define ys where "ys ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) ?xs"

  have C_C'_eq: "set C = C'"
    using unlabel_append[of "transaction_selects T" "transaction_checks T"]
    unfolding C_def C'_def by simp

  have ys: "{x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value} = set ys"
    using fv_list_sst_is_fv_sst[of S]
    unfolding ys_def S_def by force

  have "δ x ∈ set OCC"
    when x: "x ∈ set ys" for x
    using x1 x ys by blast
  moreover have "δ x = {}"
    when x: "x ∉ set ys" for x
    using x2 x ys by blast
  moreover have "poss x ⊆ δ x" when x: "x ∈ set ys" for x
  proof -
    have "s ∈ δ x" when u: "u = ⟨a: Var x ∈ Fun (Set s) []⟩" "u ∈ C'" for u a s
      using T u transaction_check_pre_InSetE[of FP TI T δ]
      unfolding C'_def by blast
    thus ?thesis
      using transaction_poschecks_comp_unfold[of C x] C_C'_eq
      unfolding poss_def by blast
  qed
  moreover have "δ x ∩ negs x = {}" when x: "x ∈ set ys" for x
  proof (cases "x ∈ fv_sst C")
    case True
    hence "s ∉ δ x" when u: "u = ⟨Var x not in Fun (Set s) []⟩" "u ∈ C'" for u s
      using T u transaction_check_pre_NotInSetE[of FP TI T δ]
      unfolding C'_def by blast
    thus ?thesis
      using transaction_negchecks_comp_unfold[of C x] C_C'_eq
      unfolding negs_def by blast
  next
    case False
    hence "negs x = {}"
      using x C_C'_eq transaction_negchecks_comp_notin_fv_empty
      unfolding negs_def by blast
```

```
    thus ?thesis by blast
  qed
  ultimately have "δ ∈ abs_substs_fun ‘ set (abs_substs_set ys OCC poss negs)"
    using abs_substs_has_all_abs[of ys δ OCC poss negs]
    by fast
  thus ?thesis
    using T
    unfolding transaction_check_comp_def Let_def S_def C_def ys_def poss_def negs_def
    by fastforce
qed

context
begin
private lemma transaction_check_comp_in_aux:
  fixes T
  defines "S ≡ set (unlabel (transaction_selects T))"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes T_adm: "admissible_transaction T"
    and a1: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
        select⟨Var x, Fun (Set s) []⟩ ∈ S ⟶ s ∈ α x)"
    and a2: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
        ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x)"
    and a3: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
        ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x)"
  shows "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∈ α x" (is ?A)
    and "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∉ α x" (is ?B)
proof -
  have T_valid: "wellformed_transaction T"
      and T_adm_S: "admissible_transaction_selects T"
      and T_adm_C: "admissible_transaction_checks T"
    using T_adm unfolding admissible_transaction_def by blast+

  note * = admissible_transaction_strand_step_cases(2,3)[OF T_adm]

  have 1: "fst x = TAtom Value" "x ∈ fv_transaction T - set (transaction_fresh T)"
    when x: "⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C" for a x s
    using * x unfolding S_def C_def by fast+

  have 2: "fst x = TAtom Value" "x ∈ fv_transaction T - set (transaction_fresh T)"
    when x: "⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C" for x s
    using * x unfolding S_def C_def by fast+

  have 3: "select⟨Var x, Fun (Set s) []⟩ ∈ S"
    when x: "select⟨Var x, Fun (Set s) []⟩ ∈ S ∪ C" for x s
    using * x unfolding S_def C_def by fast

  have 4: "⟨Var x in Fun (Set s) []⟩ ∈ C"
    when x: "⟨Var x in Fun (Set s) []⟩ ∈ S ∪ C" for x s
    using * x unfolding S_def C_def by fast

  have 5: "⟨Var x not in Fun (Set s) []⟩ ∈ C"
    when x: "⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C" for x s
    using * x unfolding S_def C_def by fast

  show ?A
  proof (intro allI impI)
    fix a x s assume u: "⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C"
    thus "s ∈ α x" using 1 3 4 a1 a2 by (cases a) metis+
  qed

  show ?B
  proof (intro allI impI)
    fix x s assume u: "⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C"
```

```
    thus "s ∉ α x" using 2 5 a3 by meson
  qed
qed

lemma transaction_check_comp_in:
  fixes T
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "S ≡ set (unlabel (transaction_selects T))"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes T_adm: "admissible_transaction T"
    and a1: "∀x ∈ set (transaction_fresh T). α x = {}"
    and a2: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ α)"
    and a3: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          select⟨Var x, Fun (Set s) []⟩ ∈ S ⟶ s ∈ α x"
    and a4: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x"
    and a5: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x"
    and a6: "∀x ∈ fv_transaction T - set (transaction_fresh T).
          fst x = TAtom Value ⟶ α x ∈ set OCC"
  shows "∃δ ∈ abs_substs_fun ‘ set (transaction_check_comp FP OCC TI T). ∀x ∈ fv_transaction T.
          fst x = TAtom Value ⟶ α x = δ x"
proof -
  let ?xs = "fv_list_sst (unlabel (transaction_strand T))"
  let ?ys = "filter (λx. x ∉ set (transaction_fresh T)) ?xs"

  define α' where "α' ≡ λx.
    if x ∈ fv_transaction T - set (transaction_fresh T) ∧ fst x = TAtom Value
    then α x
    else {}"

  have T_valid: "wellformed_transaction T"
    using T_adm unfolding admissible_transaction_def by blast

  have ϑα_Fun: "is_Fun (t · ϑ α) ⟷ is_Fun (t · ϑ α')" for t
    unfolding α'_def ϑ_def
    by (induct t) auto

  have "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ α')"
  proof (intro ballI impI)
    fix t assume t: "t ∈ trms_lsst (transaction_receive T)"

    have 1: "intruder_synth_mod_timpls FP TI (t · ϑ α)"
      using t a2
      by auto

    obtain r where r:
        "r ∈ set (unlabel (transaction_receive T))"
        "t ∈ trms_sstp r"
      using t by auto
    hence "r = receive⟨t⟩"
      using wellformed_transaction_unlabel_cases(1)[OF T_valid]
      by fastforce
    hence 2: "fv t ⊆ fv_lsst (transaction_receive T)" using r by force

    have "fv t ⊆ fv_transaction T"
      by (metis (no_types, lifting) 2 transaction_strand_def sst_vars_append_subset(1)
            unlabel_append subset_Un_eq sup.bounded_iff)
    moreover have "fv t ∩ set (transaction_fresh T) = {}"
      using 2 T_valid vars_sst_is_fv_sst_bvars_sst[of "unlabel (transaction_receive T)"]
      unfolding wellformed_transaction_def
      by fast
    ultimately have "ϑ α x = ϑ α' x" when "x ∈ fv t" for x
```

```
    using that unfolding α'_def ϑ_def by fastforce
  hence 3: "t · ϑ α = t · ϑ α'"
    using term_subst_eq by blast

  show "intruder_synth_mod_timpls FP TI (t · ϑ α')" using 1 3 by simp
qed
moreover have
    "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
      select⟨Var x, Fun (Set s) []⟩ ∈ S ⟶ s ∈ α' x)"
    "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
      ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α' x)"
    "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
      ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α' x)"
  using a3 a4 a5
  unfolding α'_def ϑ_def S_def C_def
  by meson+
hence "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∈ α' x"
      "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∉ α' x"
  using transaction_check_comp_in_aux[OF T_adm, of α']
  unfolding S_def C_def
  by fast+
ultimately have 4: "transaction_check_pre FP TI T α'"
  using a6 transaction_check_preI[of T α' OCC FP TI]
  unfolding α'_def ϑ_def S_def C_def by simp

have 5: "∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ α x = α' x"
  using a1 by (auto simp add: α'_def)

have 6: "α' ∈ abs_substs_fun ‘ set (transaction_check_comp FP OCC TI T)"
  using transaction_check_compI[OF 4 T_adm] a6
  unfolding α'_def
  by auto

show ?thesis using 5 6 by blast
qed
end

end
```

2.6.3 Automatically Checking Protocol Security in a Typed Model

```
context stateful_protocol_model
begin

definition abs_intruder_knowledge ("α_ik") where
  "α_ik S I ≡ (ik_lsst S ·_set I) ·_αset α_0 (db_lsst S I)"

definition abs_value_constants ("α_vals") where
  "α_vals S I ≡ {t ∈ subterms_set (trms_lsst S) ·_set I. ∃n. t = Fun (Val n) []} ·_αset α_0 (db_lsst S I)"

definition abs_term_implications ("α_ti") where
  "α_ti A T σ α I ≡ {(s,t) | s t x.
    s ≠ t ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T) ∧
    Fun (Abs s) [] = (σ ∘_s α) x · I ·_α α_0 (db_lsst A I) ∧
    Fun (Abs t) [] = (σ ∘_s α) x · I ·_α α_0 (db_lsst (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) I)}"

lemma abs_intruder_knowledge_append:
  "α_ik (A@B) I =
    (ik_lsst A ·_set I) ·_αset α_0 (db_lsst (A@B) I) ∪
    (ik_lsst B ·_set I) ·_αset α_0 (db_lsst (A@B) I)"
by (metis unlabel_append abs_set_union image_Un ik_sst_append abs_intruder_knowledge_def)

lemma abs_value_constants_append:
```

```
  fixes A B::"('a,'b,'c,'d) prot_strand"
  shows "α_vals (A@B) ℐ =
      {t ∈ subterms_set (trms_lsst A) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α_0 (db_lsst (A@B) ℐ) ∪
      {t ∈ subterms_set (trms_lsst B) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α_0 (db_lsst (A@B) ℐ)"
proof -
  define a0 where "a0 ≡ α_0 (db_sst (unlabel (A@B)) ℐ)"
  define M where "M ≡ λa::('a,'b,'c,'d) prot_strand.
                          {t ∈ subterms_set (trms_lsst a) ·_set ℐ. ∃n. t = Fun (Val n) []}"

  have "M (A@B) = M A ∪ M B"
    using unlabel_append[of A B] trms_sst_append[of "unlabel A" "unlabel B"]
          image_Un[of "λx. x · ℐ" "subterms_set (trms_lsst A)" "subterms_set (trms_lsst B)"]
    unfolding M_def by force
  hence "M (A@B) ·_αset a0 = (M A ·_αset a0) ∪ (M B ·_αset a0)" by (simp add: abs_set_union)
  thus ?thesis unfolding abs_value_constants_def a0_def M_def by blast
qed

lemma transaction_renaming_subst_has_no_pubconsts_abss:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "subst_range α ∩ pubval_terms = {}" (is ?A)
    and "subst_range α ∩ abs_terms = {}" (is ?B)
proof -
  { fix t assume "t ∈ subst_range α"
    then obtain x where "t = Var x"
      using transaction_renaming_subst_is_renaming[OF assms]
      by force
    hence "t ∉ pubval_terms" "t ∉ abs_terms" by simp_all
  } thus ?A ?B by auto
qed

lemma transaction_fresh_subst_has_no_pubconsts_abss:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
  shows "subst_range σ ∩ pubval_terms = {}" (is ?A)
    and "subst_range σ ∩ abs_terms = {}" (is ?B)
proof -
  { fix t assume "t ∈ subst_range σ"
    then obtain n where "t = Fun (Val (n,False)) []"
      using assms unfolding transaction_fresh_subst_def
      by force
    hence "t ∉ pubval_terms" "t ∉ abs_terms" by simp_all
  } thus ?A ?B by auto
qed

lemma reachable_constraints_no_pubconsts_abss:
  assumes "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. ∀n. Val (n,True) ∉ ⋃(funs_term ` trms_transaction T)"
           "∀T ∈ set P. ∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
           "∀T ∈ set P. bvars_lsst (transaction_strand T) = {}"
    and ℐ: "interpretation_subst ℐ" "wt_subst ℐ" "wf_trms (subst_range ℐ)"
           "∀n. Val (n,True) ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
           "∀n. Abs n ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "trms_lsst 𝒜 ·_set ℐ ⊆ GSMP (⋃T ∈ set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
    (is "?A ⊆ ?B")
using assms(1) ℐ(4,5)
proof (induction 𝒜 rule: reachable_constraints.induct)
  case (step 𝒜 T σ α)
  define trms_P where "trms_P ≡ (⋃T ∈ set P. trms_transaction T)"
  define T' where "T' ≡ transaction_strand T ·_lsst σ ∘_s α"

  have ℐ': "∀n. Val (n,True) ∉ ⋃ (funs_term ` (ℐ ` fv_lsst 𝒜))"
```

```
        "∀n. Abs n ∉ ⋃ (funs_term ` (ℐ ` fv_lsst 𝒜))"
  using step.prems fv_sst_append[of "unlabel 𝒜"] unlabel_append[of 𝒜]
  by auto

have "wt_subst (σ ∘_s α)"
  using transaction_renaming_subst_wt[OF step.hyps(4)]
        transaction_fresh_subst_wt[OF step.hyps(3)]
  by (metis step.hyps(2) P(3) wt_subst_compose)
hence "wt_subst (rm_vars (set X) (σ ∘_s α))" for X
  using wt_subst_rm_vars[of "σ ∘_s α" "set X"]
  by metis
hence wt: "wt_subst ((rm_vars (set X) (σ ∘_s α)) ∘_s ℐ)" for X
  using ℐ(2) wt_subst_compose by fast

have "wf_trms (subst_range (σ ∘_s α))"
  using transaction_fresh_subst_range_wf_trms[OF step.hyps(3)]
        transaction_renaming_subst_range_wf_trms[OF step.hyps(4)]
  by (metis wf_trms_subst_compose)
hence wftrms: "wf_trms (subst_range ((rm_vars (set X) (σ ∘_s α)) ∘_s ℐ))" for X
  using wf_trms_subst_compose[OF wf_trms_subst_rm_vars' ℐ(3)] by fast

have "trms_lsst (dual_lsst T') ·_set ℐ ⊆ ?B"
proof
  fix t assume "t ∈ trms_lsst (dual_lsst T') ·_set ℐ"
  hence "t ∈ trms_lsst T' ·_set ℐ" using trms_sst_unlabel_dual_lsst_eq by blast
  then obtain s X where s:
      "s ∈ trms_transaction T"
      "t = s · rm_vars (set X) (σ ∘_s α) ∘_s ℐ"
      "set X ⊆ bvars_transaction T"
    using trms_sst_unlabel_subst'' unfolding T'_def by blast

  define ϑ where "ϑ ≡ rm_vars (set X) (σ ∘_s α)"

  have 1: "s ∈ trms_P" using step.hyps(2) s(1) unfolding trms_P_def by auto

  have s_nin: "s ∉ pubval_terms" "s ∉ abs_terms"
    using 1 P(1,2) funs_term_Fun_subterm
    unfolding trms_P_def is_Val_def is_Abs_def
    by fastforce+

  have 2: "(ℐ ` fv_lsst (𝒜@dual_lsst T')) ∩ pubval_terms = {}"
          "(ℐ ` fv_lsst (𝒜@dual_lsst T')) ∩ abs_terms = {}"
          "subst_range (σ ∘_s α) ∩ pubval_terms = {}"
          "subst_range (σ ∘_s α) ∩ abs_terms = {}"
          "subst_range ϑ ∩ pubval_terms = {}"
          "subst_range ϑ ∩ abs_terms = {}"
          "(ϑ ` fv s) ∩ pubval_terms = {}"
          "(ϑ ` fv s) ∩ abs_terms = {}"
    unfolding T'_def ϑ_def
    using step.prems funs_term_Fun_subterm
    apply (fastforce simp add: is_Val_def,
           fastforce simp add: is_Abs_def)
    using pubval_terms_subst_range_comp[OF
            transaction_fresh_subst_has_no_pubconsts_abss(1)[OF step.hyps(3)]
            transaction_renaming_subst_has_no_pubconsts_abss(1)[OF step.hyps(4)]]
          abs_terms_subst_range_comp[OF
            transaction_fresh_subst_has_no_pubconsts_abss(2)[OF step.hyps(3)]
            transaction_renaming_subst_has_no_pubconsts_abss(2)[OF step.hyps(4)]]
    unfolding is_Val_def is_Abs_def
    by force+

  have "(ℐ ` fv (s · ϑ)) ∩ pubval_terms = {}"
       "(ℐ ` fv (s · ϑ)) ∩ abs_terms = {}"
```

```
    proof -
      have "ϑ = σ ∘s α" "bvars_transaction T = {}" "vars_lsst T' = fv_lsst T'"
        using s(3) P(4) step.hyps(2) rm_vars_empty
              vars_sst_is_fv_sst_bvars_sst[of "unlabel T'"]
              bvars_sst_subst[of "unlabel (transaction_strand T)" "σ ∘s α"]
              unlabel_subst[of "transaction_strand T" "σ ∘s α"]
        unfolding ϑ_def T'_def by simp_all
      hence "fv (s · ϑ) ⊆ fv_lsst T'"
        using trms_sst_fv_subst_subset[OF s(1), of ϑ] unlabel_subst[of "transaction_strand T" ϑ]
        unfolding T'_def by auto
      moreover have "fv_lsst T' ⊆ fv_lsst (A@dual_lsst T')"
        using fv_sst_append[of "unlabel A" "unlabel (dual_lsst T')"]
              unlabel_append[of A "dual_lsst T'"]
              fv_sst_unlabel_dual_lsst_eq[of T']
        by simp_all
      hence "I ‘ fv_lsst T' ∩ pubval_terms = {}" "I ‘ fv_lsst T' ∩ abs_terms = {}"
        using 2(1,2) by blast+
      ultimately show "(I ‘ fv (s · ϑ)) ∩ pubval_terms = {}" "(I ‘ fv (s · ϑ)) ∩ abs_terms = {}"
        by blast+
    qed
    hence σαI_disj: "((ϑ ∘s I) ‘ fv s) ∩ pubval_terms = {}"
                    "((ϑ ∘s I) ‘ fv s) ∩ abs_terms = {}"
      using pubval_terms_subst_range_comp'[of ϑ "fv s" I]
            abs_terms_subst_range_comp'[of ϑ "fv s" I]
            2(7,8)
      by (simp_all add: subst_apply_fv_unfold)

    have 3: "t ∉ pubval_terms" "t ∉ abs_terms"
      using s(2) s_nin σαI_disj
            pubval_terms_subst[of s "rm_vars (set X) (σ ∘s α) ∘s I"]
            pubval_terms_subst_range_disj[of "rm_vars (set X) (σ ∘s α) ∘s I" s]
            abs_terms_subst[of s "rm_vars (set X) (σ ∘s α) ∘s I"]
            abs_terms_subst_range_disj[of "rm_vars (set X) (σ ∘s α) ∘s I" s]
      unfolding ϑ_def
      by blast+

    have "t ∈ SMP trms_P" "fv t = {}"
      by (metis s(2) SMP.Substitution[OF SMP.MP[OF 1] wt wftrms, of X],
          metis s(2) subst_subst_compose[of s "rm_vars (set X) (σ ∘s α)" I]
                     interpretation_grounds[OF I(1), of "s · rm_vars (set X) (σ ∘s α)"])
    hence 4: "t ∈ GSMP trms_P" unfolding GSMP_def by simp

    show "t ∈ ?B" using 3 4 by (auto simp add: trms_P_def)
  qed
  thus ?case
    using step.IH[OF I'] trms_sst_append[of "unlabel A"] unlabel_append[of A]
          image_Un[of "λx. x · I" "trms_lsst A"]
    by (simp add: T'_def)
qed simp

lemma α_ti_covers_α0_aux:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@dual_lsst (transaction_strand T ·lsst σ ∘s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ subterms_set (trms_lsst A)"
           "t = Fun (Val n) [] ∨ t = Var x"
    and neq:
      "t · I ·α α0 (db_lsst A I) ≠
       t · I ·α α0 (db_lsst (A@dual_lsst (transaction_strand T ·lsst σ ∘s α)) I)"
  shows "∃y ∈ fv_transaction T - set (transaction_fresh T).
```

```
        t · ℐ = (σ ∘ₛ α) y · ℐ ∧ Γᵥ y = TAtom Value"
proof -
  let ?𝒜' = "𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘ₛ α)"
  let ?ℬ = "unlabel (dual_lsst (transaction_strand T))"
  let ?ℬ' = "?ℬ ·_sst σ ∘ₛ α"
  let ?ℬ'' = "unlabel (dual_lsst (transaction_strand T ·_lsst σ ∘ₛ α))"

  have ℐ_interp: "interpretation_subst ℐ"
    and ℐ_wt: "wt_subst ℐ"
    and ℐ_wf: "wf_trms (subst_range ℐ)"
    by (metis ℐ welltyped_constraint_model_def constraint_model_def,
        metis ℐ welltyped_constraint_model_def,
        metis ℐ welltyped_constraint_model_def constraint_model_def)

  have T_adm: "admissible_transaction T"
    using T P(1) by blast
  hence T_valid: "wellformed_transaction T"
    unfolding admissible_transaction_def by blast

  have T_adm_upds: "admissible_transaction_updates T"
    by (metis P(1) T admissible_transaction_def)

  have T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γᵥ x = TAtom Value"
    using T P(1) protocol_transaction_vars_TAtom_typed(3)[of T] P(1) by simp

  have wt_σα: "wt_subst (σ ∘ₛ α)"
    using wt_subst_compose transaction_fresh_subst_wt[OF σ T_fresh_vars_value_typed]
          transaction_renaming_subst_wt[OF α]
    by blast

  have 𝒜_wf_trms: "wf_trms (trms_lsst 𝒜)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) 𝒜_reach)
  hence t_wf: "wf_trm t" using t by auto

  have 𝒜_no_val_bvars: "¬TAtom Value ⊑ Γᵥ x"
    when "x ∈ bvars_lsst 𝒜" for x
    using P(1) reachable_constraints_no_bvars 𝒜_reach
          vars_sst_is_fv_sst_bvars_sst[of "unlabel 𝒜"] that
    unfolding admissible_transaction_def by fast

  have x': "x ∈ vars_lsst 𝒜" when "t = Var x"
    using that t by (simp add: var_subterm_trms_sst_is_vars_sst)

  have "∃f ∈ funs_term (t · ℐ). is_Val f"
    using abs_eq_if_no_Val neq by metis
  hence "∃n T. Fun (Val n) T ⊑ t · ℐ"
    using funs_term_Fun_subterm
    unfolding is_Val_def by fast
  hence "TAtom Value ⊑ Γ (Var x)" when "t = Var x"
    using wt_subst_trm''[OF ℐ_wt, of "Var x"] that
          subtermeq_imp_subtermtypeeq[of "t · ℐ"] wf_trm_subst[OF ℐ_wf, of t] t_wf
    by fastforce
  hence x_val: "Γᵥ x = TAtom Value" when "t = Var x"
    using reachable_constraints_vars_TAtom_typed[OF 𝒜_reach P x'] that
    by fastforce
  hence x_fv: "x ∈ fv_lsst 𝒜" when "t = Var x" using x'
    using reachable_constraints_Value_vars_are_fv[OF 𝒜_reach P x'] that
    by blast
  then obtain m where m: "t · ℐ = Fun (Val m) []"
    using constraint_model_Value_term_is_Val[
            OF 𝒜_reach welltyped_constraint_model_prefix[OF ℐ] P, of x]
          t(2) x_val
    by force
```

```
  hence 0: "α₀ (dbₗₛₛₜ 𝒜 ℐ) m ≠ α₀ (dbₛₛₜ (unlabel 𝒜@?ℬ'') ℐ) m"
    using neq by (simp add: unlabel_def)

  have t_val: "Γ t = TAtom Value" using x_val t by force

  obtain u s where s: "t · ℐ = u · ℐ" "insert⟨u,s⟩ ∈ set ?ℬ' ∨ delete⟨u,s⟩ ∈ set ?ℬ'"
    using to_abs_neq_imp_db_update[OF 0] m
    by (metis (no_types, lifting) dualₗₛₛₜ_subst subst_lsst_unlabel)
  then obtain u' s' where s':
      "u = u' · σ ∘ₛ α" "s = s' · σ ∘ₛ α"
      "insert⟨u',s'⟩ ∈ set ?ℬ ∨ delete⟨u',s'⟩ ∈ set ?ℬ"
    using stateful_strand_step_subst_inv_cases(4,5)
    by blast
  hence s'': "insert⟨u',s'⟩ ∈ set (unlabel (transaction_strand T)) ∨
              delete⟨u',s'⟩ ∈ set (unlabel (transaction_strand T))"
    using dualₗₛₛₜ_unlabel_steps_iff(4,5)[of u' s' "transaction_strand T"]
    by simp_all
  then obtain y where y: "y ∈ fv_transaction T" "u' = Var y"
    using transaction_inserts_are_Value_vars[OF T_valid T_adm_upds, of u' s']
          transaction_deletes_are_Value_vars[OF T_valid T_adm_upds, of u' s']
          stateful_strand_step_fv_subset_cases(4,5)[of u' s' "unlabel (transaction_strand T)"]
    by auto
  hence 1: "t · ℐ = (σ ∘ₛ α) y · ℐ" using y s(1) s'(1) by (metis subst_apply_term.simps(1))

  have 2: "y ∉ set (transaction_fresh T)" when "(σ ∘ₛ α) y · ℐ ≠ σ y"
    using transaction_fresh_subst_grounds_domain[OF σ, of y] subst_compose[of σ α y] that
    by (auto simp add: subst_ground_ident)

  have 3: "y ∉ set (transaction_fresh T)" when "(σ ∘ₛ α) y · ℐ ∈ subtermsₛₑₜ (trmsₗₛₛₜ 𝒜)"
    using 2 that σ unfolding transaction_fresh_subst_def by fastforce

  have 4: "∀x ∈ fvₗₛₛₜ 𝒜. Γᵥ x = TAtom Value ⟶
           (∃B. prefix B 𝒜 ∧ x ∉ fvₗₛₛₜ B ∧ ℐ x ∈ subtermsₛₑₜ (trmsₗₛₛₜ B))"
    by (metis welltyped_constraint_model_prefix[OF ℐ]
              constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach _ P])

  have 5: "Γᵥ y = TAtom Value"
    using 1 t_val
          wt_subst_trm''[OF wt_σα, of "Var y"]
          wt_subst_trm''[OF ℐ_wt, of t]
          wt_subst_trm''[OF ℐ_wt, of "(σ ∘ₛ α) y"]
    by (auto simp del: subst_subst_compose)

  have "y ∉ set (transaction_fresh T)"
  proof (cases "t = Var x")
    case True
    hence *: "ℐ x = Fun (Val m) []" "x ∈ fvₗₛₛₜ 𝒜" "ℐ x = (σ ∘ₛ α) y · ℐ"
      using m t(1) 1 x_fv x' by (force, blast, force)

    obtain B where B: "prefix B 𝒜" "ℐ x ∈ subtermsₛₑₜ (trmsₗₛₛₜ B)"
      using *(2) 4 x_val[OF True] by fastforce
    hence "∀t ∈ subst_range σ. t ∉ subtermsₛₑₜ (trmsₗₛₛₜ B)"
      using transaction_fresh_subst_range_fresh(1)[OF σ] trmsₛₛₜ_unlabel_prefix_subset(1)[of B]
      unfolding prefix_def by fast
    thus ?thesis using *(1,3) B(2) 2 by (metis subst_imgI term.distinct(1))
  next
    case False
    hence "t · ℐ ∈ subtermsₛₑₜ (trmsₗₛₛₜ 𝒜)" using t by simp
    thus ?thesis using 1 3 by argo
  qed
  thus ?thesis using 1 5 y(1) by fast
qed
```

```
lemma α_ti_covers_α_0_Var:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "x ∈ fv_lsst A"
  shows "I x ·_α α_0 (db_lsst (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) I) ∈
          timpl_closure_set {I x ·_α α_0 (db_lsst A I)} (α_ti A T σ α I)"
proof -
  define a0 where "a0 ≡ α_0 (db_lsst A I)"
  define a0' where "a0' ≡ α_0 (db_lsst (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) I)"
  define a3 where "a3 ≡ α_ti A T σ α I"

  have A_wf_trms: "wf_trms (trms_lsst A)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) A_reach)

  have T_adm: "admissible_transaction T" by (metis P(1) T)

  have I_interp: "interpretation_subst I"
    and I_wt: "wt_subst I"
    and I_wf_trms: "wf_trms (subst_range I)"
    by (metis I welltyped_constraint_model_def constraint_model_def,
        metis I welltyped_constraint_model_def,
        metis I welltyped_constraint_model_def constraint_model_def)

  have "Γ_v x = Var Value ∨ (∃a. Γ_v x = Var (prot_atom.Atom a))"
    using reachable_constraints_vars_TAtom_typed[OF A_reach P, of x]
          x vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
    by auto

  hence "I x ·_α a0' ∈ timpl_closure_set {I x ·_α a0} a3"
  proof
    assume x_val: "Γ_v x = TAtom Value"
    show "I x ·_α a0' ∈ timpl_closure_set {I x ·_α a0} a3"
    proof (cases "I x ·_α a0 = I x ·_α a0'")
      case False
      hence "∃y ∈ fv_transaction T - set (transaction_fresh T).
             I x = (σ ∘_s α) y · I ∧ Γ_v y = TAtom Value"
        using α_ti_covers_α_0_aux[OF A_reach T I σ α P fv_sst_is_subterm_trms_sst[OF x], of _ x]
        unfolding a0_def a0'_def
        by fastforce
      then obtain y where y:
          "y ∈ fv_transaction T - set (transaction_fresh T)"
          "I x = (σ ∘_s α) y · I"
          "I x ·_α a0 = (σ ∘_s α) y · I ·_α a0"
          "I x ·_α a0' = (σ ∘_s α) y · I ·_α a0'"
          "Γ_v y = TAtom Value"
        by metis
      then obtain n where n: "(σ ∘_s α) y · I = Fun (Val (n,False)) []"
        using Γ_v_TAtom''(2)[of y] x x_val
              transaction_var_becomes_Val[
                OF reachable_constraints.step[OF A_reach T σ α] I σ α P T, of y]
        by force

      have "a0 (n,False) ≠ a0' (n,False)"
           "y ∈ fv_transaction T"
           "y ∉ set (transaction_fresh T)"
           "absc (a0 (n,False)) = (σ ∘_s α) y · I ·_α a0"
           "absc (a0' (n,False)) = (σ ∘_s α) y · I ·_α a0'"
        using y n False by force+
      hence 1: "(a0 (n,False), a0' (n,False)) ∈ a3"
```

```
      unfolding a0_def a0'_def a3_def abs_term_implications_def
      by blast

    have 2: "ℐ x ·α a0' ∈ set ⟨a0 (n,False) -->> a0' (n,False)⟩⟨ℐ x ·α a0⟩"
      using y n timpl_apply_const by auto

    show ?thesis
      using timpl_closure.TI[OF timpl_closure.FP 1] 2
            term_variants_pred_iff_in_term_variants[
              of "(λ_. [])(Abs (a0 (n, False)) := [Abs (a0' (n, False))])"]
      unfolding timpl_closure_set_def timpl_apply_term_def
      by auto
  qed (auto intro: timpl_closure_setI)
next
  assume "∃a. Γv x = TAtom (Atom a)"
  then obtain a where x_atom: "Γv x = TAtom (Atom a)" by moura

  obtain f T where fT: "ℐ x = Fun f T"
    using interpretation_grounds[OF ℐ_interp, of "Var x"]
    by (cases "ℐ x") auto

  have fT_atom: "Γ (Fun f T) = TAtom (Atom a)"
    using wt_subst_trm''[OF ℐ_wt, of "Var x"] x_atom fT
    by simp

  have T: "T = []"
    using fT wf_trm_subst[OF ℐ_wfₜᵣₘₛ, of "Var x"] const_type_inv_wf[OF fT_atom]
    by fastforce

  have f: "¬is_Val f" using fT_atom unfolding is_Val_def by auto

  have "ℐ x ·α b = ℐ x" for b
    using T fT abs_term_apply_const(2)[OF f]
    by auto
  thus "ℐ x ·α a0' ∈ timpl_closure_set {ℐ x ·α a0} a3"
    by (auto intro: timpl_closure_setI)
qed
thus ?thesis by (metis a0_def a0'_def a3_def)
qed

lemma αₜᵢ_covers_α₀_Val:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dualₗₛₛₜ (transaction_strand T ·ₗₛₛₜ σ ∘ₛ α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and n: "Fun (Val n) [] ∈ subtermsₛₑₜ (trmsₗₛₛₜ 𝒜)"
  shows "Fun (Val n) [] ·α α₀ (dbₗₛₛₜ (𝒜@dualₗₛₛₜ (transaction_strand T ·ₗₛₛₜ σ ∘ₛ α)) ℐ) ∈
          timpl_closure_set {Fun (Val n) [] ·α α₀ (dbₗₛₛₜ 𝒜 ℐ)} (αₜᵢ 𝒜 T σ α ℐ)"
proof -
  define T' where "T' ≡ dualₗₛₛₜ (transaction_strand T ·ₗₛₛₜ σ ∘ₛ α)"
  define a0 where "a0 ≡ α₀ (dbₗₛₛₜ 𝒜 ℐ)"
  define a0' where "a0' ≡ α₀ (dbₗₛₛₜ (𝒜@T') ℐ)"
  define a3 where "a3 ≡ αₜᵢ 𝒜 T σ α ℐ"

  have 𝒜_wfₜᵣₘₛ: "wfₜᵣₘₛ (trmsₗₛₛₜ 𝒜)"
    by (metis reachable_constraints_wfₜᵣₘₛ admissible_transactions_wfₜᵣₘₛ P(1) 𝒜_reach)

  have T_adm: "admissible_transaction T" by (metis P(1) T)

  have "Fun (Abs (a0' n)) [] ∈ timpl_closure_set {Fun (Abs (a0 n)) []} a3"
  proof (cases "a0 n = a0' n")
```

```
    case False
    then obtain x where x:
        "x ∈ fv_transaction T - set (transaction_fresh T)" "Fun (Val n) [] = (σ ∘s α) x ·  I"
      using αti_covers_α0_aux[OF A_reach T I σ α P n]
      by (fastforce simp add: a0_def a0'_def T'_def)
    hence "absc (a0 n) = (σ ∘s α) x · I ·α a0" "absc (a0' n) = (σ ∘s α) x · I ·α a0'" by simp_all
    hence 1: "(a0 n, a0' n) ∈ a3"
      using False x(1)
      unfolding a0_def a0'_def a3_def abs_term_implications_def T'_def
      by blast
    show ?thesis
      using timpl_apply_Abs[of "[]" "[]" "a0 n" "a0' n"]
            timpl_closure.TI[OF timpl_closure.FP[of "Fun (Abs (a0 n)) []" a3] 1]
            term_variants_pred_iff_in_term_variants[of "(λ_. [])(Abs (a0 n) := [Abs (a0' n)])"]
      unfolding timpl_closure_set_def timpl_apply_term_def
      by force
  qed (auto intro: timpl_closure_setI)
  thus ?thesis by (simp add: a0_def a0'_def a3_def T'_def)
qed

lemma αti_covers_α0_ik:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@dual_lsst (transaction_strand T ·lsst σ ∘s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ ik_lsst A"
  shows "t · I ·α α0 (db_lsst (A@dual_lsst (transaction_strand T ·lsst σ ∘s α)) I) ∈
          timpl_closure_set {t · I ·α α0 (db_lsst A I)} (αti A T σ α I)"
proof -
  define a0 where "a0 ≡ α0 (db_lsst A I)"
  define a0' where "a0' ≡ α0 (db_lsst (A@dual_lsst (transaction_strand T ·lsst σ ∘s α)) I)"
  define a3 where "a3 ≡ αti A T σ α I"

  let ?U = "λT a. map (λs. s · I ·α a) T"

  have A_wf_trms: "wf_trms (trms_lsst A)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) A_reach)

  have T_adm: "admissible_transaction T" by (metis P(1) T)

  have "t ∈ subterms_set (ik_lsst A)" "wf_trm t" using A_wf_trms t ik_sst_trms_sst_subset by force+
  hence "∀t0 ∈ subterms t. t0 · I ·α a0' ∈ timpl_closure_set {t0 · I ·α a0} a3"
  proof (induction t)
    case (Var x) thus ?case
      using αti_covers_α0_Var[OF A_reach T I σ α P, of x]
            ik_sst_var_is_fv[of x "unlabel A"] vars_sst_is_fv_sst_bvars_sst[of "unlabel A"]
      by (simp add: a0_def a0'_def a3_def)
  next
    case (Fun f S)
    have IH: "∀t0 ∈ subterms t. t0 · I ·α a0' ∈ timpl_closure_set {t0 · I ·α a0} a3"
      when "t ∈ set S" for t
      using that Fun.prems(1) wf_trm_param[OF Fun.prems(2)] Fun.IH
      by (meson in_subterms_subset_Union params_subterms subsetCE)
    hence "t ·α a0' ∈ timpl_closure_set {t ·α a0} a3"
      when "t ∈ set (map (λs. s · I) S)" for t
      using that by auto
    hence "t ·α a0' ∈ timpl_closure (t ·α a0) a3"
      when "t ∈ set (map (λs. s · I) S)" for t
      using that timpl_closureton_is_timpl_closure by auto
    hence "(t ·α a0, t ·α a0') ∈ timpl_closure' a3"
      when "t ∈ set (map (λs. s · I) S)" for t
```

```
      using that timpl_closure_is_timpl_closure' by auto
    hence IH': "((?U S a0) ! i, (?U S a0') ! i) ∈ timpl_closure' a3"
      when "i < length (map (λs. s ·ℐ ·α a0) S)" for i
      using that by auto

    show ?case
    proof (cases "∃n. f = Val n")
      case True
      then obtain n where "Fun f S = Fun (Val n) []"
        using Fun.prems(2) unfolding wf_trm_def by force
      moreover have "Fun f S ∈ subterms_set (trms_lsst 𝒜)"
        using ik_sst_trms_sst_subset Fun.prems(1) by blast
      ultimately show ?thesis
        using α_ti_covers_α_0_Val[OF 𝒜_reach T ℐ σ α P]
        by (simp add: a0_def a0'_def a3_def)
    next
      case False
      hence "Fun f S ·ℐ ·α a = Fun f (map (λt. t ·ℐ ·α a) S)" for a by (cases f) simp_all
      hence "(Fun f S ·ℐ ·α a0, Fun f S ·ℐ ·α a0') ∈ timpl_closure' a3"
        using timpl_closure_FunI[OF IH']
        by simp
      hence "Fun f S ·ℐ ·α a0' ∈ timpl_closure_set {Fun f S ·ℐ ·α a0} a3"
        using timpl_closureton_is_timpl_closure
              timpl_closure_is_timpl_closure'
        by metis
      thus ?thesis using IH by simp
    qed
  qed
  thus ?thesis by (simp add: a0_def a0'_def a3_def)
qed

lemma transaction_prop1:
  assumes "δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T)"
    and "x ∈ fv_transaction T"
    and "x ∉ set (transaction_fresh T)"
    and "δ x ≠ absdbupd (unlabel (transaction_updates T)) x (δ x)"
    and "transaction_check FP OCC TI T"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "(δ x, absdbupd (unlabel (transaction_updates T)) x (δ x)) ∈ (set TI)⁺"
proof -
  let ?upd = "λx. absdbupd (unlabel (transaction_updates T)) x (δ x)"

  have 0: "fv_transaction T = set (fv_list_sst (unlabel (transaction_strand T)))"
    by (metis fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"])

  have 1: "transaction_check_post FP TI T δ"
    using assms(1,5)
    unfolding transaction_check_def list_all_iff
    by blast

  have "(δ x, ?upd x) ∈ set TI ⟷ (δ x, ?upd x) ∈ (set TI)⁺"
    using TI using assms(4) by blast
  thus ?thesis
    using assms(2,3,4) 0 1 in_trancl_closure_iff_in_trancl_fun[of _ _ TI]
    unfolding transaction_check_post_def List.member_def
    by (metis (no_types, lifting) DiffI)
qed

lemma transaction_prop2:
  assumes δ: "δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T)"
    and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
    and T_check: "transaction_check FP OCC TI T"
```

```
    and T_adm: "admissible_transaction T"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "x ∉ set (transaction_fresh T) ⟹ δ x ∈ set OCC" (is "?A' ⟹ ?A")
    and "absdbupd (unlabel (transaction_updates T)) x (δ x) ∈ set OCC" (is ?B)
proof -
  let ?xs = "fv_list_sst (unlabel (transaction_strand T))"
  let ?ys = "filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) ?xs"
  let ?C = "unlabel (transaction_selects T@transaction_checks T)"
  let ?poss = "transaction_poschecks_comp ?C"
  let ?negs = "transaction_negchecks_comp ?C"
  let ?δupd = "λy. absdbupd (unlabel (transaction_updates T)) y (δ y)"

  have T_wf: "wellformed_transaction T"
    and T_occ: "admissible_transaction_occurs_checks T"
    using T_adm by (metis admissible_transaction_def)+

  have 0: "{x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value} = set ?ys"
    using fv_list_sst_is_fv_sst[of "unlabel (transaction_strand T)"]
    by force

  have 1: "transaction_check_pre FP TI T δ"
    using δ unfolding transaction_check_comp_def Let_def by fastforce

  have 2: "transaction_check_post FP TI T δ"
    using δ T_check unfolding transaction_check_def list_all_iff by blast

  have 3: "δ ∈ abs_substs_fun ‘ set (abs_substs_set ?ys OCC ?poss ?negs)"
    using δ unfolding transaction_check_comp_def Let_def by force

  show A: ?A when ?A' using that 0 3 x abs_substs_abss_bounded by blast

  have 4: "x ∈ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
    when x': "x ∈ set (transaction_fresh T)"
    using T_wf x' unfolding wellformed_transaction_def by fast

  have "intruder_synth_mod_timpls FP TI (occurs (absc (?δupd x)))"
    when x': "x ∈ set (transaction_fresh T)"
    using 2 x' x T_occ
    unfolding transaction_check_post_def admissible_transaction_occurs_checks_def
    by fastforce
  hence "timpl_closure_set (set FP) (set TI) ⊢_c occurs (absc (?δupd x))"
    when x': "x ∈ set (transaction_fresh T)"
    using x' intruder_synth_mod_timpls_is_synth_timpl_closure_set[
          OF TI, of FP "occurs (absc (?δupd x))"]
    by argo
  hence "Abs (?δupd x) ∈ ⋃(funs_term ‘ timpl_closure_set (set FP) (set TI))"
    when x': "x ∈ set (transaction_fresh T)"
    using x' ideduct_synth_priv_fun_in_ik[
          of "timpl_closure_set (set FP) (set TI)" "occurs (absc (?δupd x))"]
    by simp
  hence "∃t ∈ timpl_closure_set (set FP) (set TI). Abs (?δupd x) ∈ funs_term t"
    when x': "x ∈ set (transaction_fresh T)"
    using x' by force
  hence 5: "?δupd x ∈ set OCC" when x': "x ∈ set (transaction_fresh T)"
    using x' OCC by fastforce
```

```
  have 6: "?δupd x ∈ set OCC" when x': "x ∉ set (transaction_fresh T)"
  proof (cases "δ x = ?δupd x")
    case False
    hence "(δ x, ?δupd x) ∈ (set TI)⁺" "δ x ∈ set OCC"
      using A 2 x' x TI
      unfolding transaction_check_post_def fv_list_sst_is_fv_sst Let_def
                in_trancl_closure_iff_in_trancl_fun[symmetric]
                List.member_def
      by blast+
    thus ?thesis using timpl_closure_set_absc_subset_in[OF OCC(2)] by blast
  qed (simp add: A x' x(1))

  show ?B by (metis 5 6)
qed

lemma transaction_prop3:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik A I. timpl_closure_set (set FP) (set TI) ⊢_c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
      "α_vals A I ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
  shows "∀x ∈ set (transaction_fresh T). (σ ∘_s α) x · I ·_α α0 (db_lsst A I) = absc {}" (is ?A)
    and "∀t ∈ trms_lsst (transaction_receive T).
          intruder_synth_mod_timpls FP TI (t · (σ ∘_s α) · I ·_α α0 (db_lsst A I))" (is ?B)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. select⟨Var x,Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))
                ⟶ (∃ss. (σ ∘_s α) x · I ·_α α0 (db_lsst A I) = absc ss ∧ s ∈ ss)" (is ?C)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. ⟨Var x in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
                ⟶ (∃ss. (σ ∘_s α) x · I ·_α α0 (db_lsst A I) = absc ss ∧ s ∈ ss)" (is ?D)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. ⟨Var x not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
                ⟶ (∃ss. (σ ∘_s α) x · I ·_α α0 (db_lsst A I) = absc ss ∧ s ∉ ss)" (is ?E)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T). Γ_v x = TAtom Value ⟶
         (σ ∘_s α) x · I ·_α α0 (db_lsst A I) ∈ absc ‘ set OCC" (is ?F)
proof -
  let ?T' = "dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

  define a0 where "a0 ≡ α0 (db_lsst A I)"
  define a0' where "a0' ≡ α0 (db_lsst (A@?T') I)"
  define fv_AT' where "fv_AT' ≡ fv_lsst (A@?T')"

  have T_adm: "admissible_transaction T"
    using T P(1) by blast
  hence T_valid: "wellformed_transaction T"
    unfolding admissible_transaction_def by blast

  have T_adm':
      "admissible_transaction_selects T"
      "admissible_transaction_checks T"
      "admissible_transaction_updates T"
```

```
  using T_adm unfolding admissible_transaction_def by simp_all

have 𝓘': "interpretation_subst 𝓘" "wt_subst 𝓘" "wf_trms (subst_range 𝓘)"
        "∀n. Val (n,True) ∉ ⋃(funs_term ` (𝓘 ` fv_lsst 𝒜))"
        "∀n. Abs n ∉ ⋃(funs_term ` (𝓘 ` fv_lsst 𝒜))"
        "∀n. Val (n,True) ∉ ⋃(funs_term ` (𝓘 ` fv_AT'))"
        "∀n. Abs n ∉ ⋃(funs_term ` (𝓘 ` fv_AT'))"
  using 𝓘 admissible_transaction_occurs_checks_prop'[
         OF 𝒜_reach welltyped_constraint_model_prefix[OF 𝓘] P]
       admissible_transaction_occurs_checks_prop'[
         OF reachable_constraints.step[OF 𝒜_reach T σ α] 𝓘 P]
  unfolding welltyped_constraint_model_def constraint_model_def is_Val_def is_Abs_def fv_AT'_def
  by fastforce+

have 𝒫': "∀T ∈ set P. ∀n. Val (n,True) ∉ ⋃(funs_term ` trms_transaction T)"
        "∀T ∈ set P. ∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
        "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
  and "∀T ∈ set P. ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
  using protocol_transaction_vars_TAtom_typed
       protocol_transactions_no_pubconsts
       protocol_transactions_no_abss
       funs_term_Fun_subterm P
  by fast+
hence T_no_pubconsts: "∀n. Val (n,True) ∉ ⋃(funs_term ` trms_transaction T)"
  and T_no_abss: "∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
  and T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
  and T_fv_const_typed: "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
  using T by simp_all

have wt_σα𝓘: "wt_subst (σ ∘_s α ∘_s 𝓘)"
    using 𝓘'(2) wt_subst_compose transaction_fresh_subst_wt[OF σ T_fresh_vars_value_typed]
          transaction_renaming_subst_wt[OF α]
    by blast

have 1: "(σ ∘_s α) y · 𝓘 = σ y" when "y ∈ set (transaction_fresh T)" for y
  using transaction_fresh_subst_grounds_domain[OF σ that] subst_compose[of σ α y]
  by (simp add: subst_ground_ident)

have 2: "(σ ∘_s α) y · 𝓘 ∉ subterms_set (trms_lsst 𝒜)" when "y ∈ set (transaction_fresh T)" for y
  using 1[OF that] that σ unfolding transaction_fresh_subst_def by auto

have 3: "∀x ∈ fv_lsst 𝒜. Γ_v x = TAtom Value ⟶
         (∃B. prefix B 𝒜 ∧ x ∉ fv_lsst B ∧ 𝓘 x ∈ subterms_set (trms_lsst B))"
  by (metis welltyped_constraint_model_prefix[OF 𝓘]
            constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach _ P])

have 4: "∃n. (σ ∘_s α) y · 𝓘 = Fun (Val n) []"
  when "y ∈ fv_transaction T" "Γ_v y = TAtom Value" for y
  using transaction_var_becomes_Val[OF reachable_constraints.step[OF 𝒜_reach T σ α] 𝓘 σ α P T]
        that T_fv_const_typed Γ_v_TAtom''[of y]
  by metis

have 𝓘_is_T_model: "strand_sem_stateful (ik_lsst 𝒜 ·_set 𝓘) (set (db_lsst 𝒜 𝓘)) (unlabel ?T') 𝓘"
  using 𝓘 unlabel_append[of 𝒜 ?T'] db_sst_set_is_dbupd_sst[of "unlabel 𝒜" 𝓘 "[]"]
        strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel ?T'" 𝓘]
  by (simp add: welltyped_constraint_model_def constraint_model_def db_sst_def)

have T_rcv_no_val_bvars: "bvars_lsst (transaction_receive T) ∩ subst_domain (σ ∘_s α) = {}"
  using transaction_no_bvars[OF T_adm] bvars_transaction_unfold[of T] by blast

show ?A
proof
  fix y assume y: "y ∈ set (transaction_fresh T)"
```

```
  then obtain yn where yn: "(σ ∘s α) y · I = Fun (Val yn) []" "Fun (Val yn) [] ∈ subst_range σ"
    by (metis transaction_fresh_subst_sends_to_val'[OF σ])

  { — since y is fresh (σ ∘s α) y · I cannot be part of the database state of I A
    fix t' s assume t': "insert⟨t',s⟩ ∈ set (unlabel A)" "t' · I = Fun (Val yn) []"
    then obtain z where t'_z: "t' = Var z" using 2[OF y] yn(1) by (cases t') auto
    hence z: "z ∈ fv_lsst A" "I z = (σ ∘s α) y · I" using t' yn(1) by force+
    hence z': "Γv z = TAtom Value"
      by (metis Γ.simps(1) Γ_consts_simps(2) t'(2) t'_z wt_subst_trm'' I'(2))

    obtain B where B: "prefix B A" "I z ∈ subterms_set (trms_lsst B)" using z z' 3 by fastforce
    hence "∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst B)"
      using transaction_fresh_subst_range_fresh(1)[OF σ] trms_sst_unlabel_prefix_subset(1)[of B]
      unfolding prefix_def by fast
    hence False using B(2) 1[OF y] z yn(1) by (metis subst_imgI term.distinct(1))
  } hence "∄s. ((σ ∘s α) y · I, s) ∈ set (db_lsst A I)"
    using db_sst_in_cases[of "(σ ∘s α) y · I" _ "unlabel A" I "[]"] yn(1)
    by (force simp add: db_sst_def)
  thus "(σ ∘s α) y · I ·α α0 (db_lsst A I) = absc {}"
    using to_abs_empty_iff_notin_db[of yn "db'_lsst A I []"] yn(1)
    by (simp add: db_sst_def)
qed

show receives_covered: ?B
proof
  fix t assume t: "t ∈ trms_lsst (transaction_receive T)"
  hence t_in_T: "t ∈ trms_transaction T"
    using trms_sst_unlabel_prefix_subset(1)[of "transaction_receive T"]
    unfolding transaction_strand_def by fast

  have t_rcv: "receive⟨t · σ ∘s α⟩ ∈ set (unlabel (transaction_receive T ·lsst σ ∘s α))"
    using subst_lsst_unlabel_member[of "receive⟨t⟩" "transaction_receive T" "σ ∘s α"]
          wellformed_transaction_unlabel_cases(1)[OF T_valid] trms_sst_in[OF t]
    by fastforce
  hence *: "ik_lsst A ·set I ⊢ t · σ ∘s α · I"
    using wellformed_transaction_sem_receives[OF T_valid I_is_T_model]
    by simp

  have t_fv: "fv (t · σ ∘s α) ⊆ fv_AT'"
    using fv_sst_append[of "unlabel A"] unlabel_append[of A]
          fv_sst_unlabel_dual_lsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
          t_rcv fv_transaction_subst_unfold[of T " σ ∘s α"]
    unfolding fv_AT'_def by force

  have **: "∀t ∈ (ik_lsst A ·set I) ·αset a0. timpl_closure_set (set FP) (set TI) ⊢c t"
    using FP(3) by (auto simp add: a0_def abs_intruder_knowledge_def)

  note lms1 = pubval_terms_subst[OF _ pubval_terms_subst_range_disj[
                OF transaction_fresh_subst_has_no_pubconsts_abss(1)[OF σ], of t]]
              pubval_terms_subst[OF _ pubval_terms_subst_range_disj[
                OF transaction_renaming_subst_has_no_pubconsts_abss(1)[OF α], of "t · σ"]]

  note lms2 = abs_terms_subst[OF _ abs_terms_subst_range_disj[
                OF transaction_fresh_subst_has_no_pubconsts_abss(2)[OF σ], of t]]
              abs_terms_subst[OF _ abs_terms_subst_range_disj[
                OF transaction_renaming_subst_has_no_pubconsts_abss(2)[OF α], of "t · σ"]]

  have "t ∈ (⋃T∈set P. trms_transaction T)" "fv (t · σ ∘s α · I) = {}"
    using t_in_T T interpretation_grounds[OF I'(1)] by fast+
  moreover have "wf_trms (subst_range (σ ∘s α ∘s I))"
    using wf_trm_subst_rangeI[of σ, OF transaction_fresh_subst_is_wf_trm[OF σ]]
          wf_trm_subst_rangeI[of α, OF transaction_renaming_subst_is_wf_trm[OF α]]
          wf_trms_subst_compose[of σ α, THEN wf_trms_subst_compose[OF _ I'(3)]]
```

```
    by blast
  moreover
  have "t ∉ pubval_terms"
    using t_in_T T_no_pubconsts funs_term_Fun_subterm
    unfolding is_Val_def by fastforce
  hence "t · σ ∘s α ∉ pubval_terms"
    using lms1
    by auto
  hence "t · σ ∘s α · I ∉ pubval_terms"
    using I'(6) t_fv pubval_terms_subst'[of "t · σ ∘s α" I]
    by auto
  moreover have "t ∉ abs_terms"
    using t_in_T T_no_abss funs_term_Fun_subterm
    unfolding is_Abs_def by force
  hence "t · σ ∘s α ∉ abs_terms"
    using lms2
    by auto
  hence "t · σ ∘s α · I ∉ abs_terms"
    using I'(7) t_fv abs_terms_subst'[of "t · σ ∘s α" I]
    by auto
  ultimately have ***:
      "t · σ ∘s α · I ∈ GSMP (⋃T∈set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
    using SMP.Substitution[OF SMP.MP[of t "⋃T∈set P. trms_transaction T"], of "σ ∘s α ∘s I"]
          subst_subst_compose[of t "σ ∘s α" I] wt_σαI
    unfolding GSMP_def by fastforce

  have "∀T∈set P. bvars_transaction T = {}"
    using transaction_no_bvars P unfolding list_all_iff by blast
  hence ****:
      "ik_lsst A ·set I ⊆ GSMP (⋃T∈set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
    using reachable_constraints_no_pubconsts_abss[OF A_reach P' _ I'(1,2,3,4,5)]
          ik_sst_trms_sst_subset[of "unlabel A"]
    by blast

  show "intruder_synth_mod_timpls FP TI (t · σ ∘s α · I ·α α0 (db_lsst A I))"
    using deduct_FP_if_deduct[OF **** ** * ***] deducts_eq_if_analyzed[OF FP(1)]
          intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI, of FP]
    unfolding a0_def by force
qed

show ?C
proof (intro ballI allI impI)
  fix y s
  assume y: "y ∈ fv_transaction T - set (transaction_fresh T)"
     and s: "select⟨Var y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))"
  hence "select⟨Var y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_strand T))"
    unfolding transaction_strand_def unlabel_def by auto
  hence y_val: "Γv y = TAtom Value"
    using transaction_selects_are_Value_vars[OF T_valid T_adm'(1)]
    by fastforce

  have "select⟨(σ ∘s α) y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T ·lsst (σ ∘s α)))"
    using subst_lsst_unlabel_member[OF s]
    by fastforce
  hence "((σ ∘s α) y · I, Fun (Set s) []) ∈ set (db_lsst A I)"
    using wellformed_transaction_sem_selects[
            OF T_valid I_is_T_model,
            of "(σ ∘s α) y" "Fun (Set s) []"]
    by simp
  thus "∃ss. (σ ∘s α) y · I ·α α0 (db_lsst A I) = absc ss ∧ s ∈ ss"
    using to_abs_alt_def[of "db_lsst A I"] 4[of y] y y_val by auto
qed
```

```
show ?D
proof (intro ballI allI impI)
  fix y s
  assume y: "y ∈ fv_transaction T - set (transaction_fresh T)"
     and s: "⟨Var y in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
  hence "⟨Var y in Fun (Set s) []⟩ ∈ set (unlabel (transaction_strand T))"
    unfolding transaction_strand_def unlabel_def by auto
  hence y_val: "Γv y = TAtom Value"
    using transaction_inset_checks_are_Value_vars[OF T_valid T_adm'(2)]
    by fastforce

  have "⟨(σ ∘s α) y in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T ·lsst (σ ∘s α)))"
    using subst_lsst_unlabel_member[OF s]
    by fastforce
  hence "((σ ∘s α) y · I, Fun (Set s) []) ∈ set (dblsst A I)"
    using wellformed_transaction_sem_pos_checks[
            OF T_valid I_is_T_model,
            of "(σ ∘s α) y" "Fun (Set s) []"]
    by simp
  thus "∃ss. (σ ∘s α) y · I ·α α0 (dblsst A I) = absc ss ∧ s ∈ ss"
    using to_abs_alt_def[of "dblsst A I"] 4[of y] y y_val by auto
qed

show ?E
proof (intro ballI allI impI)
  fix y s
  assume y: "y ∈ fv_transaction T - set (transaction_fresh T)"
     and s: "⟨Var y not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))"
  hence "⟨Var y not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_strand T))"
    unfolding transaction_strand_def unlabel_def by auto
  hence y_val: "Γv y = TAtom Value"
    using transaction_notinset_checks_are_Value_vars[OF T_valid T_adm'(2)]
    by fastforce

  have "⟨(σ ∘s α) y not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T ·lsst (σ ∘s α)))"
    using subst_lsst_unlabel_member[OF s]
    by fastforce
  hence "((σ ∘s α) y · I, Fun (Set s) []) ∉ set (dblsst A I)"
    using wellformed_transaction_sem_neg_checks(2)[
            OF T_valid I_is_T_model,
            of "[]" "(σ ∘s α) y" "Fun (Set s) []"]
    by simp
  moreover have "list_all admissible_transaction_updates P"
    using Ball_set[of P "admissible_transaction"] P(1)
          Ball_set[of P admissible_transaction_updates]
    unfolding admissible_transaction_def
    by fast
  moreover have "list_all wellformed_transaction P"
    using P(1) Ball_set[of P "admissible_transaction"] Ball_set[of P wellformed_transaction]
    unfolding admissible_transaction_def
    by blast
  ultimately have "((σ ∘s α) y · I, Fun (Set s) S) ∉ set (dblsst A I)" for S
    using reachable_constraints_dblsst_set_args_empty[OF A_reach]
    unfolding admissible_transaction_updates_def
    by auto
  thus "∃ss. (σ ∘s α) y · I ·α α0 (dblsst A I) = absc ss ∧ s ∉ ss"
    using to_abs_alt_def[of "dblsst A I"] 4[of y] y y_val by auto
qed

show ?F
proof (intro ballI impI)
  fix y assume y: "y ∈ fv_transaction T - set (transaction_fresh T)" "Γv y = TAtom Value"
  then obtain yn where yn: "(σ ∘s α) y · I = Fun (Val yn) []" using 4 by moura
```

```
    hence y_abs: "(σ ∘s α) y · 𝓘 ·α α0 (dblsst 𝒜 𝓘) = Fun (Abs (α0 (dblsst 𝒜 𝓘) yn)) []" by simp

    have *: "∀r ∈ set (unlabel (transaction_selects T)). ∃x s. r = select⟨Var x, Fun (Set s) []⟩"
      using admissible_transaction_strand_step_cases(2)[OF T_adm] by fast

    have "y ∈ fvlsst (transaction_receive T) ∨ y ∈ fvlsst (transaction_selects T)"
      using wellformed_transaction_fv_in_receives_or_selects[OF T_valid] y by blast
    thus "(σ ∘s α) y · 𝓘 ·α α0 (dblsst 𝒜 𝓘) ∈ absc ‘ set OCC"
    proof
      assume "y ∈ fvlsst (transaction_receive T)"
      then obtain t where t: "receive⟨t⟩ ∈ set (unlabel (transaction_receive T))" "y ∈ fv t"
        using wellformed_transaction_unlabel_cases(1)[OF T_valid]
        by (force simp add: unlabel_def)

      have **: "(σ ∘s α) y · 𝓘 ∈ subterms (t · σ ∘s α ∘s 𝓘)"
               "timpl_closure_set (set FP) (set TI) ⊢c t · σ ∘s α · 𝓘 ·α α0 (dblsst 𝒜 𝓘)"
        using fv_subterms_substI[OF t(2), of "σ ∘s α ∘s 𝓘"] subst_compose[of "σ ∘s α" 𝓘 y]
              subterms_subst_subset[of "σ ∘s α ∘s 𝓘" t] receives_covered t(1)
        unfolding intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI, symmetric]
        by auto

      have "Abs (α0 (dblsst 𝒜 𝓘) yn) ∈ ⋃(funs_term ‘ (timpl_closure_set (set FP) (set TI)))"
        using y_abs abs_subterms_in[OF **(1), of "α0 (dblsst 𝒜 𝓘)"]
              ideduct_synth_priv_fun_in_ik[
                OF **(2) funs_term_Fun_subterm'[of "Abs (α0 (dblsst 𝒜 𝓘) yn)" "[]"]]
        by force
      hence "(σ ∘s α) y · 𝓘 ·α α0 (dblsst 𝒜 𝓘) ∈ subtermsset (timpl_closure_set (set FP) (set TI))"
        using y_abs wf_trms_subterms[OF timpl_closure_set_wf_trms[OF FP(2), of "set TI"]]
              funs_term_Fun_subterm[of "Abs (α0 (dblsst 𝒜 𝓘) yn)"]
        unfolding wftrm_def by fastforce
      hence "funs_term ((σ ∘s α) y · 𝓘 ·α α0 (dblsst 𝒜 𝓘))
              ⊆ (⋃t ∈ timpl_closure_set (set FP) (set TI). funs_term t)"
        using funs_term_subterms_eq(2)[of "timpl_closure_set (set FP) (set TI)"] by blast
      thus ?thesis using y_abs OCC(1) by fastforce
    next
      assume "y ∈ fvlsst (transaction_selects T)"
      then obtain l s where "(l,select⟨Var y, Fun (Set s) []⟩) ∈ set (transaction_selects T)"
        using * by (auto simp add: unlabel_def)
      then obtain U where U:
         "prefix (U@[(l,select⟨Var y, Fun (Set s) []⟩)]) (transaction_selects T)"
        using in_set_conv_decomp[of "(l, select⟨Var y,Fun (Set s) []⟩)" "transaction_selects T"]
        by (auto simp add: prefix_def)
      hence "select⟨Var y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))"
        by (force simp add: prefix_def unlabel_def)
      hence "select⟨(σ ∘s α) y, Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T ·lsst σ ∘s α))"
        using subst_lsst_unlabel_member
        by fastforce
      hence "(Fun (Val yn) [], Fun (Set s) []) ∈ set (dblsst 𝒜 𝓘)"
        using yn wellformed_transaction_sem_selects[
                OF T_valid 𝓘_is_T_model, of "(σ ∘s α) y" "Fun (Set s) []"]
        by fastforce
      hence "Fun (Val yn) [] ∈ subtermsset (trmslsst 𝒜) ·set 𝓘"
        using dbsst_in_cases[of "Fun (Val yn) []"]
        by (fastforce simp add: dbsst_def)
      thus ?thesis
        using OCC(3) yn abs_in[of "Fun (Val yn) []" _ "α0 (dblsst 𝒜 𝓘)"]
        unfolding abs_value_constants_def
        by (metis (mono_tags, lifting) mem_Collect_eq subsetCE)
    qed
  qed
qed

lemma transaction_prop4:
```

```
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "x ∈ set (transaction_fresh T)"
    and y: "y ∈ fv_transaction T - set (transaction_fresh T)" "Γ_v y = TAtom Value"
  shows "(σ ∘_s α) x · I ∉ subterms_set (trms_lsst (A ·_lsst I))" (is ?A)
    and "(σ ∘_s α) y · I ∈ subterms_set (trms_lsst (A ·_lsst I))" (is ?B)
proof -
  let ?T' = "dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"

  from I have I': "welltyped_constraint_model I A"
    using welltyped_constraint_model_prefix by auto

  have T_P_addm: "admissible_transaction T'" when T': "T' ∈ set P " for T'
    by (meson T' P)

  have T_adm: "admissible_transaction T"
    by (metis (full_types) P T)

  from T_adm have T_valid: "wellformed_transaction T"
    unfolding admissible_transaction_def by blast

  have be: "bvars_lsst A = {}"
    using T_P_addm A_reach reachable_constraints_no_bvars transaction_no_bvars(2) by blast

  have T_no_bvars: "fv_transaction T = vars_transaction T"
    using transaction_no_bvars[OF T_adm] by simp

  have I_wt: "wt_subst I" by (metis I welltyped_constraint_model_def)

  obtain xn where xn: "σ x = Fun (Val xn) []"
    using σ x unfolding transaction_fresh_subst_def by force

  then have xnxn: "(σ ∘_s α) x = Fun (Val xn) []"
    unfolding subst_compose_def by auto

  from xn xnxn have a0: "(σ ∘_s α) x · I = Fun (Val xn) []"
    by auto

  have b0: "Γ_v x = TAtom Value"
    using P x T protocol_transaction_vars_TAtom_typed(3)
    by metis

  note 0 = a0 b0

  have xT: "x ∈ fv_transaction T"
    using x transaction_fresh_vars_subset[OF T_valid]
    by fast

  have σ_x_nin_A: "σ x ∉ subterms_set (trms_lsst A)"
  proof -
    have "σ x ∈ subst_range σ"
      by (metis σ transaction_fresh_subst_sends_to_val x)
    moreover
    have "(∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst A))"
      using σ transaction_fresh_subst_def[of σ T A] by auto
    ultimately
    show ?thesis
      by auto
  qed
```

```
have *: "y ∉ set (transaction_fresh T)"
   using assms by auto

have **: "y ∈ fv_lsst (transaction_receive T) ∨ y ∈ fv_lsst (transaction_selects T)"
  using * y wellformed_transaction_fv_in_receives_or_selects[OF T_valid]
  by blast

have y_fv: "y ∈ fv_transaction T" using y fv_transaction_unfold by blast

have y_val: "fst y = TAtom Value" using y(2) Γ_v_TAtom''(2) by blast

have "list_all (λx. fst x = Var Value) (transaction_fresh T)"
  using x T_adm unfolding admissible_transaction_def by fast
hence x_val: "fst x = TAtom Value" using x unfolding list_all_iff by blast

have "σ x · I ∉ subterms_set (trms_lsst (A ·_lsst I))"
proof (rule ccontr)
  assume "¬σ x · I ∉ subterms_set (trms_lsst (A ·_lsst I))"
  then have a: "σ x · I ∈ subterms_set (trms_lsst (A ·_lsst I))"
    by auto

  then have σ_x_I_in_A: "σ x · I ∈ subterms_set (trms_lsst A) ·_set I"
    using reachable_constraints_subterms_subst[OF A_reach I' P] by blast

  have "∃u. u ∈ fv_lsst A ∧ I u = σ x"
  proof -
    from σ_x_I_in_A have "∃tu. tu ∈ ⋃ (subterms ` (trms_lsst A)) ∧ tu · I = σ x · I"
      by force
    then obtain tu where tu: "tu ∈ ⋃ (subterms ` (trms_lsst A)) ∧ tu · I = σ x · I"
      by auto
    then have "tu ≠ σ x"
      using σ_x_nin_A by auto
    moreover
    have "tu · I = σ x"
      using tu by (simp add: xn)
    ultimately
    have "∃u. tu = Var u"
      unfolding xn by (cases tu) auto
    then obtain u where "tu = Var u"
      by auto
    have "u ∈ fv_lsst A ∧ I u = σ x"
    proof -
      have "u ∈ vars_lsst A"
        using ⟨tu = Var u⟩ tu var_subterm_trms_sst_is_vars_sst by fastforce
      then have "u ∈ fv_lsst A"
        using be vars_sst_is_fv_sst_bvars_sst[of "unlabel A"] by blast
      moreover
      have "I u = σ x"
        using ⟨tu = Var u⟩ ⟨tu · I = σ x⟩ by auto
      ultimately
      show ?thesis
        by auto
    qed
    then show "∃u. u ∈ fv_lsst A ∧ I u = σ x"
      by metis
  qed
  then obtain u where u:
    "u ∈ fv_lsst A" "I u = σ x"
    by auto
  then have u_TA: "Γ_v u = TAtom Value"
    using P(1) T x_val Γ_v_TAtom''(2)[of x]
          wt_subst_trm''[OF I_wt, of "Var u"] wt_subst_trm''[of σ "Var x"]
```

```
        transaction_fresh_subst_wt[OF σ] protocol_transaction_vars_TAtom_typed(3)
    by force
  have "∃B. prefix B 𝒜 ∧ u ∉ fv_lsst B ∧ ℐ u ∈ subterms_set (trms_lsst B)"
    using u u_TA
    by (metis welltyped_constraint_model_prefix[OF ℐ]
          constraint_model_Value_var_in_constr_prefix[OF 𝒜_reach _ P])
  then obtain B where "prefix B 𝒜 ∧ u ∉ fv_lsst B ∧ ℐ u ∈ subterms_set (trms_lsst B)"
    by blast
  moreover have "⋃(subterms ` trms_lsst xs) ⊆ ⋃(subterms ` trms_lsst ys)"
    when "prefix xs ys"
    for xs ys::"('fun,'atom,'sets,'lbl) prot_strand"
    using that subterms_set_mono trms_sst_mono unlabel_mono set_mono_prefix by metis
  ultimately have "ℐ u ∈ subterms_set (trms_lsst 𝒜)"
    by blast
  then have "σ x ∈ subterms_set (trms_lsst 𝒜)"
    using u by auto
  then show "False"
    using σ_x_nin_A by auto
qed
then show ?A
  unfolding subst_compose_def xn by auto

from ** show ?B
proof
  define T' where "T' ≡ transaction_receive T"
  define ϑ where "ϑ ≡ σ ∘ₛ α"

  assume y: "y ∈ fv_lsst (transaction_receive T)"
  hence "Var y ∈ subterms_set (trms_lsst T')" by (metis T'_def fv_sst_is_subterm_trms_sst)
  then obtain z where z: "z ∈ set (unlabel T')" "Var y ∈ subterms_set (trms_sstp z)"
    by (induct T') auto

  have "is_Receive z"
    using T_adm Ball_set[of "unlabel T'" is_Receive] z(1)
    unfolding admissible_transaction_def wellformed_transaction_def T'_def
    by blast
  then obtain ty where "z = receive⟨ty⟩" by (cases z) auto
  hence ty: "receive⟨ty · ϑ⟩ ∈ set (unlabel (T' ·lsst ϑ))" "ϑ y ∈ subterms (ty · ϑ)"
    using z subst_mono unfolding subst_apply_labeled_stateful_strand_def unlabel_def by force+
  hence y_deduct: "ik_lsst 𝒜 ·set ℐ ⊢ ty · ϑ · ℐ"
    using transaction_receive_deduct[OF T_adm _ σ α]
    by (metis ℐ T'_def ϑ_def welltyped_constraint_model_def)

  obtain zn where zn: "(σ ∘ₛ α) y · ℐ = Fun (Val (zn, False)) []"
    using transaction_var_becomes_Val[
            OF reachable_constraints.step[OF 𝒜_reach T σ α] ℐ σ α P T, of y]
          transaction_fresh_subst_transaction_renaming_subst_range(2)[OF σ α *]
          y_fv y_val
    by (metis subst_apply_term.simps(1))

  have "(σ ∘ₛ α) y · ℐ ∈ subterms_set (ik_lsst 𝒜 ·set ℐ)"
    using private_fun_deduct_in_ik[OF y_deduct, of "Val (zn, False)"]
    by (metis ϑ_def ty(2) zn subst_mono public.simps(3) snd_eqD)
  thus ?B
    using ik_sst_subst[of "unlabel 𝒜" ℐ] unlabel_subst[of 𝒜 ℐ]
          subterms_set_mono[OF ik_sst_trms_sst_subset[of "unlabel (𝒜 ·lsst ℐ)"]]
    by fastforce
next
  assume y': "y ∈ fv_lsst (transaction_selects T)"
  then obtain s where s: "select⟨Var y,s⟩ ∈ set (unlabel (transaction_selects T))"
                         "fst y = TAtom Value"
    using admissible_transaction_strand_step_cases(1,2)[OF T_adm] by fastforce
```

```
obtain z zn where zn: "(σ ∘s α) y = Var z" "I z = Fun (Val zn) []"
  using transaction_var_becomes_Val[
          OF reachable_constraints.step[OF A_reach T σ α] I σ α P T]
        transaction_fresh_subst_transaction_renaming_subst_range(2)[OF σ α *]
        y_fv T_no_bvars(1) s(2)
  by (metis subst_apply_term.simps(1))

have transaction_selects_db_here:
    "⋀n s. select⟨Var (TAtom Value, n), Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))
          ⟹ (α (TAtom Value, n) · I, Fun (Set s) []) ∈ set (db_lsst A I)"
  using transaction_selects_db[OF T_adm _ σ α] I
  unfolding welltyped_constraint_model_def by auto

have "∃n. y = (Var Value, n)"
  using T Γv_TAtom_inv(2) y_fv y(2)
  by blast
moreover
have "admissible_transaction_selects T"
  using T_adm admissible_transaction_def
  by blast
then have "is_Fun_Set (the_set_term (select⟨Var y,s⟩))"
  using s unfolding admissible_transaction_selects_def
  by auto
then have "∃ss. s = Fun (Set ss) []"
  using is_Fun_Set_exi
  by auto
ultimately
obtain n ss where nss: "y = (TAtom Value, n)" "s = Fun (Set ss) []"
  by auto
then have "select⟨Var (TAtom Value, n), Fun (Set ss) []⟩ ∈ set (unlabel (transaction_selects T))"
  using s by auto
then have in_db: "(α (TAtom Value, n) · I, Fun (Set ss) []) ∈ set (db_lsst A I)"
  using transaction_selects_db_here[of n ss] by auto
have "(I z, s) ∈ set (db_lsst A I)"
proof -
  have "(α y · I, s) ∈ set (db_lsst A I)"
    using in_db nss by auto
  moreover
  have "α y = Var z"
    using zn
    by (metis (no_types, hide_lams) σ subst_compose_def subst_imgI subst_to_var_is_var
              term.distinct(1) transaction_fresh_subst_def var_comp(2))
  then have "α y · I = I z"
    by auto
  ultimately
  show "(I z, s) ∈ set (db_lsst A I)"
    by auto
qed
then have "∃t' s'. insert⟨t',s'⟩ ∈ set (unlabel A) ∧ I z = t' · I ∧ s = s' · I"
  using db_sst_in_cases[of "I z" s "unlabel A" I "[]"] unfolding db_sst_def by auto
then obtain t' s' where t's': "insert⟨t',s'⟩ ∈ set (unlabel A) ∧ I z = t' · I ∧ s = s' · I"
  by auto
then have "t' ∈ subterms_set (trms_lsst A)"
  by force
then have "t' · I ∈ (subterms_set (trms_lsst A)) ·set I"
  by auto
then have "I z ∈ (subterms_set (trms_lsst A)) ·set I"
  using t's' by auto
then have "I z ∈ subterms_set (trms_lsst (A ·lsst I))"
  using reachable_constraints_subterms_subst[
          OF A_reach welltyped_constraint_model_prefix[OF I] P]
  by auto
then show ?B
```

```
    using zn(1) by simp
  qed
qed

lemma transaction_prop5:
  fixes T σ α 𝒜 ℐ T' a0 a0' ϑ
  defines "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"
    and "a0 ≡ α₀ (db_lsst 𝒜 ℐ)"
    and "a0' ≡ α₀ (db_lsst (𝒜@T') ℐ)"
    and "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@T')"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and step: "list_all (transaction_check FP OCC TI) P"
  shows "∃δ ∈ abs_substs_fun ‘ set (transaction_check_comp FP OCC TI T).
        ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
           (σ ∘_s α) x · ℐ ·_α a0 = absc (δ x) ∧
           (σ ∘_s α) x · ℐ ·_α a0' = absc (absdbupd (unlabel (transaction_updates T)) x (δ x))"
proof -
  define comp0 where "comp0 ≡ abs_substs_fun ‘ set (transaction_check_comp FP OCC TI T)"
  define check0 where "check0 ≡ transaction_check FP OCC TI T"
  define upd where "upd ≡ λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)"
  define b0 where "b0 ≡ λx. THE b. absc b = (σ ∘_s α) x · ℐ ·_α a0"

  note all_defs = comp0_def check0_def a0_def a0'_def upd_def b0_def ϑ_def T'_def

  have ϑ_wt: "wt_subst (ϑ δ)" for δ
    unfolding ϑ_def wt_subst_def
    by fastforce

  have 𝒜_wf_trms: "wf_trms (trms_lsst 𝒜)"
    by (metis reachable_constraints_wf_trms admissible_transactions_wf_trms P(1) 𝒜_reach)

  have ℐ_interp: "interpretation_subst ℐ"
    and ℐ_wt: "wt_subst ℐ"
    and ℐ_wf_trms: "wf_trms (subst_range ℐ)"
    by (metis ℐ welltyped_constraint_model_def constraint_model_def,
        metis ℐ welltyped_constraint_model_def,
        metis ℐ welltyped_constraint_model_def constraint_model_def)

  have ℐ_is_T_model: "strand_sem_stateful (ik_lsst 𝒜 ·_set ℐ) (set (db_lsst 𝒜 ℐ)) (unlabel T') ℐ"
    using ℐ unlabel_append[of 𝒜 T'] db_sst_set_is_dbupd_sst[of "unlabel 𝒜" ℐ "[]"]
          strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" "unlabel T'" ℐ]
    by (simp add: welltyped_constraint_model_def constraint_model_def db_sst_def)

  have T_adm: "admissible_transaction T"
    using T P(1) Ball_set[of P "admissible_transaction"]
    by blast
  hence T_valid: "wellformed_transaction T"
```

```
  unfolding admissible_transaction_def by blast

have T_no_bvars: "fv_transaction T = vars_transaction T" "bvars_transaction T = {}"
  using transaction_no_bvars[OF T_adm] by simp_all

have T_vars_const_typed: "∀x ∈ fv_transaction T. Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
  and T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  using T P protocol_transaction_vars_TAtom_typed(2,3)[of T] by simp_all

have wt_σαℐ: "wtsubst (σ ∘s α ∘s ℐ)" and wt_σα: "wtsubst (σ ∘s α)"
  using ℐ_wt wt_subst_compose transaction_fresh_subst_wt[OF σ T_fresh_vars_value_typed]
        transaction_renaming_subst_wt[OF α]
  by blast+

have T_vars_vals: "∀x ∈ fv_transaction T. ∃n. (σ ∘s α) x · ℐ = Fun (Val (n, False)) []"
proof
  fix x assume x: "x ∈ fv_transaction T"
  show "∃n. (σ ∘s α) x · ℐ = Fun (Val (n, False)) []"
  proof (cases "x ∈ subst_domain σ")
    case True
    then obtain n where "σ x = Fun (Val (n, False)) []"
      using σ unfolding transaction_fresh_subst_def
      by moura
    thus ?thesis by (simp add: subst_compose_def)
  next
    case False
    hence *: "(σ ∘s α) x = α x" by (auto simp add: subst_compose_def)

    obtain y where y: "Γv x = Γv y" "α x = Var y"
      using transaction_renaming_subst_wt[OF α]
            transaction_renaming_subst_is_renaming[OF α]
      by (metis Γ.simps(1) prod.exhaust wtsubst_def)
    hence "y ∈ fvlsst (transaction_strand T ·lsst σ ∘s α)"
      using x * T_no_bvars(2) unlabel_subst[of "transaction_strand T" "σ ∘s α"]
            fvsst_subst_fv_subset[of x "unlabel (transaction_strand T)" "σ ∘s α"]
      by auto
    hence "y ∈ fvlsst (𝒜@duallsst (transaction_strand T ·lsst σ ∘s α))"
      using fvsst_unlabel_duallsst_eq[of "transaction_strand T ·lsst σ ∘s α"]
            fvsst_append[of "unlabel 𝒜"] unlabel_append[of 𝒜]
      by auto
    thus ?thesis
      using x y * T P
            constraint_model_Value_term_is_Val[
              OF reachable_constraints.step[OF 𝒜_reach T σ α] ℐ[unfolded T'_def] P(1), of y]
            admissible_transaction_Value_vars[of T]
      by simp
  qed
qed

have T_vars_absc: "∀x ∈ fv_transaction T. ∃!n. (σ ∘s α) x · ℐ ·α a0 = absc n"
  using T_vars_vals by fastforce
hence "(absc ∘ b0) x = (σ ∘s α) x · ℐ ·α a0" when "x ∈ fv_transaction T" for x
  using that unfolding b0_def by fastforce
hence T_vars_absc': "t · (absc ∘ b0) = t · (σ ∘s α) · ℐ ·α a0"
  when "fv t ⊆ fv_transaction T" "∄n T. Fun (Val n) T ∈ subterms t" for t
  using that(1) abs_term_subst_eq'[OF _ that(2), of "σ ∘s α ∘s ℐ" a0 "absc ∘ b0"]
        subst_compose[of "σ ∘s α" ℐ] subst_subst_compose[of t "σ ∘s α" ℐ]
  by fastforce

have "∃δ ∈ comp0. ∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ b0 x = δ x"
proof -
  let ?S = "set (unlabel (transaction_selects T))"
  let ?C = "set (unlabel (transaction_checks T))"
```

```
let ?xs = "fv_transaction T - set (transaction_fresh T)"

note * = transaction_prop3[OF 𝒜_reach T ℐ[unfolded T'_def] σ α FP OCC TI P(1)]

have **:
    "∀x ∈ set (transaction_fresh T). b0 x = {}"
    "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ b0)"
      (is ?B)
proof -
  show ?B
  proof (intro ballI impI)
    fix t assume t: "t ∈ trms_lsst (transaction_receive T)"
    hence t': "fv t ⊆ fv_transaction T" "∄n T. Fun (Val n) T ∈ subterms t"
      using trms_transaction_unfold[of T] vars_transaction_unfold[of T]
            trms_sst_fv_vars_sst_subset[of t "unlabel (transaction_strand T)"]
            transactions_have_no_Value_consts'[OF T_adm]
            wellformed_transaction_send_receive_fv_subset(1)[OF T_valid t(1)]
      by blast+

    have "intruder_synth_mod_timpls FP TI (t · (absc ∘ b0))"
      using t(1) t' *(2) T_vars_absc'
      by (metis a0_def)
    moreover have "(absc ∘ b0) x = (ϑ b0) x" when "x ∈ fv t" for x
      using that T P admissible_transaction_Value_vars[of T]
            ⟨fv t ⊆ fv_transaction T⟩ Γ_v_TAtom''(2)[of x]
      unfolding ϑ_def by fastforce
    hence "t · (absc ∘ b0) = t · ϑ b0"
      using term_subst_eq[of t "absc ∘ b0" "ϑ b0"] by argo
    ultimately show "intruder_synth_mod_timpls FP TI (t · ϑ b0)"
      using intruder_synth.simps[of "set FP"] by (cases "t · ϑ b0") metis+
  qed
qed (simp add: *(1) a0_def b0_def)

have ***: "∀x ∈ ?xs. ∀s. select⟨Var x,Fun (Set s) []⟩ ∈ ?S ⟶ s ∈ b0 x"
          "∀x ∈ ?xs. ∀s. ⟨Var x in Fun (Set s) []⟩ ∈ ?C ⟶ s ∈ b0 x"
          "∀x ∈ ?xs. ∀s. ⟨Var x not in Fun (Set s) []⟩ ∈ ?C ⟶ s ∉ b0 x"
          "∀x ∈ ?xs. fst x = TAtom Value ⟶ b0 x ∈ set OCC"
  unfolding a0_def b0_def
  using *(3,4) apply (force, force)
  using *(5) apply force
  using *(6) admissible_transaction_Value_vars[OF bspec[OF P T]] by force

show ?thesis
  using transaction_check_comp_in[OF T_adm **[unfolded ϑ_def] ***]
  unfolding comp0_def
  by metis
qed
hence 1: "∃δ ∈ comp0. ∀x ∈ fv_transaction T.
           fst x = TAtom Value ⟶ (σ ∘_s α) x · ℐ ·_α a0 = absc (δ x)"
  using T_vars_absc unfolding b0_def a0_def by fastforce

obtain δ where δ:
    "δ ∈ comp0" "∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ (σ ∘_s α) x · ℐ ·_α a0 = absc (δ x)"
  using 1 by moura

have 2: "ϑ x · ℐ ·_α α0 (db'_lsst (dual_lsst (A ·_lsst ϑ)) ℐ D) = absc (absdbupd (unlabel A) x d)"
  when "ϑ x · ℐ ·_α α0 D = absc d"
  and "∀t u. insert⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃y s. t = Var y ∧ u = Fun (Set s) [])"
  and "∀t u. delete⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃y s. t = Var y ∧ u = Fun (Set s) [])"
  and "∀y ∈ fv_lsst A. ϑ x · ℐ = ϑ y · ℐ ⟶ x = y"
  and "∀y ∈ fv_lsst A. ∃n. ϑ y · ℐ = Fun (Val n) []"
  and x: "ϑ x · ℐ = Fun (Val n) []"
  and D: "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
```

```
  for A::"('fun,'atom,'sets,'nat) prot_strand" and x ϑ D n d
  using that(2,3,4,5)
proof (induction A rule: List.rev_induct)
  case (snoc a A)
  then obtain l b where a: "a = (l,b)" by (metis surj_pair)

  have IH: "α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) I D) n = absdbupd (unlabel A) x d"
    using snoc unlabel_append[of A "[a]"] a x
    by (simp del: unlabel_append)

  have b_prems: "∀y ∈ fv_sstp b. ϑ x · I = ϑ y · I ⟶ x = y"
                "∀y ∈ fv_sstp b. ∃n. ϑ y · I = Fun (Val n) []"
    using snoc.prems(3,4) a by (simp_all add: unlabel_def)

  have *: "filter is_Update (unlabel (dual_lsst (A@[a] ·_lsst ϑ))) =
           filter is_Update (unlabel (dual_lsst (A ·_lsst ϑ)))"
          "filter is_Update (unlabel (A@[a])) = filter is_Update (unlabel A)"
    when "¬is_Update b"
    using that a
    by (cases b, simp_all add: dual_lsst_def unlabel_def subst_apply_labeled_stateful_strand_def)+

  note ** = IH a dual_lsst_subst_append[of A "[a]" ϑ]

  note *** = * absdbupd_filter[of "unlabel (A@[a])"]
             absdbupd_filter[of "unlabel A"]
             db_sst_filter[of "unlabel (dual_lsst (A@[a] ·_lsst ϑ))"]
             db_sst_filter[of "unlabel (dual_lsst (A ·_lsst ϑ))"]

  note **** = **(2,3) dual_lsst_subst_snoc[of A a ϑ]
              unlabel_append[of "dual_lsst A ·_lsst ϑ" "[dual_lsstp a ·_lsstp ϑ]"]
              db_sst_append[of "unlabel (dual_lsst A ·_lsst ϑ)" "unlabel [dual_lsstp a ·_lsstp ϑ]" I D]

  have "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) I D) n = absdbupd (unlabel (A@[a])) x d" using ** ***
  proof (cases b)
    case (Insert t t')
    then obtain y s m where y: "t = Var y" "t' = Fun (Set s) []" "ϑ y · I = Fun (Val m) []"
      using snoc.prems(1) b_prems(2) a by (fastforce simp add: unlabel_def)
    hence a': "db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) I D =
               List.insert ((Fun (Val m) [], Fun (Set s) [])) (db'_lsst (dual_lsst A ·_lsst ϑ) I D)"
              "unlabel [dual_lsstp a ·_lsstp ϑ] = [insert⟨ϑ y, Fun (Set s) []⟩]"
              "unlabel [a] = [insert⟨Var y, Fun (Set s) []⟩]"
      using **** Insert by simp_all

    show ?thesis
    proof (cases "x = y")
      case True
      hence "ϑ x · I = ϑ y · I" by simp
      hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) I D) n =
             insert s (α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) I D) n)"
        by (metis (no_types, lifting) y(3) a'(1) x dual_lsst_subst to_abs_list_insert')
      thus ?thesis using True IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
    next
      case False
      hence "ϑ x · I ≠ ϑ y · I" using b_prems(1) y Insert by simp
      hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) I D) n = α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) I D) n"
        by (metis (no_types, lifting) y(3) a'(1) x dual_lsst_subst to_abs_list_insert)
      thus ?thesis using False IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
    qed
  next
    case (Delete t t')
    then obtain y s m where y: "t = Var y" "t' = Fun (Set s) []" "ϑ y · I = Fun (Val m) []"
      using snoc.prems(2) b_prems(2) a by (fastforce simp add: unlabel_def)
    hence a': "db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) I D =
```

```
            List.removeAll ((Fun (Val m) [], Fun (Set s) [])) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)"
           "unlabel [dual_lsstp a ·_lsstp ϑ] = [delete⟨ϑ y, Fun (Set s) []⟩]"
           "unlabel [a] = [delete⟨Var y, Fun (Set s) []⟩]"
    using **** Delete by simp_all

  have "∃s S. snd d = Fun (Set s) []" when "d ∈ set (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)" for d
    using snoc.prems(1,2) db_lsst_dual_lsst_set_ex[OF that _ _ D] by (simp add: unlabel_def)
  moreover {
    fix t::"('fun,'atom,'sets) prot_term"
      and D::"(('fun,'atom,'sets) prot_term × ('fun,'atom,'sets) prot_term) list"
    assume "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
    hence "removeAll (t, Fun (Set s) []) D = filter (λd. ♯S. d = (t, Fun (Set s) S)) D"
      by (induct D) auto
  } ultimately have a'':
      "List.removeAll ((Fun (Val m) [], Fun (Set s) [])) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D) =
       filter (λd. ♯S. d = (Fun (Val m) [], Fun (Set s) S)) (db'_lsst (dual_lsst A ·_lsst ϑ) ℐ D)"
    by simp

  show ?thesis
  proof (cases "x = y")
    case True
    hence "ϑ x · ℐ = ϑ y · ℐ" by simp
    hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D) n =
           (α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) ℐ D) n) - {s}"
      using y(3) a'' a'(1) x by (simp add: dual_lsst_subst to_abs_list_remove_all')
    thus ?thesis using True IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
  next
    case False
    hence "ϑ x · ℐ ≠ ϑ y · ℐ" using b_prems(1) y Delete by simp
    hence "α₀ (db'_lsst (dual_lsst (A@[a] ·_lsst ϑ)) ℐ D) n = α₀ (db'_lsst (dual_lsst (A ·_lsst ϑ)) ℐ D) n"
      by (metis (no_types, lifting) y(3) a'(1) x dual_lsst_subst to_abs_list_remove_all)
    thus ?thesis using False IH a'(3) absdbupd_append[of "unlabel A"] by (simp add: unlabel_def)
  qed
 qed simp_all
 thus ?case by (simp add: x)
qed (simp add: that(1))

have 3: "x = y"
 when xy: "(σ ∘ₛ α) x · ℐ = (σ ∘ₛ α) y · ℐ" "x ∈ fv_transaction T" "y ∈ fv_transaction T"
 for x y
proof -
 have "x ∉ set (transaction_fresh T) ⟹ y ∉ set (transaction_fresh T) ⟹ ?thesis"
   using xy admissible_transaction_strand_sem_fv_ineq[OF T_adm ℐ_is_T_model[unfolded T'_def]]
   by fast
 moreover {
   assume *: "x ∈ set (transaction_fresh T)" "y ∈ set (transaction_fresh T)"
   then obtain xn yn where "σ x = Fun (Val xn) []" "σ y = Fun (Val yn) []"
     by (metis transaction_fresh_subst_sends_to_val[OF σ])
   hence "σ x = σ y" using that(1) by (simp add: subst_compose)
   moreover have "inj_on σ (subst_domain σ)" "x ∈ subst_domain σ" "y ∈ subst_domain σ"
     using * σ unfolding transaction_fresh_subst_def by auto
   ultimately have ?thesis unfolding inj_on_def by blast
 } moreover have False when "x ∈ set (transaction_fresh T)" "y ∉ set (transaction_fresh T)"
   using that(2) xy T_no_bvars admissible_transaction_Value_vars[OF bspec[OF P T], of y]
         transaction_prop4[OF 𝒜_reach T ℐ[unfolded T'_def] σ α P that(1), of y]
   by auto
 moreover have False when "x ∉ set (transaction_fresh T)" "y ∈ set (transaction_fresh T)"
   using that(1) xy T_no_bvars admissible_transaction_Value_vars[OF bspec[OF P T], of x]
         transaction_prop4[OF 𝒜_reach T ℐ[unfolded T'_def] σ α P that(2), of x]
   by fastforce
 ultimately show ?thesis by metis
qed
```

```
  have 4: "∃y s. t = Var y ∧ u = Fun (Set s) []"
    when "insert⟨t,u⟩ ∈ set (unlabel (transaction_strand T))" for t u
    using that admissible_transaction_strand_step_cases(4)[OF T_adm] T_valid
    by blast

  have 5: "∃y s. t = Var y ∧ u = Fun (Set s) []"
    when "delete⟨t,u⟩ ∈ set (unlabel (transaction_strand T))" for t u
    using that admissible_transaction_strand_step_cases(4)[OF T_adm] T_valid
    by blast

  have 6: "∃n. (σ ∘s α) y · I = Fun (Val (n, False)) []" when "y ∈ fv_transaction T" for y
    using that by (simp add: T_vars_vals)

  have "list_all wellformed_transaction P" "list_all admissible_transaction_updates P"
    using P(1) Ball_set[of P "admissible_transaction"] Ball_set[of P wellformed_transaction]
          Ball_set[of P admissible_transaction_updates]
    unfolding admissible_transaction_def by fastforce+
  hence 7: "∃s. snd d = Fun (Set s) []" when "d ∈ set (db_lsst A I)" for d
    using that reachable_constraints_db_lsst_set_args_empty[OF A_reach]
    unfolding admissible_transaction_updates_def by (cases d) simp

  have "(σ ∘s α) x · I ·α a0' = absc (upd δ x)"
    when x: "x ∈ fv_transaction T" "fst x = TAtom Value" for x
  proof -
    have "(σ ∘s α) x · I ·α α0 (db'_lsst (dual_lsst (transaction_strand T ·lsst σ ∘s α)) I (db_lsst A I))
          = absc (absdbupd (unlabel (transaction_strand T)) x (δ x))"
      using 2[of "σ ∘s α" x "db_lsst A I" "δ x" "transaction_strand T"]
            3[OF _ x(1)] 4 5 6[OF that(1)] 6 7 x δ(2)
      unfolding all_defs by blast
    thus ?thesis
      using x db_sst_append[of "unlabel A"] absdbupd_wellformed_transaction[OF T_valid]
      unfolding all_defs db_sst_def by force
  qed
  thus ?thesis using δ Γ_v_TAtom''(2) unfolding all_defs by blast
qed

lemma transaction_prop6:
  fixes T σ α A I T' a0 a0'
  defines "T' ≡ dual_lsst (transaction_strand T ·lsst σ ∘s α)"
    and "a0 ≡ α0 (db_lsst A I)"
    and "a0' ≡ α0 (db_lsst (A@T') I)"
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@T')"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik A I. timpl_closure_set (set FP) (set TI) ⊢c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
      "α_vals A I ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and step: "list_all (transaction_check FP OCC TI) P"
  shows "∀t ∈ timpl_closure_set (α_ik A I) (α_ti A T σ α I).
         timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
    and "timpl_closure_set (α_vals A I) (α_ti A T σ α I) ⊆ absc ‘ set OCC" (is ?B)
    and "∀t ∈ trms_lsst (transaction_send T). is_Fun (t · (σ ∘s α) · I ·α a0') ⟶
```

```
        timpl_closure_set (set FP) (set TI) ⊢_c t · (σ ∘_s α) · ℐ ·_α a0'" (is ?C)
    and "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
        (σ ∘_s α) x · ℐ ·_α a0' ∈ absc ` set OCC" (is ?D)
proof -
  define comp0 where "comp0 ≡ abs_substs_fun ` set (transaction_check_comp FP OCC TI T)"
  define check0 where "check0 ≡ transaction_check FP OCC TI T"

  define upd where "upd ≡ λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)"

  define ϑ where "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"

  have T_adm: "admissible_transaction T" using T P(1) by metis
  hence T_valid: "wellformed_transaction T" by (metis admissible_transaction_def)

  have ϑ_prop: "ϑ σ x = absc (σ x)" when "Γ_v x = TAtom Value" for σ x
    using that Γ_v_TAtom''(2)[of x] unfolding ϑ_def by simp


  have 0: "∃δ ∈ comp0. ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
          (σ ∘_s α) x · ℐ ·_α a0 = absc (δ x) ∧
          (σ ∘_s α) x · ℐ ·_α a0' = absc (upd δ x)"
    using transaction_prop5[OF 𝒜_reach T ℐ[unfolded T'_def] σ α FP OCC TI P step]
    unfolding a0_def a0'_def T'_def upd_def comp0_def
    by blast


  have 1: "(δ x, upd δ x) ∈ (set TI)⁺"
    when "δ ∈ comp0" "δ x ≠ upd δ x" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
    for x δ
    using T that step Ball_set[of P "transaction_check FP OCC TI"]
          transaction_prop1[of δ FP OCC TI T x] TI
    unfolding upd_def comp0_def
    by blast


  have 2:  "upd δ x ∈ set OCC"
    when "δ ∈ comp0" "x ∈ fv_transaction T" "fst x = TAtom Value" for x δ
    using T that step Ball_set[of P "transaction_check FP OCC TI"]
          T_adm FP OCC TI transaction_prop2[of δ FP OCC TI T x]
    unfolding upd_def comp0_def
    by blast+

  obtain δ where δ:
      "δ ∈ comp0"
      "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
        (σ ∘_s α) x · ℐ ·_α a0 = absc (δ x) ∧
        (σ ∘_s α) x · ℐ ·_α a0' = absc (upd δ x)"
    using 0 by moura

  have "∃x. ab = (δ x, upd δ x) ∧ x ∈ fv_transaction T - set (transaction_fresh T) ∧ δ x ≠ upd δ x"
    when ab: "ab ∈ α_ti 𝒜 T σ α ℐ" for ab
  proof -
    obtain a b where ab': "ab = (a,b)" by (metis surj_pair)
    then obtain x where x:
        "a ≠ b" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
        "absc a = (σ ∘_s α) x · ℐ ·_α a0" "absc b = (σ ∘_s α) x · ℐ ·_α a0'"
      using ab unfolding abs_term_implications_def a0_def a0'_def T'_def by blast
    hence "absc a = absc (δ x)" "absc b = absc (upd δ x)"
      using δ(2) admissible_transaction_Value_vars[OF bspec[OF P T] x(2)]
      by metis+
    thus ?thesis using x ab' by blast
  qed
  hence α_ti_TI_subset: "α_ti 𝒜 T σ α ℐ ⊆ {(a,b) ∈ (set TI)⁺. a ≠ b}" using 1[OF δ(1)] by blast
```

```
  have "timpl_closure_set (timpl_closure_set (set FP) (set TI)) (α_ti 𝒜 T σ α ℐ) ⊢_c t"
    when t: "t ∈ timpl_closure_set (α_ik 𝒜 ℐ) (α_ti 𝒜 T σ α ℐ)" for t
    using timpl_closure_set_is_timpl_closure_union[of "α_ik 𝒜 ℐ" "α_ti 𝒜 T σ α ℐ"]
          intruder_synth_timpl_closure_set FP(3) t
    by blast
  thus ?A
    using ideduct_synth_mono[OF _ timpl_closure_set_mono[OF
            subset_refl[of "timpl_closure_set (set FP) (set TI)"]
            α_ti_TI_subset]]
          timpl_closure_set_timpls_trancl_eq'[of "timpl_closure_set (set FP) (set TI)" "set TI"]
    unfolding timpl_closure_set_idem
    by force

  have "timpl_closure_set (α_vals 𝒜 ℐ) (α_ti 𝒜 T σ α ℐ) ⊆
       timpl_closure_set (absc ‘ set OCC) {(a,b) ∈ (set TI)⁺. a ≠ b}"
    using timpl_closure_set_mono[OF _ α_ti_TI_subset] OCC(3) by blast
  thus ?B using OCC(2) timpl_closure_set_timpls_trancl_subset' by blast

  have "transaction_check_post FP TI T δ"
    using T δ(1) step
    unfolding transaction_check_def comp0_def list_all_iff
    by blast
  hence 3: "timpl_closure_set (set FP) (set TI) ⊢_c t · ϑ (upd δ)"
    when "t ∈ trms_lsst (transaction_send T)" "is_Fun (t · ϑ (upd δ))" for t
    using that
    unfolding transaction_check_post_def upd_def ϑ_def
              intruder_synth_mod_timpls_is_synth_timpl_closure_set[OF TI, symmetric]
    by meson

  have 4: "∀x ∈ fv t. (σ ∘_s α ∘_s ℐ) x ·_α a0' = ϑ (upd δ) x"
    when "t ∈ trms_lsst (transaction_send T)" for t
    using wellformed_transaction_send_receive_fv_subset(2)[OF T_valid that]
          δ(2) subst_compose[of "σ ∘_s α" ℐ] ϑ_prop
          admissible_transaction_Value_vars[OF bspec[OF P T]]
    by fastforce

  have 5: "∄n T. Fun (Val n) T ∈ subterms t" when "t ∈ trms_lsst (transaction_send T)" for t
    using that transactions_have_no_Value_consts'[OF T_adm] trms_transaction_unfold[of T]
    by blast

  show ?D using 2[OF δ(1)] δ(2) Γ_v_TAtom''(2) unfolding a0'_def T'_def by blast

  show ?C using 3 abs_term_subst_eq'[OF 4 5] by simp
qed

lemma reachable_constraints_covered_step:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ‘ set OCC"
    and TI:
```

```
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
  shows "∀t ∈ α_ik (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) I.
        timpl_closure_set (set FP) (set TI) ⊢_c t" (is ?A)
    and "α_vals (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) I ⊆ absc ` set OCC" (is ?B)
proof -
  note step_props = transaction_prop6[OF A_reach T I σ α FP(1,2,3) OCC TI P transactions_covered]

  define T' where "T' ≡ dual_lsst (transaction_strand T ·_lsst σ ∘_s α)"
  define a0 where "a0 ≡ α₀ (db_lsst A I)"
  define a0' where "a0' ≡ α₀ (db_lsst (A@T') I)"

  define vals where "vals ≡ λS::('fun,'atom,'sets,'lbl) prot_constr.
      {t ∈ subterms_set (trms_lsst S) ·_set I. ∃n. t = Fun (Val n) []}"

  define vals_sym where "vals_sym ≡ λS::('fun,'atom,'sets,'lbl) prot_constr.
      {t ∈ subterms_set (trms_lsst S). (∃n. t = Fun (Val n) []) ∨ (∃m. t = Var (TAtom Value,m))}"

  have I_wt: "wt_subst I" by (metis I welltyped_constraint_model_def)

  have I_grounds: "fv (t · I) = {}" for t
    using I interpretation_grounds[of I]
    unfolding welltyped_constraint_model_def constraint_model_def by auto

  have T_fresh_vars_value_typed: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    using protocol_transaction_vars_TAtom_typed[OF bspec[OF P(1) T]] by simp_all

  have wt_σαI: "wt_subst (σ ∘_s α ∘_s I)" and wt_σα: "wt_subst (σ ∘_s α)"
    using I_wt wt_subst_compose transaction_fresh_subst_wt[OF σ T_fresh_vars_value_typed]
        transaction_renaming_subst_wt[OF α]
    by blast+

  have "∀T∈set P. bvars_transaction T = {}"
    using P unfolding list_all_iff admissible_transaction_def by metis
  hence A_no_bvars: "bvars_lsst A = {}"
    using reachable_constraints_no_bvars[OF A_reach] by metis

  have I_vals: "∃n. I (TAtom Value, m) = Fun (Val n) []"
    when "(TAtom Value, m) ∈ fv_lsst A" for m
    using constraint_model_Value_term_is_Val'[
        OF A_reach welltyped_constraint_model_prefix[OF I] P(1)]
      A_no_bvars vars_sst_is_fv_sst_bvars_sst[of "unlabel A"] that
    by blast

  have vals_sym_vals: "t · I ∈ vals A" when t: "t ∈ vals_sym A" for t
  proof (cases t)
    case (Var x)
    then obtain m where *: "x = (TAtom Value,m)" using t unfolding vals_sym_def by blast
    moreover have "t ∈ subterms_set (trms_lsst A)" using t unfolding vals_sym_def by blast
    hence "t · I ∈ subterms_set (trms_lsst A) ·_set I" "∃n. I (Var Value, m) = Fun (Val n) []"
      using Var * I_vals[of m] var_subterm_trms_sst_is_vars_sst[of x "unlabel A"]
          Γ_v_TAtom[of Value m] reachable_constraints_Value_vars_are_fv[OF A_reach P(1), of x]
      by blast+
    ultimately show ?thesis using Var unfolding vals_def by auto
  next
    case (Fun f T)
    then obtain n where "f = Val n" "T = []" using t unfolding vals_sym_def by blast
    moreover have "t ∈ subterms_set (trms_lsst A)" using t unfolding vals_sym_def by blast
    hence "t · I ∈ subterms_set (trms_lsst A) ·_set I" using Fun by blast
    ultimately show ?thesis using Fun unfolding vals_def by auto
  qed
```

```
have vals_vals_sym: "∃s. s ∈ vals_sym 𝒜 ∧ t = s ⋅ ℐ" when "t ∈ vals 𝒜" for t
  using that constraint_model_Val_is_Value_term[OF ℐ]
  unfolding vals_def vals_sym_def by fast

have T_adm: "admissible_transaction T" and T_valid: "wellformed_transaction T"
  apply (metis P(1) T)
  using P(1) T Ball_set[of P "admissible_transaction"]
  unfolding admissible_transaction_def by fastforce

have 0:
    "α_ik (𝒜@T') ℐ = (ik_lsst 𝒜 ⋅_set ℐ) ⋅_αset a0' ∪ (ik_lsst T' ⋅_set ℐ) ⋅_αset a0'"
    "α_vals (𝒜@T') ℐ = vals 𝒜 ⋅_αset a0' ∪ vals T' ⋅_αset a0'"
  by (metis abs_intruder_knowledge_append a0'_def,
      metis abs_value_constants_append[of 𝒜 T' ℐ] a0'_def vals_def)

have 1: "(ik_lsst T' ⋅_set ℐ) ⋅_αset a0' =
         (trms_lsst (transaction_send T) ⋅_set (σ ∘_s α) ⋅_set ℐ) ⋅_αset a0'"
  by (metis T'_def dual_transaction_ik_is_transaction_send''[OF T_valid])

have 2: "bvars_lsst (transaction_strand T) ∩ subst_domain σ = {}"
        "bvars_lsst (transaction_strand T) ∩ subst_domain α = {}"
  using T_adm unfolding admissible_transaction_def
  by blast+

have "vals T' ⊆ (σ ∘_s α) ‘ fv_transaction T ⋅_set ℐ"
proof
  fix t assume "t ∈ vals T'"
  then obtain s n where s:
      "s ∈ subterms_set (trms_lsst T')" "t = s ⋅ ℐ" "t = Fun (Val n) []"
    unfolding vals_def by fast
  then obtain u where u:
      "u ∈ subterms_set (trms_lsst (transaction_strand T))"
      "s = u ⋅ (σ ∘_s α)"
    using transaction_fresh_subst_transaction_renaming_subst_trms[OF σ α 2]
          trms_sst_unlabel_dual_lsst_eq[of "transaction_strand T ⋅_lsst σ ∘_s α"]
    unfolding T'_def by blast

  have *: "t = u ⋅ (σ ∘_s α ∘_s ℐ)" by (metis subst_subst_compose s(2) u(2))
  then obtain x where x: "u = Var x"
    using s(3) transactions_have_no_Value_consts(1)[OF T_adm u(1)] by (cases u) force+
  hence **: "x ∈ vars_transaction T"
    by (metis u(1) var_subterm_trms_sst_is_vars_sst)

  have "Γ_v x = TAtom Value"
    using * x s(3) wt_subst_trm''[OF wt_σαℐ, of u]
    by simp
  thus "t ∈ (σ ∘_s α) ‘ fv_transaction T ⋅_set ℐ"
    using transaction_Value_vars_are_fv[OF T_adm **] x *
    by (metis subst_comp_set_image rev_image_eqI subst_apply_term.simps(1))
qed
hence 3: "vals T' ⋅_αset a0' ⊆ ((σ ∘_s α) ‘ fv_transaction T ⋅_set ℐ) ⋅_αset a0'"
  by (simp add: abs_apply_terms_def image_mono)

have "t ⋅ ℐ ⋅_α a0' ∈ timpl_closure_set (α_ik 𝒜 ℐ) (α_ti 𝒜 T σ α ℐ)"
  when "t ∈ ik_lsst 𝒜" for t
  using that abs_in[OF imageI[OF that]]
        α_ti_covers_α_0_ik[OF 𝒜_reach T ℐ σ α P(1)]
        timpl_closure_set_mono[of "{t ⋅ ℐ ⋅_α a0}" "α_ik 𝒜 ℐ" "α_ti 𝒜 T σ α ℐ" "α_ti 𝒜 T σ α ℐ"]
  unfolding a0_def a0'_def T'_def abs_intruder_knowledge_def by fast
hence A: "α_ik (𝒜@T') ℐ ⊆
            timpl_closure_set (α_ik 𝒜 ℐ) (α_ti 𝒜 T σ α ℐ) ∪
            (trms_lsst (transaction_send T) ⋅_set (σ ∘_s α) ⋅_set ℐ) ⋅_αset a0'"
```

```
    using 0(1) 1 by (auto simp add: abs_apply_terms_def)

  have "t · I ·α a0' ∈ timpl_closure_set {t · I ·α a0} (αti A T σ α I)"
    when t: "t ∈ vals_sym A" for t
  proof -
    have "(∃n. t = Fun (Val n) [] ∧ t ∈ subterms_set (trms_lsst A)) ∨
          (∃n. t = Var (TAtom Value,n) ∧ (TAtom Value,n) ∈ fv_lsst A)"
      (is "?P ∨ ?Q")
      using t var_subterm_trms_sst_is_vars_sst[of _ "unlabel A"]
            Γv_TAtom[of Value] reachable_constraints_Value_vars_are_fv[OF A_reach P(1)]
      unfolding vals_sym_def by fast
    thus ?thesis
    proof
      assume ?P
      then obtain n where n: "t = Fun (Val n) []" "t ∈ subterms_set (trms_lsst A)" by moura
      thus ?thesis
        using αti_covers_α0_Val[OF A_reach T I σ α P(1), of n]
        unfolding a0_def a0'_def T'_def by fastforce
    next
      assume ?Q
      thus ?thesis
        using αti_covers_α0_Var[OF A_reach T I σ α P(1)]
        unfolding a0_def a0'_def T'_def by fastforce
    qed
  qed
  moreover have "t · I ·α a0 ∈ αvals A I"
    when "t ∈ vals_sym A" for t
    using that abs_in vals_sym_vals
    unfolding a0_def abs_value_constants_def vals_sym_def vals_def
    by (metis (mono_tags, lifting))
  ultimately have "t · I ·α a0' ∈ timpl_closure_set (αvals A I) (αti A T σ α I)"
    when t: "t ∈ vals_sym A" for t
    using t timpl_closure_set_mono[of "{t · I ·α a0}" "αvals A I" "αti A T σ α I" "αti A T σ α I"]
    by blast
  hence "t ·α a0' ∈ timpl_closure_set (αvals A I) (αti A T σ α I)"
    when t: "t ∈ vals A" for t
    using vals_vals_sym[OF t] by blast
  hence B: "αvals (A@T') I ⊆
              timpl_closure_set (αvals A I) (αti A T σ α I) ∪
              ((σ ∘s α) ` fv_transaction T ·set I) ·αset a0'"
    using 0(2) 3
    by (simp add: abs_apply_terms_def image_subset_iff)

  have 4: "fv (t · σ ∘s α · I ·α a) = {}" for t a
    using I_grounds[of "t · σ ∘s α"] abs_fv[of "t · σ ∘s α · I" a]
    by argo

  have "is_Fun (t · σ ∘s α · I ·α a0')" for t
    using 4[of t a0'] by force
  thus ?A
    using A step_props(1,3)
    unfolding T'_def a0_def a0'_def abs_apply_terms_def
    by blast

  show ?B
    using B step_props(2,4) admissible_transaction_Value_vars[OF bspec[OF P T]]
    by (auto simp add: T'_def a0_def a0'_def abs_apply_terms_def)
qed

lemma reachable_constraints_covered:
  assumes A_reach: "A ∈ reachable_constraints P"
    and I: "welltyped_constraint_model I A"
    and FP:
```

```
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
  shows "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
    and "α_vals 𝒜 ℐ ⊆ absc ‘ set OCC"
using 𝒜_reach ℐ
proof (induction rule: reachable_constraints.induct)
  case init
  { case 1 show ?case by (simp add: abs_intruder_knowledge_def) }
  { case 2 show ?case by (simp add: abs_value_constants_def) }
next
  case (step 𝒜 T σ α)
  { case 1
    hence "welltyped_constraint_model ℐ 𝒜"
      by (metis welltyped_constraint_model_prefix)
    hence IH: "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
              "α_vals 𝒜 ℐ ⊆ absc ‘ set OCC"
      using step.IH by metis+
    show ?case
      using reachable_constraints_covered_step[
              OF step.hyps(1,2) "1.prems" step.hyps(3,4) FP(1,2) IH(1)
                 FP(3) OCC IH(2) TI P transactions_covered]
      by metis
  }
  { case 2
    hence "welltyped_constraint_model ℐ 𝒜"
      by (metis welltyped_constraint_model_prefix)
    hence IH: "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
              "α_vals 𝒜 ℐ ⊆ absc ‘ set OCC"
      using step.IH by metis+
    show ?case
      using reachable_constraints_covered_step[
              OF step.hyps(1,2) "2.prems" step.hyps(3,4) FP(1,2) IH(1)
                 FP(3) OCC IH(2) TI P transactions_covered]
      by metis
  }
qed

lemma attack_in_fixpoint_if_attack_in_ik:
  fixes FP::"('fun,'atom,'sets) prot_terms"
  assumes "∀t ∈ IK ·_αset a. FP ⊢_c t"
    and "attack⟨n⟩ ∈ IK"
  shows "attack⟨n⟩ ∈ FP"
proof -
  have "attack⟨n⟩ ·_α a ∈ IK ·_αset a" by (rule abs_in[OF assms(2)])
  hence "FP ⊢_c attack⟨n⟩ ·_α a" using assms(1) by blast
  moreover have "attack⟨n⟩ ·_α a = attack⟨n⟩" by simp
  ultimately have "FP ⊢_c attack⟨n⟩" by metis
  thus ?thesis using ideduct_synth_priv_const_in_ik[of FP "Attack n"] by simp
qed

lemma attack_in_fixpoint_if_attack_in_timpl_closure_set:
  fixes FP::"('fun,'atom,'sets) prot_terms"
  assumes "attack⟨n⟩ ∈ timpl_closure_set FP TI"
  shows "attack⟨n⟩ ∈ FP"
```

```
proof -
  have "∀f ∈ funs_term (attack⟨n⟩). ¬is_Abs f" by auto
  thus ?thesis using timpl_closure_set_no_Abs_in_set[OF assms] by blast
qed

theorem prot_secure_if_fixpoint_covered_typed:
  assumes FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
    and attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "∄ℐ. welltyped_constraint_model ℐ (𝒜@[(1, send⟨attack⟨n⟩⟩)])" (is "∄ℐ. ?P ℐ")
proof
  assume "∃ℐ. ?P ℐ"
  then obtain ℐ where ℐ: "welltyped_constraint_model ℐ (𝒜@[(1, send⟨attack⟨n⟩⟩)])"
    by moura
  hence ℐ': "constr_sem_stateful ℐ (unlabel (𝒜@[(1, send⟨attack⟨n⟩⟩)]))"
          "interpretation_subst ℐ" "wf_trms (subst_range ℐ)" "wt_subst ℐ"
    unfolding welltyped_constraint_model_def constraint_model_def by metis+

  have 0: "attack⟨n⟩ ∉ ik_lsst 𝒜 ·_set ℐ"
    using welltyped_constraint_model_prefix[OF ℐ]
          reachable_constraints_covered(1)[OF 𝒜 _ FP OCC TI P transactions_covered]
          attack_in_fixpoint_if_attack_in_ik[
            of "ik_lsst 𝒜 ·_set ℐ" "α₀ (db_lsst 𝒜 ℐ)" "timpl_closure_set (set FP) (set TI)" n]
          attack_in_fixpoint_if_attack_in_timpl_closure_set
          attack_notin_FP
    unfolding abs_intruder_knowledge_def by blast

  have 1: "ik_lsst 𝒜 ·_set ℐ ⊢ attack⟨n⟩"
    using ℐ strand_sem_append_stateful[of "{}" "{}" "unlabel 𝒜" _ ℐ]
    unfolding welltyped_constraint_model_def constraint_model_def by force

  have 2: "wf_trms (ik_lsst 𝒜 ·_set ℐ)"
    using reachable_constraints_wf_trms[OF _ 𝒜] admissible_transactions_wf_trms P(1)
          ik_sst_trms_sst_subset[of "unlabel 𝒜"] wf_trms_subst[OF ℐ'(3)]
    by fast

  have 3: "∀x ∈ fv_set (ik_lsst 𝒜). ¬TAtom AttackType ⊑ Γ_v x"
    using reachable_constraints_vars_TAtom_typed[OF 𝒜 P(1)]
          fv_ik_subset_vars_sst'[of "unlabel 𝒜"]
    by fastforce

  have 4: "attack⟨n⟩ ∉ set (snd (Ana t)) ·_set ℐ" when t: "t ∈ subterms_set (ik_lsst 𝒜)" for t
  proof
    assume "attack⟨n⟩ ∈ set (snd (Ana t)) ·_set ℐ"
    then obtain s where s: "s ∈ set (snd (Ana t))" "s · ℐ = attack⟨n⟩" by moura

    obtain x where x: "s = Var x"
      by (cases s) (use s reachable_constraints_no_Ana_Attack[OF 𝒜 P(1) t] in auto)

    have "x ∈ fv t" using x Ana_subterm'[OF s(1)] vars_iff_subtermeq by force
    hence "x ∈ fv_set (ik_lsst 𝒜)" using t fv_subterms by fastforce
    hence "Γ_v x ≠ TAtom AttackType" using 3 by fastforce
```

```
    thus False using s(2) x wt_subst_trm''[OF 𝓘'(4), of "Var x"] by fastforce
  qed

  have 5: "attack⟨n⟩ ∉ set (snd (Ana t))" when t: "t ∈ subterms_set (ik_lsst 𝒜 ·_set 𝓘)" for t
  proof
    assume "attack⟨n⟩ ∈ set (snd (Ana t))"
    then obtain s where s:
        "s ∈ subterms_set (𝓘 ‘ fv_set (ik_lsst 𝒜))" "attack⟨n⟩ ∈ set (snd (Ana s))"
      using Ana_subst_subterms_cases[OF t] 4 by fast
    then obtain x where x: "x ∈ fv_set (ik_lsst 𝒜)" "s ⊑ 𝓘 x" by moura
    hence "𝓘 x ∈ subterms_set (ik_lsst 𝒜 ·_set 𝓘)"
      using var_is_subterm[of x] subterms_subst_subset'[of 𝓘 "ik_lsst 𝒜"]
      by force
    hence *: "wf_trm (𝓘 x)" "wf_trm s"
      using wf_trms_subterms[OF 2] wf_trm_subtermeq[OF _ x(2)]
      by auto

    show False
      using term.order_trans[
              OF subtermeq_imp_subtermtypeeq[OF *(2) Ana_subterm'[OF s(2)]]
                 subtermeq_imp_subtermtypeeq[OF *(1) x(2)]]
            3 x(1) wt_subst_trm''[OF 𝓘'(4), of "Var x"]
      by force
  qed

  show False
    using 0 private_const_deduct[OF _ 1] 5
    by simp
qed

end
```

2.6.4 Theorem: A Protocol is Secure if it is Covered by a Fixed-Point

```
context stateful_protocol_model
begin

theorem prot_secure_if_fixpoint_covered:
  fixes P
  assumes FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ‘ set OCC"
      "timpl_closure_set (absc ‘ set OCC) (set TI) ⊆ absc ‘ set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and M:
      "has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
    and attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    and A: "𝒜 ∈ reachable_constraints P"
  shows "♯𝓘. constraint_model 𝓘 (𝒜@[(1, send⟨attack⟨n⟩⟩)])"
    (is "♯𝓘. ?P 𝒜 𝓘")
proof
  assume "∃𝓘. ?P 𝒜 𝓘"
```

```
  then obtain 𝓘 where I:
      "interpretation_subst 𝓘" "wf_trms (subst_range 𝓘)"
      "constr_sem_stateful 𝓘 (unlabel (𝒜@[(1, send⟨attack⟨n⟩⟩)]))"
    unfolding constraint_model_def by moura

  let ?n = "[(1, send⟨attack⟨n⟩⟩)]"
  let ?A = "𝒜@?n"

  have "∀T ∈ set P. wellformed_transaction T"
       "∀T ∈ set P. admissible_transaction_terms T"
    using P(1) unfolding admissible_transaction_def by blast+
  moreover have "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    using P(1) unfolding admissible_transaction_def admissible_transaction_terms_def by blast
  ultimately have 0: "wf_sst (unlabel 𝒜)" "tfr_sst (unlabel 𝒜)" "wf_trms (trms_lsst 𝒜)"
    using reachable_constraints_tfr[OF _ M P A] reachable_constraints_wf[OF _ _ A] by metis+

  have 1: "wf_sst (unlabel ?A)" "tfr_sst (unlabel ?A)" "wf_trms (trms_lsst ?A)"
  proof -
    show "wf_sst (unlabel ?A)"
      using 0(1) wf_sst_append_suffix'[of "{}" "unlabel 𝒜" "unlabel ?n"] unlabel_append[of 𝒜 ?n]
      by simp

    show "wf_trms (trms_lsst ?A)"
      using 0(3) trms_sst_append[of "unlabel 𝒜" "unlabel ?n"] unlabel_append[of 𝒜 ?n]
      by fastforce

    have "∀t ∈ trms_lsst ?n ∪ pair ` setops_sst (unlabel ?n). ∃c. t = Fun c []"
         "∀t ∈ trms_lsst ?n ∪ pair ` setops_sst (unlabel ?n). Ana t = ([],[])"
      by (simp_all add: setops_sst_def)
    hence "tfr_set (trms_lsst 𝒜 ∪ pair ` setops_sst (unlabel 𝒜) ∪
                (trms_lsst ?n ∪ pair ` setops_sst (unlabel ?n)))"
      using 0(2) tfr_consts_mono unfolding tfr_sst_def by blast
    hence "tfr_set (trms_lsst (𝒜@?n) ∪ pair ` setops_sst (unlabel (𝒜@?n)))"
      using unlabel_append[of 𝒜 ?n] trms_sst_append[of "unlabel 𝒜" "unlabel ?n"]
            setops_sst_append[of "unlabel 𝒜" "unlabel ?n"]
      by (simp add: setops_sst_def)
    thus "tfr_sst (unlabel ?A)"
      using 0(2) unlabel_append[of ?A ?n]
      unfolding tfr_sst_def by auto
  qed

  obtain 𝓘_τ where I':
      "welltyped_constraint_model 𝓘_τ ?A"
    using stateful_typing_result[OF 1 I(1,3)]
    by (metis welltyped_constraint_model_def constraint_model_def)

  note a = FP_OCC_TI P(1) transactions_covered attack_notin_FP A

  show False
    using prot_secure_if_fixpoint_covered_typed[OF a] I'
    by force
qed

end
```

2.6.5 Automatic Fixed-Point Computation

```
context stateful_protocol_model
begin

definition compute_fixpoint_fun' where
  "compute_fixpoint_fun' P (n::nat option) enable_traces S0 ≡
    let sy = intruder_synth_mod_timpls;
```

```
      FP' = λS. fst (fst S);
      TI' = λS. snd (fst S);
      OCC' = λS. remdups (
        (map (λt. the_Abs (the_Fun (args t ! 1)))
             (filter (λt. is_Fun t ∧ the_Fun t = OccursFact) (FP' S)))@
        (map snd (TI' S)));

      equal_states = λS S'. set (FP' S) = set (FP' S') ∧ set (TI' S) = set (TI' S');

      trace' = λS. snd S;

      close = λM f. let g = remdups ∘ f in while (λA. set (g A) ≠ set A) g M;
      close' = λM f. let g = remdups ∘ f in while (λA. set (g A) ≠ set A) g M;
      trancl_minus_refl = λTI.
        let aux = λts p. map (λq. (fst p,snd q)) (filter ((=) (snd p) ∘ fst) ts)
        in filter (λp. fst p ≠ snd p) (close' TI (λts. concat (map (aux ts) ts)@ts));
      snd_Ana = λN M TI. let N' = filter (λt. ∀k ∈ set (fst (Ana t)). sy M TI k) N in
        filter (λt. ¬sy M TI t)
               (concat (map (λt. filter (λs. s ∈ set (snd (Ana t))) (args t)) N'));
      Ana_cl = λFP TI.
        close FP (λM. (M@snd_Ana M M TI));
      TI_cl = λFP TI.
        close FP (λM. (M@filter (λt. ¬sy M TI t)
                                (concat (map (λm. concat (map (λ(a,b). ⟨a -->> b⟩⟨m⟩) TI)) M))));
      Ana_cl' = λFP TI.
        let N = λM. comp_timpl_closure_list (filter (λt. ∃k∈set (fst (Ana t)). ¬sy M TI k) M) TI
        in close FP (λM. M@snd_Ana (N M) M TI);

      Δ = λS. transaction_check_comp (FP' S) (OCC' S) (TI' S);
      result = λS T δ.
        let not_fresh = λx. x ∉ set (transaction_fresh T);
            xs = filter not_fresh (fv_list_sst (unlabel (transaction_strand T)));
            u = λδ x. absdbupd (unlabel (transaction_strand T)) x (δ x)
        in (remdups (filter (λt. ¬sy (FP' S) (TI' S) t)
                            (map (λt. the_msg t · (absc ∘ u δ))
                                 (filter is_Send (unlabel (transaction_send T))))),
            remdups (filter (λs. fst s ≠ snd s) (map (λx. (δ x, u δ x)) xs)));
      update_state = λS. if list_ex (λt. is_Fun t ∧ is_Attack (the_Fun t)) (FP' S) then S
        else let results = map (λT. map (λδ. result S T (abs_substs_fun δ)) (Δ S T)) P;
                 newtrace_flt = (λn. let x = results ! n; y = map fst x; z = map snd x
                  in set (concat y) - set (FP' S) ≠ {} ∨ set (concat z) - set (TI' S) ≠ {});
                 trace =
                  if enable_traces
                  then trace' S@[filter newtrace_flt [0..<length results]]
                  else [];
                 U = concat results;
                 V = ((remdups (concat (map fst U)@FP' S),
                       remdups (filter (λx. fst x ≠ snd x) (concat (map snd U)@TI' S))),
                      trace);
                 W = ((Ana_cl (TI_cl (FP' V) (TI' V)) (TI' V),
                       trancl_minus_refl (TI' V)),
                      trace' V)
        in if ¬equal_states W S then W
        else ((Ana_cl' (FP' W) (TI' W), TI' W), trace' W);

      S = ((λh. case n of None ⇒ while (λS. ¬equal_states S (h S)) h | Some m ⇒ h ^^ m)
           update_state S0)
  in ((FP' S, OCC' S, TI' S), trace' S)"

definition compute_fixpoint_fun where
  "compute_fixpoint_fun P ≡ fst (compute_fixpoint_fun' P None False (([],[]),[]))"
```

end

2.6.6 Locales for Protocols Proven Secure through Fixed-Point Coverage

```
type_synonym ('f,'a,'s) fixpoint_triple =
  "('f,'a,'s) prot_term list × 's set list × ('s set × 's set) list"

context stateful_protocol_model
begin

definition "attack_notin_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) ≡
  list_all (λt. ∀f ∈ funs_term t. ¬is_Attack f) (fst FPT)"

definition "protocol_covered_by_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) P ≡
  let (FP, OCC, TI) = FPT
  in list_all (transaction_check FP OCC TI) P"

definition "analyzed_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) ≡
  let (FP, _, TI) = FPT
  in analyzed_closed_mod_timpls FP TI"

definition "wellformed_protocol' (P::('fun,'atom,'sets,'lbl) prot) N ≡
  list_all admissible_transaction P ∧
  has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) (set N) ∧
  comp_tfr_set arity Ana Γ N ∧
  list_all (λT. list_all (comp_tfr_sstp Γ Pair) (unlabel (transaction_strand T))) P"

definition "wellformed_protocol (P::('fun,'atom,'sets,'lbl) prot) ≡
  let f = λM. remdups (concat (map subterms_list M@map (fst ∘ Ana) M));
      N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P));
      N = while (λA. set (f A) ≠ set A) f N0
  in wellformed_protocol' P N"

definition "wellformed_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) ≡
  let (FP, OCC, TI) = FPT; OCC' = set OCC
  in list_all (λt. wf_trm' arity t ∧ fv t = {}) FP ∧
     list_all (λa. a ∈ OCC') (map snd TI) ∧
     list_all (λ(a,b). list_all (λ(c,d). b = c ∧ a ≠ d ⟶ List.member TI (a,d)) TI) TI ∧
     list_all (λp. fst p ≠ snd p) TI ∧
     list_all (λt. ∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ OCC') FP"

lemma protocol_covered_by_fixpoint_I1[intro]:
  assumes "list_all (protocol_covered_by_fixpoint FPT) P"
  shows "protocol_covered_by_fixpoint FPT (concat P)"
using assms by (auto simp add: protocol_covered_by_fixpoint_def list_all_iff)

lemma protocol_covered_by_fixpoint_I2[intro]:
  assumes "protocol_covered_by_fixpoint FPT P1"
    and "protocol_covered_by_fixpoint FPT P2"
  shows "protocol_covered_by_fixpoint FPT (P1@P2)"
using assms by (auto simp add: protocol_covered_by_fixpoint_def)

lemma protocol_covered_by_fixpoint_I3[intro]:
  assumes "∀T ∈ set P. ∀δ::('fun,'atom,'sets) prot_var ⇒ 'sets set.
    transaction_check_pre FP TI T δ ⟶ transaction_check_post FP TI T δ"
  shows "protocol_covered_by_fixpoint (FP,OCC,TI) P"
using assms
unfolding protocol_covered_by_fixpoint_def transaction_check_def transaction_check_comp_def
          list_all_iff Let_def case_prod_unfold Product_Type.fst_conv Product_Type.snd_conv
by fastforce

lemmas protocol_covered_by_fixpoint_intros =
  protocol_covered_by_fixpoint_I1
```

```
  protocol_covered_by_fixpoint_I2
  protocol_covered_by_fixpoint_I3

lemma prot_secure_if_prot_checks:
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
  assumes attack_notin_fixpoint: "attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint: "analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "wellformed_protocol' P N"
    and wellformed_fixpoint: "wellformed_fixpoint FP_OCC_TI"
  shows "∀𝒜 ∈ reachable_constraints P. ∄ℐ. constraint_model ℐ (𝒜@[(1, send⟨attack⟨n⟩⟩)])"
proof -
  define FP where "FP ≡ let (FP,_,_) = FP_OCC_TI in FP"
  define OCC where "OCC ≡ let (_,OCC,_) = FP_OCC_TI in OCC"
  define TI where "TI ≡ let (_,_,TI) = FP_OCC_TI in TI"

  have attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    using attack_notin_fixpoint[unfolded attack_notin_fixpoint_def]
    unfolding list_all_iff FP_def by force

  have 1: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
    using wellformed_fixpoint
    unfolding wellformed_fixpoint_def wf_trms_code[symmetric] Let_def TI_def
              list_all_iff member_def case_prod_unfold
    by auto

  have 0: "wf_trms (set FP)"
    and 2: "∀(a,b) ∈ set TI. a ≠ b"
    and 3: "snd ` set TI ⊆ set OCC"
    and 4: "∀t ∈ set FP. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
    and 5: "ground (set FP)"
    using wellformed_fixpoint
    unfolding wellformed_fixpoint_def wf_trm_code[symmetric] is_Abs_def the_Abs_def
              list_all_iff Let_def case_prod_unfold set_map FP_def OCC_def TI_def
    by (fast, fast, blast, fastforce, simp)

  have 8: "finite (set N)"
    and 9: "has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) (set N)"
    and 10: "tfr_set (set N)"
    and 11: "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 12: "∀T ∈ set P. admissible_transaction T"
    using wellformed_protocol tfr_set_if_comp_tfr_set[of N]
    unfolding Let_def list_all_iff wellformed_protocol_def wellformed_protocol'_def
              wf_trms_code[symmetric] tfr_sstp_is_comp_tfr_sstp[symmetric]
    by fast+

  have 13: "wf_trms (set N)"
    using wellformed_protocol
    unfolding wellformed_protocol_def wellformed_protocol'_def
              wf_trm_code[symmetric] comp_tfr_set_def list_all_iff
              finite_SMP_representation_def
    by blast

  note TI0 = trancl_eqI'[OF 1 2]

  have "analyzed (timpl_closure_set (set FP) (set TI))"
    using analyzed_fixpoint[unfolded analyzed_fixpoint_def]
          analyzed_closed_mod_timpls_is_analyzed_timpl_closure_set[OF TI0 0]
    unfolding FP_def TI_def
    by force
  note FP0 = this 0 5
```

```
  note OCC0 = funs_term_OCC_TI_subset(1)[OF 4 3]
              timpl_closure_set_supset'[OF funs_term_OCC_TI_subset(2)[OF 4 3]]

  note M0 = 9 8 10 13

  have "list_all (transaction_check FP OCC TI) P"
    using transactions_covered[unfolded protocol_covered_by_fixpoint_def]
    unfolding FP_def OCC_def TI_def
    by force
  note P0 = 12 11 this attack_notin_FP

  show ?thesis by (metis prot_secure_if_fixpoint_covered[OF FP0 OCC0 TI0 M0 P0])
qed

end

locale secure_stateful_protocol =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
    and P_SMP::"('fun, 'atom, 'sets) prot_term list"
  assumes attack_notin_fixpoint: "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "pm.protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint: "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "pm.wellformed_protocol' P P_SMP"
    and wellformed_fixpoint: "pm.wellformed_fixpoint FP_OCC_TI"
begin

theorem protocol_secure:
  "∀ A ∈ pm.reachable_constraints P. ∄I. pm.constraint_model I (A@[(1, send⟨attack⟨n⟩⟩)])"
by (rule pm.prot_secure_if_prot_checks[OF
            attack_notin_fixpoint transactions_covered
            analyzed_fixpoint wellformed_protocol wellformed_fixpoint])

end

locale secure_stateful_protocol' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
  assumes attack_notin_fixpoint': "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered': "pm.protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint': "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol': "pm.wellformed_protocol P"
    and wellformed_fixpoint': "pm.wellformed_fixpoint FP_OCC_TI"
begin
```

```
sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P
  FP_OCC_TI
  "let f = λM. remdups (concat (map subterms_list M@map (fst ∘ pm.Ana) M));
       N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P))
   in while (λA. set (f A) ≠ set A) f N0"
apply unfold_locales
using attack_notin_fixpoint' transactions_covered' analyzed_fixpoint'
      wellformed_protocol'[unfolded pm.wellformed_protocol_def Let_def] wellformed_fixpoint'
unfolding Let_def by blast+

end

locale secure_stateful_protocol'' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ (('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes checks: "let FPT = pm.compute_fixpoint_fun P
    in pm.attack_notin_fixpoint FPT ∧ pm.protocol_covered_by_fixpoint FPT P ∧
       pm.analyzed_fixpoint FPT ∧ pm.wellformed_protocol P ∧ pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P "pm.compute_fixpoint_fun P"
using checks[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ (('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
    and P_SMP::"('fun, 'atom, 'sets) prot_term list"
  assumes checks': "let P' = P; FPT = FP_OCC_TI; P'_SMP = P_SMP
                    in pm.attack_notin_fixpoint FPT ∧
                       pm.protocol_covered_by_fixpoint FPT P' ∧
                       pm.analyzed_fixpoint FPT ∧
                       pm.wellformed_protocol' P' P'_SMP ∧
                       pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI P_SMP
using checks'[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end

locale secure_stateful_protocol'''' =
```

```
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ (('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
  assumes checks'': "let P' = P; FPT = FP_OCC_TI
                     in pm.attack_notin_fixpoint FPT ∧
                        pm.protocol_covered_by_fixpoint FPT P' ∧
                        pm.analyzed_fixpoint FPT ∧
                        pm.wellformed_protocol P' ∧
                        pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI
using checks''[unfolded Let_def case_prod_unfold] by unfold_locales meson+

end
```

2.6.7 Automatic Protocol Composition

```
context stateful_protocol_model
begin

definition wellformed_composable_protocols where
  "wellformed_composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) N ≡
    let
      Ts = concat P;
      steps = concat (map transaction_strand Ts);
      MP0 = ⋃T ∈ set Ts. trms_transaction T ∪ pair' Pair ‘ setops_transaction T
    in
      list_all (wf_trm' arity) N ∧
      has_all_wt_instances_of Γ MP0 (set N) ∧
      comp_tfr_set arity Ana Γ N ∧
      list_all (comp_tfr_sstp Γ Pair ∘ snd) steps ∧
      list_all (λT. wellformed_transaction T) Ts ∧
      list_all (λT. wf_trms' arity (trms_transaction T)) Ts ∧
      list_all (λT. list_all (λx. Γ_v x = TAtom Value) (transaction_fresh T)) Ts"

definition composable_protocols where
  "composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) Ms S ≡
    let
      Ts = concat P;
      steps = concat (map transaction_strand Ts);
      MP0 = ⋃T ∈ set Ts. trms_transaction T ∪ pair' Pair ‘ setops_transaction T;
      M_fun = (λl. case find ((=) l ∘ fst) Ms of Some M ⇒ snd M | None ⇒ [])
    in comp_par_comp_lsst public arity Ana Γ Pair steps M_fun S"

lemma composable_protocols_par_comp_constr:
  fixes S f
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Sec ≡ (f (set S)) - {m. intruder_synth {} m}"
  assumes Ps_pc: "wellformed_composable_protocols Ps N" "composable_protocols Ps Ms S"
  shows "∀A ∈ reachable_constraints (concat Ps). ∀I. constraint_model I A ⟶
           (∃I_τ. welltyped_constraint_model I_τ A ∧
                 ((∀n. welltyped_constraint_model I_τ (proj n A)) ∨
                  (∃A'. prefix A' A ∧ strand_leaks_lsst A' Sec I_τ)))"
```

```
    (is "∀𝒜 ∈ _. ∀_. _ ⟶ ?Q 𝒜 ℐ")
proof (intro allI ballI impI)
  fix 𝒜 ℐ
  assume 𝒜: "𝒜 ∈ reachable_constraints (concat Ps)" and ℐ: "constraint_model ℐ 𝒜"

  let ?Ts = "concat Ps"
  let ?steps = "concat (map transaction_strand ?Ts)"
  let ?MP0 = "⋃T ∈ set ?Ts. trms_transaction T ∪ pair' Pair ‘ setops_transaction T"
  let ?M_fun = "λl. case find ((=) l ∘ fst) Ms of Some M ⇒ snd M | None ⇒ []"

  have M:
      "has_all_wt_instances_of Γ ?MP0 (set N)"
      "finite (set N)" "tfr_set (set N)" "wf_trms (set N)"
    using Ps_pc tfr_set_if_comp_tfr_set[of N]
    unfolding composable_protocols_def wellformed_composable_protocols_def
              Let_def list_all_iff wf_trm_code[symmetric]
    by fast+

  have P:
      "∀T ∈ set ?Ts. wellformed_transaction T"
      "∀T ∈ set ?Ts. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set ?Ts. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set ?Ts. list_all tfr_sstp (unlabel (transaction_strand T))"
      "comp_par_comp_lsst public arity Ana Γ Pair ?steps ?M_fun S"
    using Ps_pc tfr_sstp_is_comp_tfr_sstp
    unfolding wellformed_composable_protocols_def composable_protocols_def
              Let_def list_all_iff unlabel_def wf_trms_code[symmetric]
    by (meson, meson, meson, fastforce, blast)

  show "?Q 𝒜 ℐ"
    using reachable_constraints_par_comp_constr[OF M P 𝒜 ℐ]
    unfolding Sec_def f_def by fast
qed

end

end
```

3 Trac Support and Automation

3.1 Useful Eisbach Methods for Automating Protocol Verification (Eisbach_Protocol_Verification)

```
theory Eisbach_Protocol_Verification
  imports Main "HOL-Eisbach.Eisbach_Tools"
begin

named_theorems exhausts
named_theorems type_class_instance_lemmata
named_theorems protocol_checks
named_theorems coverage_check_unfold_protocol_lemma
named_theorems coverage_check_unfold_lemmata
named_theorems coverage_check_intro_lemmata
named_theorems transaction_coverage_lemmata

method UNIV_lemma =
  (rule UNIV_eq_I; (subst insert_iff)+; subst empty_iff; smt exhausts)+

method type_class_instance =
  (intro_classes; auto simp add: type_class_instance_lemmata)

method protocol_model_subgoal =
  (((rule allI, case_tac f); (erule forw_subst)+)?; simp_all)

method protocol_model_interpretation =
  (unfold_locales; protocol_model_subgoal+)

method check_protocol_intro =
  (unfold_locales, unfold protocol_checks[symmetric])

method check_protocol_with methods meth =
  (check_protocol_intro, meth)

method check_protocol' =
  (check_protocol_with ‹code_simp+›)

method check_protocol_unsafe' =
  (check_protocol_with ‹eval+›)

method check_protocol =
  (check_protocol_with ‹
    code_simp,
    code_simp,
    code_simp,
    code_simp,
    code_simp›)

method check_protocol_unsafe =
  (check_protocol_with ‹
    eval,
    eval,
    eval,
    eval,
    eval›)
```

```
method coverage_check_intro =
  (((unfold coverage_check_unfold_protocol_lemma)?;
    intro coverage_check_intro_lemmata;
    simp only: list_all_simps list_all_append list.map concat.simps map_append product_concat_map;
    intro conjI TrueI);
   (clarsimp+)?;
   ((rule transaction_coverage_lemmata)+)?)

method coverage_check_unfold =
  (unfold coverage_check_unfold_protocol_lemma coverage_check_unfold_lemmata
          list_all_iff Let_def case_prod_unfold Product_Type.fst_conv Product_Type.snd_conv)

end
```

3.2 ML Yacc Library (ml_yacc_lib)

```
theory
  "ml_yacc_lib"
  imports
  Main
begin
ML_file "ml-yacc-lib/base.sig"
ML_file "ml-yacc-lib/join.sml"
ML_file "ml-yacc-lib/lrtable.sml"
ML_file "ml-yacc-lib/stream.sml"
ML_file "ml-yacc-lib/parser2.sml"


end
```

3.3 Abstract Syntax for Trac Terms (trac_term)

```
theory
  trac_term
  imports
    "First_Order_Terms.Term"
    "ml_yacc_lib"

begin
datatype cMsg = cVar "string * string"
              | cConst string
              | cFun "string * cMsg list"

ML‹
structure Trac_Utils =
struct

  fun list_find p ts =
    let
      fun aux _ [] = NONE
        | aux n (t::ts) =
            if p t
            then SOME (t,n)
            else aux (n+1) ts
    in
      aux 0 ts
    end

  fun map_prod f (a,b) = (f a, f b)
```

```
fun list_product [] = [[]]
  | list_product (xs::xss) =
      List.concat (map (fn x => map (fn ys => x::ys) (list_product xss)) xs)

fun list_toString elem_toString xs =
  let
    fun aux [] = ""
      | aux [x] = elem_toString x
      | aux (x::y::xs) = elem_toString x ^ ", " ^ aux (y::xs)
  in
    "[" ^ aux xs ^ "]"
  end

val list_to_str = list_toString (fn x => x)

fun list_triangle_product _ [] = []
  | list_triangle_product f (x::xs) = map (f x) xs@list_triangle_product f xs

fun list_subseqs [] = [[]]
  | list_subseqs (x::xs) = let val xss = list_subseqs xs in map (cons x) xss@xss end

fun list_intersect xs ys =
    List.exists (fn x => member (op =) ys x) xs orelse
    List.exists (fn y => member (op =) xs y) ys

fun list_partitions xs constrs =
  let
    val peq = eq_set (op =)
    val pseq = eq_set peq
    val psseq = eq_set pseq

    fun illegal p q =
      let
        val pq = union (op =) p q
        fun f (a,b) = member (op =) pq a andalso member (op =) pq b
      in
        List.exists f constrs
      end

    fun merges _ [] = []
      | merges q (p::ps) =
          if illegal p q then map (cons p) (merges q ps)
          else (union (op =) p q::ps)::(map (cons p) (merges q ps))

    fun merges_all [] = []
      | merges_all (p::ps) = merges p ps@map (cons p) (merges_all ps)

    fun step pss = fold (union pseq) (map merges_all pss) []

    fun loop pss pssprev =
      let val pss' = step pss
      in if psseq (pss,pss') then pssprev else loop pss' (union pseq pss' pssprev)
      end

    val init = [map single xs]
  in
    loop init init
  end

fun mk_unique  [] = []
  | mk_unique (x::xs) = x::mk_unique(List.filter (fn y => y <> x) xs)
```

```
  fun list_rm_pair sel l x = filter (fn e => sel e <> x) l

  fun list_minus list_rm l m = List.foldl (fn (a,b) => list_rm b a) l m

  fun list_upto n =
    let
      fun aux m = if m >= n then [] else m::aux (m+1)
    in
      aux 0
    end
end
```

ML

```
structure Trac_Term (* : TRAC_TERM *) =
struct
open Trac_Utils
exception TypeError

type TypeDecl = string * string

datatype Msg = Var of string
             | Const of string
             | Fun of string * Msg list
             | Attack

datatype VarType = EnumType of string
                 | ValueType
                 | Untyped

datatype cMsg = cVar of string * VarType
              | cConst of string
              | cFun of string * cMsg list
              | cAttack
              | cSet of string * cMsg list
              | cAbs of (string * string list) list
              | cOccursFact of cMsg
              | cPrivFunSec
              | cEnum of string

fun type_of et vt n =
  case List.find (fn (v,_) => v = n) et of
    SOME (_,t) => EnumType t
  | NONE =>
      if List.exists (fn v => v = n) vt
      then ValueType
      else Untyped

fun certifyMsg et vt (Var n)       = cVar (n, type_of et vt n)
  | certifyMsg _  _  (Const c)     = cConst c
  | certifyMsg et vt (Fun (f, ts)) = cFun (f, map (certifyMsg et vt) ts)
  | certifyMsg _  _  Attack        = cAttack

fun mk_Value_cVar x = cVar (x,ValueType)

val fv_Msg =
  let
    fun aux (Var x) = [x]
      | aux (Fun (_,ts)) = List.concat (map aux ts)
      | aux _ = []
  in
    mk_unique o  aux
```

```
  end

val fv_cMsg =
  let
    fun aux (cVar x) = [x]
      | aux (cFun (_,ts)) = List.concat (map aux ts)
      | aux (cSet (_,ts)) = List.concat (map aux ts)
      | aux (cOccursFact bs) = aux bs
      | aux _ = []
  in
    mk_unique o aux
  end

fun subst_apply' (delta:(string * VarType) -> cMsg) (t:cMsg) =
  case t of
    cVar x => delta x
  | cFun (f,ts) => cFun (f, map (subst_apply' delta) ts)
  | cSet (s,ts) => cSet (s, map (subst_apply' delta) ts)
  | cOccursFact bs => cOccursFact (subst_apply' delta bs)
  | c => c

fun subst_apply (delta:(string * cMsg) list) =
  subst_apply' (fn (n,tau) => (
      case List.find (fn x => fst x = n) delta of
        SOME x => snd x
      | NONE => cVar (n,tau)))
end
```

ML

```
structure TracProtocol (* : TRAC_TERM *) =
struct
open Trac_Utils
datatype type_spec_elem =
  Consts of string list
| Union of string list

fun is_Consts t = case t of Consts _ => true | _ => false
fun the_Consts t = case t of Consts cs => cs | _ => error "Consts"

type type_spec = (string * type_spec_elem) list
type set_spec  = (string * string)

fun extract_Consts (tspec:type_spec) =
  (List.concat o map the_Consts o filter is_Consts o map snd) tspec

type funT = (string * string)
type fun_spec = {private: funT list, public: funT list}

type ruleT = (string * string list) * Trac_Term.Msg list * string list
type anaT = ruleT list

datatype prot_label = LabelN | LabelS

datatype action = RECEIVE of Trac_Term.Msg
                | SEND of Trac_Term.Msg
                | IN of Trac_Term.Msg * (string * Trac_Term.Msg list)
                | NOTIN of Trac_Term.Msg * (string * Trac_Term.Msg list)
                | NOTINANY of Trac_Term.Msg * string
                | INSERT of Trac_Term.Msg * (string * Trac_Term.Msg list)
                | DELETE of Trac_Term.Msg * (string * Trac_Term.Msg list)
                | NEW of string
```

```
                | ATTACK

datatype cAction = cReceive of Trac_Term.cMsg
                 | cSend of Trac_Term.cMsg
                 | cInequality of Trac_Term.cMsg * Trac_Term.cMsg
                 | cInSet of Trac_Term.cMsg * Trac_Term.cMsg
                 | cNotInSet of Trac_Term.cMsg * Trac_Term.cMsg
                 | cNotInAny of Trac_Term.cMsg * string
                 | cInsert of Trac_Term.cMsg * Trac_Term.cMsg
                 | cDelete of Trac_Term.cMsg * Trac_Term.cMsg
                 | cNew of string
                 | cAssertAttack

type transaction_name = string * (string * string) list * (string * string) list

type transaction={transaction:transaction_name,actions:(prot_label * action) list}

type cTransaction={
  transaction:transaction_name,
  receive_actions:(prot_label * cAction) list,
  checksingle_actions:(prot_label * cAction) list,
  checkall_actions:(prot_label * cAction) list,
  fresh_actions:(prot_label * cAction) list,
  update_actions:(prot_label * cAction) list,
  send_actions:(prot_label * cAction) list,
  attack_actions:(prot_label * cAction) list}

fun mkTransaction transaction actions = {transaction=transaction,
                                         actions=actions}:transaction

fun is_RECEIVE a = case a of RECEIVE _ => true | _ => false
fun is_SEND a = case a of SEND _ => true | _ => false
fun is_IN a = case a of IN _ => true | _ => false
fun is_NOTIN a = case a of NOTIN _ => true | _ => false
fun is_NOTINANY a = case a of NOTINANY _ => true | _ => false
fun is_INSERT a = case a of INSERT _ => true | _ => false
fun is_DELETE a = case a of DELETE _ => true | _ => false
fun is_NEW a = case a of NEW _ => true | _ => false
fun is_ATTACK a = case a of ATTACK => true | _ => false

fun the_RECEIVE a = case a of RECEIVE t => t | _ => error "RECEIVE"
fun the_SEND a = case a of SEND t => t | _ => error "SEND"
fun the_IN a = case a of IN t => t | _ => error "IN"
fun the_NOTIN a = case a of NOTIN t => t | _ => error "NOTIN"
fun the_NOTINANY a = case a of NOTINANY t => t | _ => error "NOTINANY"
fun the_INSERT a = case a of INSERT t => t | _ => error "INSERT"
fun the_DELETE a = case a of DELETE t => t | _ => error "DELETE"
fun the_NEW a = case a of NEW t => t | _ => error "FRESH"

fun maybe_the_RECEIVE a = case a of RECEIVE t => SOME t | _ => NONE
fun maybe_the_SEND a = case a of SEND t => SOME t | _ => NONE
fun maybe_the_IN a = case a of IN t => SOME t | _ => NONE
fun maybe_the_NOTIN a = case a of NOTIN t => SOME t | _ => NONE
fun maybe_the_NOTINANY a = case a of NOTINANY t => SOME t | _ => NONE
fun maybe_the_INSERT a = case a of INSERT t => SOME t | _ => NONE
fun maybe_the_DELETE a = case a of DELETE t => SOME t | _ => NONE
fun maybe_the_NEW a = case a of NEW t => SOME t | _ => NONE

fun is_Receive a = case a of cReceive _ => true | _ => false
fun is_Send a = case a of cSend _ => true | _ => false
fun is_Inequality a = case a of cInequality _ => true | _ => false
fun is_InSet a = case a of cInSet _ => true | _ => false
fun is_NotInSet a = case a of cNotInSet _ => true | _ => false
```

```
fun is_NotInAny a = case a of cNotInAny _ => true | _ => false
fun is_Insert a = case a of cInsert _ => true | _ => false
fun is_Delete a = case a of cDelete _ => true | _ => false
fun is_Fresh a = case a of cNew _ => true | _ => false
fun is_Attack a = case a of cAssertAttack => true | _ => false

fun the_Receive a = case a of cReceive t => t | _ => error "Receive"
fun the_Send a = case a of cSend t => t | _ => error "Send"
fun the_Inequality a = case a of cInequality t => t | _ => error "Inequality"
fun the_InSet a = case a of cInSet t => t | _ => error "InSet"
fun the_NotInSet a = case a of cNotInSet t => t | _ => error "NotInSet"
fun the_NotInAny a = case a of cNotInAny t => t | _ => error "NotInAny"
fun the_Insert a = case a of cInsert t => t | _ => error "Insert"
fun the_Delete a = case a of cDelete t => t | _ => error "Delete"
fun the_Fresh a = case a of cNew t => t | _ => error "New"

fun maybe_the_Receive a = case a of cReceive t => SOME t | _ => NONE
fun maybe_the_Send a = case a of cSend t => SOME t | _ => NONE
fun maybe_the_Inequality a = case a of cInequality t => SOME t | _ => NONE
fun maybe_the_InSet a = case a of cInSet t => SOME t | _ => NONE
fun maybe_the_NotInSet a = case a of cNotInSet t => SOME t | _ => NONE
fun maybe_the_NotInAny a = case a of cNotInAny t => SOME t | _ => NONE
fun maybe_the_Insert a = case a of cInsert t => SOME t | _ => NONE
fun maybe_the_Delete a = case a of cDelete t => SOME t | _ => NONE
fun maybe_the_Fresh a = case a of cNew t => SOME t | _ => NONE

fun certifyAction et vt (lbl,SEND t)            = (lbl,cSend    (Trac_Term.certifyMsg et vt t))
  | certifyAction et vt (lbl,RECEIVE t)         = (lbl,cReceive (Trac_Term.certifyMsg et vt t))
  | certifyAction et vt (lbl,IN (x,(s,ps)))     = (lbl,cInSet
      (Trac_Term.certifyMsg et vt x, Trac_Term.cSet (s, map (Trac_Term.certifyMsg et vt) ps)))
  | certifyAction et vt (lbl,NOTIN (x,(s,ps)))  = (lbl,cNotInSet
      (Trac_Term.certifyMsg et vt x, Trac_Term.cSet (s, map (Trac_Term.certifyMsg et vt) ps)))
  | certifyAction et vt (lbl,NOTINANY (x,s))    = (lbl,cNotInAny (Trac_Term.certifyMsg et vt x, s))
  | certifyAction et vt (lbl,INSERT (x,(s,ps))) = (lbl,cInsert
      (Trac_Term.certifyMsg et vt x, Trac_Term.cSet (s, map (Trac_Term.certifyMsg et vt) ps)))
  | certifyAction et vt (lbl,DELETE (x,(s,ps))) = (lbl,cDelete
      (Trac_Term.certifyMsg et vt x, Trac_Term.cSet (s, map (Trac_Term.certifyMsg et vt) ps)))
  | certifyAction _  _  (lbl,NEW x)             = (lbl,cNew x)
  | certifyAction _  _  (lbl,ATTACK)            = (lbl,cAssertAttack)

fun certifyTransaction (tr:transaction) =
  let
    val mk_cOccurs = Trac_Term.cOccursFact
    fun mk_Value_cVar x = Trac_Term.cVar (x,Trac_Term.ValueType)
    fun mk_cInequality x y = cInequality (mk_Value_cVar x, mk_Value_cVar y)
    val mk_cInequalities = list_triangle_product mk_cInequality

    val fresh_vals = map_filter (maybe_the_NEW o snd) (#actions tr)
    val decl_vars = map fst (#2 (#transaction tr))
    val neq_constrs = #3 (#transaction tr)

    val _ = if     List.exists (fn x => List.exists (fn y => x = y) fresh_vals) decl_vars
            orelse List.exists (fn x => List.exists (fn y => x = y) decl_vars)  fresh_vals
            then error "the fresh and the declared variables must not overlap"
            else ()

    val _ = case List.find (fn (x,y) => x = y) neq_constrs of
              SOME (x,y) => error ("illegal inequality constraint: " ^ x ^ " != " ^ y)
            | NONE => ()

    val nonfresh_vals = map fst (filter (fn x => snd x = "value") (#2 (#transaction tr)))
    val enum_vars = filter (fn x => snd x <> "value") (#2 (#transaction tr))
```

```
    fun lblS t = (LabelS,t)

    val cactions = map (certifyAction enum_vars (nonfresh_vals@fresh_vals)) (#actions tr)

    val nonfresh_occurs = map (lblS o cReceive o mk_cOccurs o mk_Value_cVar) nonfresh_vals
    val receives = filter (is_Receive o snd) cactions
    val value_inequalities = map lblS (mk_cInequalities nonfresh_vals)
    val checksingles = filter (fn (_,a) => is_InSet a orelse is_NotInSet a) cactions
    val checkalls = filter (is_NotInAny o snd) cactions
    val updates = filter (fn (_,a) => is_Insert a orelse is_Delete a) cactions
    val fresh = filter (is_Fresh o snd) cactions
    val sends = filter (is_Send o snd) cactions
    val fresh_occurs = map (lblS o cSend o mk_cOccurs o mk_Value_cVar) fresh_vals
    val attack_signals = filter (is_Attack o snd) cactions
  in
    {transaction = #transaction tr,
     receive_actions = nonfresh_occurs@receives,
     checksingle_actions = value_inequalities@checksingles,
     checkall_actions = checkalls,
     fresh_actions = fresh,
     update_actions = updates,
     send_actions = sends@fresh_occurs,
     attack_actions = attack_signals}:cTransaction
  end

fun subst_apply_action (delta:(string * Trac_Term.cMsg) list) (lbl:prot_label,a:cAction) =
  let
    val apply = Trac_Term.subst_apply delta
  in
    case a of
      cReceive t => (lbl,cReceive (apply t))
    | cSend t => (lbl,cSend (apply t))
    | cInequality (x,y) => (lbl,cInequality (apply x, apply y))
    | cInSet (x,s) => (lbl,cInSet (apply x, apply s))
    | cNotInSet (x,s) => (lbl,cNotInSet (apply x, apply s))
    | cNotInAny (x,s) => (lbl,cNotInAny (apply x, s))
    | cInsert (x,s) => (lbl,cInsert (apply x, apply s))
    | cDelete (x,s) => (lbl,cDelete (apply x, apply s))
    | cNew x => (lbl,cNew x)
    | cAssertAttack => (lbl,cAssertAttack)
  end

fun subst_apply_actions delta =
  map (subst_apply_action delta)


type protocol = {
  name:string
 ,type_spec:type_spec
 ,set_spec:set_spec list
 ,function_spec:fun_spec option
 ,analysis_spec:anaT
 ,transaction_spec:(string option * transaction list) list
 ,fixed_point: (Trac_Term.cMsg list * (string * string list) list list *
                ((string * string list) list * (string * string list) list) list) option
}

exception TypeError

val fun_empty = {
                  public=[]
                 ,private=[]
                }:fun_spec
```

```
fun update_fun_public (fun_spec:fun_spec) public =
    ({public = public
     ,private = #private fun_spec
    }):fun_spec

fun update_fun_private (fun_spec:fun_spec) private =
    ({public = #public fun_spec
     ,private = private
    }):fun_spec


val empty={
            name=""
           ,type_spec=[]
           ,set_spec=[]
           ,function_spec=NONE
           ,analysis_spec=[]
           ,transaction_spec=[]
           ,fixed_point = NONE
          }:protocol

fun update_name (protocol_spec:protocol) name =
    ({name = name
     ,type_spec = #type_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_sets (protocol_spec:protocol) set_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,set_spec =
        if has_duplicates (op =) (map fst set_spec)
        then error "Multiple declarations of the same set family"
        else set_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_type_spec (protocol_spec:protocol) type_spec =
    ({name = #name protocol_spec
     ,type_spec =
        if has_duplicates (op =) (map fst type_spec)
        then error "Multiple declarations of the same enumeration type"
        else type_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_functions (protocol_spec:protocol) function_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = case function_spec of
          SOME fs =>
            if has_duplicates (op =) (map fst ((#public fs)@(#private fs)))
            then error "Multiple declarations of the same constant or function symbol"
```

```
            else SOME fs
        | NONE => NONE
     ,analysis_spec = #analysis_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_analysis (protocol_spec:protocol) analysis_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec =
        if has_duplicates (op =) (map (#1 o #1) analysis_spec)
        then error "Multiple analysis rules declared for the same function symbol"
        else if List.exists (has_duplicates (op =)) (map (#2 o #1) analysis_spec)
        then error "The heads of the analysis rules must be linear terms"
        else if let fun f ((_,xs),ts,ys) =
                          subset (op =) (ys@List.concat (map Trac_Term.fv_Msg ts), xs)
                in List.exists (not o f) analysis_spec end
        then error "Variables occurring in the body of an analysis rule should also occur in its head"
        else analysis_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_transactions (prot_name:string option) (protocol_spec:protocol) transaction_spec =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,transaction_spec = (prot_name,transaction_spec)::(#transaction_spec protocol_spec)
     ,fixed_point = #fixed_point protocol_spec
    }):protocol
fun update_fixed_point (protocol_spec:protocol) fixed_point =
    ({name = #name protocol_spec
     ,type_spec = #type_spec protocol_spec
     ,set_spec = #set_spec protocol_spec
     ,function_spec = #function_spec protocol_spec
     ,analysis_spec = #analysis_spec protocol_spec
     ,transaction_spec = #transaction_spec protocol_spec
     ,fixed_point = fixed_point
    }):protocol


end
```

end

3.4 Parser for Trac FP definitions (trac_fp_parser)

theory
  trac_fp_parser
  imports
    "trac_term"
begin

ML_file *"trac_parser/trac_fp.grm.sig"*
ML_file *"trac_parser/trac_fp.lex.sml"*
ML_file *"trac_parser/trac_fp.grm.sml"*

ML⟨

```
structure TracFpParser : sig
     val parse_file: string -> (Trac_Term.cMsg) list
       val parse_str: string -> (Trac_Term.cMsg) list
       (* val term_of_trac: Trac_Term.cMsg -> term *)
       val attack: Trac_Term.cMsg list -> bool
end =
struct

  open Trac_Term

  structure TracLrVals =
    TracLrValsFun(structure Token = LrParser.Token)

  structure TracLex =
    TracLexFun(structure Tokens = TracLrVals.Tokens)

  structure TracParser =
    Join(structure LrParser = LrParser
  structure ParserData = TracLrVals.ParserData
  structure Lex = TracLex)

  fun invoke lexstream =
      let fun print_error (s,i:(int * int * int),_) =
       TextIO.output(TextIO.stdOut,
       "Error, line .... " ^ (Int.toString (#1 i)) ^"."^(Int.toString (#2 i ))^ ", " ^ s ^ "\n")
       in TracParser.parse(0,lexstream,print_error,())
      end

 fun parse_fp lexer =  let
   val dummyEOF = TracLrVals.Tokens.EOF((0,0,0),(0,0,0))
    fun certify (m,t) = Trac_Term.certifyMsg t [] m
   fun loop lexer =
       let
    val _ = (TracLex.UserDeclarations.pos := (0,0,0);())
    val (res,lexer) = invoke lexer
    val (nextToken,lexer) = TracParser.Stream.get lexer
        in if TracParser.sameToken(nextToken,dummyEOF) then ((),res)
    else loop lexer
       end
       in  map certify (#2(loop lexer))
      end

 fun parse_file tracFile = let
      val infile = TextIO.openIn tracFile
      val lexer = TracParser.makeLexer  (fn _ => case ((TextIO.inputLine) infile) of
                                                  SOME s => s
                                                | NONE   => "")
     in
       parse_fp lexer
     end

 fun parse_str trac_fp_str = let
       val parsed = Unsynchronized.ref false
       fun input_string _  = if !parsed then "" else (parsed := true ;trac_fp_str)
      val lexer = TracParser.makeLexer input_string
     in
       parse_fp lexer
     end
  fun attack fp = List.exists (fn e => e = cAttack) fp

(*   fun term_of_trac (Trac_Term.cVar (n,t)) = @{const "cVar"}$(HOLogic.mk_tuple[HOLogic.mk_string n,
                                                                                HOLogic.mk_string t])
```

```
    | term_of_trac (Trac_Term.cConst n)   = @{const "cConst"}$HOLogic.mk_string n
    | term_of_trac (Trac_Term.cFun (n,l))   = @{const "cFun"}
                          $(HOLogic.mk_tuple[HOLogic.mk_string n, HOLogic.mk_list @{typ "cMsg"}
                             (map term_of_trac l)]) *)
end
⟩
```

end

3.5 Parser for the Trac Format (trac_protocol_parser)

theory
  trac_protocol_parser
  imports
    "trac_term"
begin

ML_file *"trac_parser/trac_protocol.grm.sig"*
ML_file *"trac_parser/trac_protocol.lex.sml"*
ML_file *"trac_parser/trac_protocol.grm.sml"*

ML⟨
```
structure TracProtocolParser : sig
     val parse_file: string -> TracProtocol.protocol
       val parse_str: string ->  TracProtocol.protocol
end =
struct

  structure TracLrVals =
    TracTransactionLrValsFun(structure Token = LrParser.Token)

  structure TracLex =
    TracTransactionLexFun(structure Tokens = TracLrVals.Tokens)

  structure TracParser =
    Join(structure LrParser = LrParser
  structure ParserData = TracLrVals.ParserData
  structure Lex = TracLex)

  fun invoke lexstream =
      let fun print_error (s,i:(int * int * int),_) =
       error("Error, line .... " ^ (Int.toString (#1 i)) ^"."^(Int.toString (#2 i ))^ ", " ^ s ^ "\n")
       in TracParser.parse(0,lexstream,print_error,())
      end

 fun parse_fp lexer =  let
   val dummyEOF = TracLrVals.Tokens.EOF((0,0,0),(0,0,0))
   fun loop lexer =
       let
    val _ = (TracLex.UserDeclarations.pos := (0,0,0);())
    val (res,lexer) = invoke lexer
    val (nextToken,lexer) = TracParser.Stream.get lexer
        in if TracParser.sameToken(nextToken,dummyEOF) then ((),res)
    else loop lexer
       end
       in  (#2(loop lexer))
      end

 fun parse_file tracFile =
     let
         val infile = TextIO.openIn tracFile
```

```
        val lexer = TracParser.makeLexer  (fn _ => case ((TextIO.inputLine) infile) of
                                                     SOME s => s
                                                   | NONE   => "")
    in
      parse_fp lexer
     handle LrParser.ParseError => TracProtocol.empty
    end

 fun parse_str str =
    let
         val parsed = Unsynchronized.ref false
         fun input_string _  = if !parsed then "" else (parsed := true ;str)
        val lexer = TracParser.makeLexer input_string
    in
      parse_fp lexer
     handle LrParser.ParseError => TracProtocol.empty
    end

end
⟩
```

end

3.6 Support for the Trac Format (trac)

theory
  "trac"
  imports
  trac_fp_parser
  trac_protocol_parser
keywords
      "trac" :: thy_decl
  and *"trac_import" :: thy_decl*
  and *"trac_trac" :: thy_decl*
  and *"trac_import_trac" :: thy_decl*
  and *"protocol_model_setup" :: thy_decl*
  and *"protocol_security_proof" :: thy_decl*
  and *"manual_protocol_model_setup" :: thy_decl*
  and *"manual_protocol_security_proof" :: thy_decl*
  and *"compute_fixpoint" :: thy_decl*
  and *"compute_SMP" :: thy_decl*
  and *"setup_protocol_model'" :: thy_decl*
  and *"protocol_security_proof'" :: thy_decl*
  and *"setup_protocol_checks" :: thy_decl*
begin

ML ⟨
```
(* Some of this is based on code from the following files distributed with Isabelle 2018:
    * HOL/Tools/value_command.ML
    * HOL/Code_Evaluation.thy
    * Pure.thy
*)

fun protocol_model_interpretation_defs name =
  let
    fun f s =
      (Binding.empty_atts:Attrib.binding, ((Binding.name s, NoSyn), name ^ "." ^ s))
  in
    (map f [
      "public", "arity", "Ana", "Γ", "Γ⇩v", "timpls_transformable_to", "intruder_synth_mod_timpls",
      "analyzed_closed_mod_timpls", "timpls_transformable_to'", "intruder_synth_mod_timpls'",
```

```
      "analyzed_closed_mod_timpls'", "admissible_transaction_terms", "admissible_transaction",
      "abs_substs_set", "abs_substs_fun", "in_trancl", "transaction_poschecks_comp",
      "transaction_negchecks_comp", "transaction_check_comp", "transaction_check",
      "transaction_check_pre", "transaction_check_post", "compute_fixpoint_fun'",
      "compute_fixpoint_fun", "attack_notin_fixpoint", "protocol_covered_by_fixpoint",
      "analyzed_fixpoint", "wellformed_protocol'", "wellformed_protocol", "wellformed_fixpoint",
      "wellformed_composable_protocols", "composable_protocols"
    ]):string Interpretation.defines
 end

fun protocol_model_interpretation_params name =
  let
    fun f s = name ^ "_" ^ s
  in
    map SOME  [f "arity", "λ_. 0", f "public", f "Ana", f "Γ", "0::nat", "1::nat"]
  end

fun declare_thm_attr attribute name print lthy =
  let
    val arg = [(Facts.named name, [[Token.make_string (attribute, Position.none)]])]
    val (_, lthy') = Specification.theorems_cmd "" [(Binding.empty_atts, arg)] [] print lthy
  in
    lthy'
  end

fun declare_def_attr attribute name = declare_thm_attr attribute (name ^ "_def")

val declare_code_eqn = declare_def_attr "code"

val declare_protocol_check = declare_def_attr "protocol_checks"

fun declare_protocol_checks print =
  declare_protocol_check "attack_notin_fixpoint" print #>
  declare_protocol_check "protocol_covered_by_fixpoint" print #>
  declare_protocol_check "analyzed_fixpoint" print #>
  declare_protocol_check "wellformed_protocol'" print #>
  declare_protocol_check "wellformed_protocol" print #>
  declare_protocol_check "wellformed_fixpoint" print #>
  declare_protocol_check "compute_fixpoint_fun" print

fun eval_define (name, raw_t) lthy =
  let
    val t = Code_Evaluation.dynamic_value_strict lthy (Syntax.read_term lthy raw_t)
    val arg = ((Binding.name name, NoSyn), ((Binding.name (name ^ "_def"),[]), t))
    val (_, lthy') = Local_Theory.define arg lthy
  in
    (t, lthy')
  end

fun eval_define_declare (name, raw_t) print =
  eval_define (name, raw_t) ##> declare_code_eqn name print

val _ = Outer_Syntax.local_theory' @{command_keyword "compute_fixpoint"}
        "evaluate and define protocol fixpoint"
        (Parse.name -- Parse.name >> (fn (protocol, fixpoint) => fn print =>
          snd o eval_define_declare (fixpoint, "compute_fixpoint_fun " ^ protocol) print));

val _ = Outer_Syntax.local_theory' @{command_keyword "compute_SMP"}
        "evaluate and define a finite representation of the sub-message patterns of a protocol"
        ((Scan.optional (keyword‹[› |-- Parse.name --| keyword‹]›) "no_optimizations") --
          Parse.name -- Parse.name >> (fn ((opt,protocol), smp) => fn print =>
          let
            val rmd = "List.remdups"
```

```
            val f = "Stateful_Strands.trms_list_sst"
            val g =
              "(λT. " ^ f ^ " T@map (pair' prot_fun.Pair) (Stateful_Strands.setops_list_sst T))"
            fun s trms =
              "(" ^ rmd ^ " (List.concat (List.map (" ^ trms ^
              " ∘ Labeled_Strands.unlabel ∘ transaction_strand) " ^ protocol ^ ")))"
            val opt1 = "remove_superfluous_terms Γ"
            val opt2 = "generalize_terms Γ is_Var"
            val gsmp_opt =
              "generalize_terms Γ (λt. is_Var t ∧ t ≠ TAtom AttackType ∧ " ^
              "t ≠ TAtom SetType ∧ t ≠ TAtom OccursSecType ∧ ¬is_Atom (the_Var t))"
            val smp_fun = "SMP0 Ana Γ"
            fun smp_fun' opts =
              "(λT. let T' = (" ^ rmd ^ " ∘ " ^ opts ^ " ∘ " ^ smp_fun ^
              ") T in List.map (λt. t · Typed_Model.var_rename (Typed_Model.max_var_set " ^
              "(Messages.fv_set (set (T@T'))))) T')"
            val cmd =
              if opt = "no_optimizations" then smp_fun ^ " " ^ s f
              else if opt = "optimized"
              then smp_fun' (opt1 ^ " ∘ " ^ opt2) ^ " " ^ s f
              else if opt = "GSMP"
              then smp_fun' (opt1 ^ " ∘ " ^ gsmp_opt) ^ " " ^ s g
              else error ("Invalid option: " ^ opt)
          in
            snd o eval_define_declare (smp, cmd) print
          end));

val _ = Outer_Syntax.local_theory' @{command_keyword "setup_protocol_checks"}
        "setup protocol checks"
        (Parse.name -- Parse.name >> (fn (protocol_model, protocol_name) => fn print =>
          let
            val a1 = "coverage_check_intro_lemmata"
            val a2 = "coverage_check_unfold_lemmata"
            val a3 = "coverage_check_unfold_protocol_lemma"
          in
            declare_protocol_checks print #>
            declare_thm_attr a1 (protocol_model ^ ".protocol_covered_by_fixpoint_intros") print #>
            declare_def_attr a2 (protocol_model ^ ".protocol_covered_by_fixpoint") print #>
            declare_def_attr a3 protocol_name print
          end
        ));

val _ =
  Outer_Syntax.local_theory_to_proof command_keyword⟨setup_protocol_model'⟩
    "prove interpretation of protocol model locale into global theory"
    (Parse.!!! (Parse.name -- Parse_Spec.locale_expression) >> (fn (prefix,expr) => fn lthy =>
    let
      fun f x y z = ([(x,(y,(Expression.Positional z,[])))],[])
      val (a,(b,c)) = nth (fst expr) 0
      val name = fst b
      val _ = case c of (Expression.Named [],[]) => () | _ => error "Invalid arguments"
      val pexpr = f a b (protocol_model_interpretation_params prefix)
      val pdefs = protocol_model_interpretation_defs name
    in
      if name = ""
      then error "No name given"
      else Interpretation.global_interpretation_cmd pexpr pdefs lthy
  end));

val _ =
  Outer_Syntax.local_theory_to_proof' command_keyword⟨protocol_security_proof'⟩
    "prove interpretation of secure protocol locale into global theory"
    (Parse.!!! (Parse.name -- Parse_Spec.locale_expression) >> (fn (prefix,expr) => fn print =>
```

```
    let
      fun f x y z = ([(x,(y,(Expression.Positional z,[])))],[])
      val (a,(b,c)) = nth (fst expr) 0
      val d = case c of (Expression.Positional ps,[]) => ps | _ => error "Invalid arguments"
      val pexpr = f a b (protocol_model_interpretation_params prefix@d)
    in
      declare_protocol_checks print #> Interpretation.global_interpretation_cmd pexpr []
    end
    ));
⟩
```

ML⟨

```
structure ml_isar_wrapper = struct
  fun define_constant_definition (constname, trm) lthy =
    let
      val arg = ((Binding.name constname, NoSyn), ((Binding.name (constname^"_def"),[]), trm))
      val ((_, (_ , thm)), lthy') = Local_Theory.define arg lthy
    in
      (thm, lthy')
    end

  fun define_constant_definition' (constname, trm) print lthy =
    let
      val arg = ((Binding.name constname, NoSyn), ((Binding.name (constname^"_def"),[]), trm))
      val ((_, (_ , thm)), lthy') = Local_Theory.define arg lthy
      val lthy'' = declare_code_eqn constname print lthy'
    in
      (thm, lthy'')
    end

  fun define_simple_abbrev (constname, trm) lthy =
    let
      val arg = ((Binding.name constname, NoSyn), trm)
      val ((_, _), lthy') = Local_Theory.abbrev Syntax.mode_default arg lthy
    in
      lthy'
    end

  fun define_simple_type_synonym (name, typedecl) lthy =
    let
      val (_, lthy') = Typedecl.abbrev_global (Binding.name name, [], NoSyn) typedecl lthy
    in
      lthy'
    end

  fun define_simple_datatype (dt_tyargs, dt_name) constructors =
    let
      val options = Plugin_Name.default_filter
      fun lift_c (tyargs, name) =  (((Binding.empty, Binding.name name), map (fn t => (Binding.empty, t))
tyargs), NoSyn)
      val c_spec = map lift_c constructors
      val datatyp = ((map (fn ty => (NONE, ty)) dt_tyargs, Binding.name dt_name), NoSyn)
      val dtspec =
        ((options,false),
         [(((datatyp, c_spec), (Binding.empty, Binding.empty, Binding.empty)), [])])
    in
      BNF_FP_Def_Sugar.co_datatypes BNF_Util.Least_FP BNF_LFP.construct_lfp dtspec
    end

   fun define_simple_primrec pname precs lthy =
     let
       val rec_eqs = map (fn (lhs,rhs) => (((Binding.empty,[]), HOLogic.mk_Trueprop (HOLogic.mk_eq (lhs,rhs))),[],
precs
```

```
    in
      snd (BNF_LFP_Rec_Sugar.primrec false [] [(Binding.name pname, NONE, NoSyn)] rec_eqs lthy)
    end

  fun define_simple_fun pname precs lthy =
    let
      val rec_eqs = map (fn (lhs,rhs) => (((Binding.empty,[]), HOLogic.mk_Trueprop (HOLogic.mk_eq (lhs,rhs))),[],
precs
    in
      Function_Fun.add_fun [(Binding.name pname, NONE, NoSyn)] rec_eqs  Function_Common.default_config
lthy
    end

  fun prove_simple name stmt tactic lthy =
    let
      val thm = Goal.prove lthy [] [] stmt (fn {context, ...} => tactic context)
                |> Goal.norm_result lthy
                |> Goal.check_finished lthy
    in
      lthy |>
      snd o  Local_Theory.note ((Binding.name name, []), [thm])
    end

   fun prove_state_simple method proof_state =
         Seq.the_result "error in proof state" ( (Proof.refine method proof_state))
             |> Proof.global_done_proof

end
```

ML

```
structure trac_definitorial_package = struct
  (* constant names *)
  open Trac_Utils
  val enum_constsN="enum_consts"
  val setsN="sets"
  val funN="fun"
  val atomN="atom"
  val arityN="arity"
  val publicN = "public"
  val gammaN = "Γ"
  val anaN = "Ana"
  val valN = "val"
  val timpliesN = "timplies"
  val occursN = "occurs"
  val enumN = "enum"
  val priv_fun_secN = "PrivFunSec"
  val secret_typeN = "SecretType"
  val enum_typeN = "EnumType"
  val other_pubconsts_typeN = "PubConstType"

  val types = [enum_typeN, secret_typeN]
  val special_funs = ["occurs", "zero", valN, priv_fun_secN]

  fun mk_listT T =  Type ("List.list", [T])
  val mk_setT = HOLogic.mk_setT
  val boolT = HOLogic.boolT
  val natT = HOLogic.natT
  val mk_tupleT =  HOLogic.mk_tupleT
  val mk_prodT = HOLogic.mk_prodT

  val mk_set = HOLogic.mk_set
```

```
val mk_list = HOLogic.mk_list
val mk_nat = HOLogic.mk_nat
val mk_eq = HOLogic.mk_eq
val mk_Trueprop = HOLogic.mk_Trueprop
val mk_tuple = HOLogic.mk_tuple
val mk_prod = HOLogic.mk_prod

fun mkN (a,b) = a^"_"^b

val info = Output.information

fun rm_special_funs sel l = list_minus (list_rm_pair sel) l special_funs

fun is_priv_fun (trac:TracProtocol.protocol) f = let
  val funs = #private (Option.valOf (#function_spec trac))
  in
    (* not (List.find (fn g => fst g = f) funs = NONE) *)
    List.exists (fn (g,n) => f = g andalso n <> "0") funs
  end

fun full_name name lthy =
  Local_Theory.full_name lthy (Binding.name name)

fun full_name' n (trac:TracProtocol.protocol) lthy = full_name (mkN (#name trac, n)) lthy

fun mk_prot_type name targs (trac:TracProtocol.protocol) lthy =
  Term.Type (full_name' name trac lthy, targs)

val enum_constsT = mk_prot_type enum_constsN []

fun mk_enum_const a trac lthy =
  Term.Const (full_name' enum_constsN trac lthy ^ "." ^ a, enum_constsT trac lthy)

val databaseT = mk_prot_type setsN []

val funT = mk_prot_type funN []

val atomT = mk_prot_type atomN []

fun messageT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Transactions.prot_term", [funT trac lthy, atomT trac lthy, databaseT trac lthy])

fun message_funT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Transactions.prot_fun", [funT trac lthy, atomT trac lthy, databaseT trac lthy])

fun message_varT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Transactions.prot_var", [funT trac lthy, atomT trac lthy, databaseT trac lthy])

fun message_term_typeT (trc:TracProtocol.protocol) lthy =
  Term.Type ("Transactions.prot_term_type", [funT trc lthy, atomT trc lthy, databaseT trc lthy])

fun message_atomT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Transactions.prot_atom", [atomT trac lthy])

fun messageT' varT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Term.term", [message_funT trac lthy, varT])

fun message_listT (trac:TracProtocol.protocol) lthy =
  mk_listT (messageT trac lthy)

fun message_listT' varT (trac:TracProtocol.protocol) lthy =
  mk_listT (messageT' varT trac lthy)
```

```
fun absT (trac:TracProtocol.protocol) lthy =
  mk_setT (databaseT trac lthy)

fun abssT (trac:TracProtocol.protocol) lthy =
  mk_setT (absT trac lthy)

val poscheckvariantT =
  Term.Type ("Strands_and_Constraints.poscheckvariant", [])

val strand_labelT =
  Term.Type ("Labeled_Strands.strand_label", [natT])

fun strand_stepT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Stateful_Strands.stateful_strand_step",
             [message_funT trac lthy, message_varT trac lthy])

fun labeled_strand_stepT (trac:TracProtocol.protocol) lthy =
  mk_prodT (strand_labelT, strand_stepT trac lthy)

fun prot_strandT (trac:TracProtocol.protocol) lthy =
  mk_listT (labeled_strand_stepT trac lthy)

fun prot_transactionT (trac:TracProtocol.protocol) lthy =
  Term.Type ("Transactions.prot_transaction",
             [funT trac lthy, atomT trac lthy, databaseT trac lthy, natT])

val mk_star_label =
  Term.Const ("Labeled_Strands.strand_label.LabelS", strand_labelT)

fun mk_prot_label (lbl:int) =
  Term.Const ("Labeled_Strands.strand_label.LabelN", natT --> strand_labelT) $
    mk_nat lbl

fun mk_labeled_step (label:term) (step:term) =
  mk_prod (label, step)

fun mk_Send_step (trac:TracProtocol.protocol) lthy (label:term) (msg:term) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Send",
                 messageT trac lthy --> strand_stepT trac lthy) $ msg)

fun mk_Receive_step (trac:TracProtocol.protocol) lthy (label:term) (msg:term) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Receive",
                 messageT trac lthy --> strand_stepT trac lthy) $ msg)

fun mk_InSet_step (trac:TracProtocol.protocol) lthy (label:term) (elem:term) (set:term) =
  let
    val psT = [poscheckvariantT, messageT trac lthy, messageT trac lthy]
  in
    mk_labeled_step label
      (Term.Const ("Stateful_Strands.stateful_strand_step.InSet",
                   psT ---> strand_stepT trac lthy) $
       Term.Const ("Strands_and_Constraints.poscheckvariant.Check", poscheckvariantT) $
       elem $ set)
  end

fun mk_NotInSet_step (trac:TracProtocol.protocol) lthy (label:term) (elem:term) (set:term) =
  let
    val varT = message_varT trac lthy
    val trm_prodT = mk_prodT (messageT trac lthy, messageT trac lthy)
    val psT = [mk_listT varT, mk_listT trm_prodT, mk_listT trm_prodT]
  in
```

```
    mk_labeled_step label
      (Term.Const ("Stateful_Strands.stateful_strand_step.NegChecks",
                   psT ---> strand_stepT trac lthy) $
       mk_list varT [] $
       mk_list trm_prodT [] $
       mk_list trm_prodT [mk_prod (elem,set)])
  end

fun mk_Inequality_step (trac:TracProtocol.protocol) lthy (label:term) (t1:term) (t2:term) =
  let
    val varT = message_varT trac lthy
    val trm_prodT = mk_prodT (messageT trac lthy, messageT trac lthy)
    val psT = [mk_listT varT, mk_listT trm_prodT, mk_listT trm_prodT]
  in
    mk_labeled_step label
      (Term.Const ("Stateful_Strands.stateful_strand_step.NegChecks",
                   psT ---> strand_stepT trac lthy) $
       mk_list varT [] $
       mk_list trm_prodT [mk_prod (t1,t2)] $
       mk_list trm_prodT [])
  end

fun mk_Insert_step (trac:TracProtocol.protocol) lthy (label:term) (elem:term) (set:term) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Insert",
                 [messageT trac lthy, messageT trac lthy] ---> strand_stepT trac lthy) $
     elem $ set)

fun mk_Delete_step (trac:TracProtocol.protocol) lthy (label:term) (elem:term) (set:term) =
  mk_labeled_step label
    (Term.Const ("Stateful_Strands.stateful_strand_step.Delete",
                 [messageT trac lthy, messageT trac lthy] ---> strand_stepT trac lthy) $
     elem $ set)

fun mk_Transaction (trac:TracProtocol.protocol) lthy S1 S2 S3 S4 S5 S6 =
  let
    val varT = message_varT trac lthy
    val msgT = messageT trac lthy
    val var_listT = mk_listT varT
    val msg_listT = mk_listT msgT
    val trT = prot_transactionT trac lthy
    (* val decl_elemT = mk_prodT (varT, mk_listT msgT)
    val declT = mk_listT decl_elemT *)
    val stepT = labeled_strand_stepT trac lthy
    val strandT = prot_strandT trac lthy
    val strandsT = mk_listT strandT
    val paramsT = [(* declT,  *)var_listT, strandT, strandT, strandT, strandT, strandT]
  in
    Term.Const ("Transactions.prot_transaction.Transaction", paramsT ---> trT) $
    (* mk_list decl_elemT [] $ *)
    (if null S4 then mk_list varT []
     else (Term.Const (@{const_name "map"}, [msgT --> varT, msg_listT] ---> var_listT) $
           Term.Const (@{const_name "the_Var"}, msgT --> varT) $
           mk_list msgT S4)) $
    mk_list stepT S1 $
    mk_list stepT [] $
    (if null S3 then mk_list stepT S2
     else (Term.Const (@{const_name "append"}, [strandT,strandT] ---> strandT) $
           mk_list stepT S2 $
          (Term.Const (@{const_name "concat"}, strandsT --> strandT) $ mk_list strandT S3))) $
    mk_list stepT S5 $
    mk_list stepT S6
  end
```

```
fun get_funs (trac:TracProtocol.protocol) =
    let
      fun append_sec fs = fs@[(priv_fun_secN, "0")]
      val filter_funs = filter (fn (_,n) => n <> "0")
      val filter_consts = filter (fn (_,n) => n = "0")
      fun inc_ar (s,n) = (s, Int.toString (1+Option.valOf (Int.fromString n)))
    in
      case (#function_spec trac) of
           NONE => ([],[],[])
         | SOME ({public=pub, private=priv}) =>
           let
             val pub_symbols = rm_special_funs fst (pub@map inc_ar (filter_funs priv))
             val pub_funs = filter_funs pub_symbols
             val pub_consts = filter_consts pub_symbols
             val priv_consts = append_sec (rm_special_funs fst (filter_consts priv))
           in
             (pub_funs, pub_consts, priv_consts)
           end
    end

fun get_set_spec (trac:TracProtocol.protocol) =
  mk_unique (map (fn (s,n) => (s,Option.valOf (Int.fromString n))) (#set_spec trac))

fun set_arity (trac:TracProtocol.protocol) s =
  case List.find (fn x => fst x = s) (get_set_spec trac) of
    SOME (_,n) => SOME n
  | NONE => NONE

fun get_enums (trac:TracProtocol.protocol) =
  mk_unique (TracProtocol.extract_Consts (#type_spec trac))

fun flatten_type_spec (trac:TracProtocol.protocol) =
  let
    fun find_type taus tau =
      case List.find (fn x => fst x = tau) taus of
        SOME x => snd x
      | NONE => error ("Type " ^ tau ^ " has not been declared")
    fun step taus (s,e) =
      case e of
        TracProtocol.Union ts =>
          let
            val es = map (find_type taus) ts
            fun f es' = mk_unique (List.concat (map TracProtocol.the_Consts es'))
          in
            if List.all TracProtocol.is_Consts es
            then (s,TracProtocol.Consts (f es))
            else (s,TracProtocol.Union ts)
          end
      | c => (s,c)
    fun loop taus =
      let
        val taus' = map (step taus) taus
      in
        if taus = taus'
        then taus
        else loop taus'
      end
    val flat_type_spec =
      let
        val x = loop (#type_spec trac)
        val errpre = "Couldn't flatten the enumeration types: "
      in
```

```
        if List.all (fn (_,e) => TracProtocol.is_Consts e) x
        then
          let
            val y = map (fn (s,e) => (s,TracProtocol.the_Consts e)) x
          in
            if List.all (not o List.null o snd) y
            then y
            else error (errpre ^ "does every type have at least one value?")
          end
        else error (errpre ^ "have all types been declared?")
      end
  in
    flat_type_spec
  end

fun is_attack_transaction (tr:TracProtocol.cTransaction) =
  not (null (#attack_actions tr))

fun get_transaction_name (tr:TracProtocol.cTransaction) =
  #1 (#transaction tr)

fun get_fresh_value_variables (tr:TracProtocol.cTransaction) =
  map_filter (TracProtocol.maybe_the_Fresh o snd) (#fresh_actions tr)

fun get_nonfresh_value_variables (tr:TracProtocol.cTransaction) =
  map fst (filter (fn x => snd x = "value") (#2 (#transaction tr)))

fun get_value_variables (tr:TracProtocol.cTransaction) =
  get_nonfresh_value_variables tr@get_fresh_value_variables tr

fun get_enum_variables (tr:TracProtocol.cTransaction) =
  mk_unique (filter (fn x => snd x <> "value") (#2 (#transaction tr)))

fun get_variable_restrictions (tr:TracProtocol.cTransaction) =
  let
    val enum_vars = get_enum_variables tr
    val value_vars = get_value_variables tr
    fun enum_member x = List.exists (fn y => x = fst y)
    fun value_member x = List.exists (fn y => x = y)
    fun aux [] = ([],[])
      | aux ((a,b)::rs) =
          if enum_member a enum_vars andalso enum_member b enum_vars
          then let val (es,vs) = aux rs in ((a,b)::es,vs) end
          else if value_member a value_vars andalso value_member b value_vars
          then let val (es,vs) = aux rs in (es,(a,b)::vs) end
          else error ("Ill-formed or ill-typed variable restriction: " ^ a ^ " != " ^ b)
  in
    aux (#3 (#transaction tr))
  end

fun conv_enum_consts trac (t:Trac_Term.cMsg) =
  let
    open Trac_Term
    val enums = get_enums trac
    fun aux (cFun (f,ts)) =
          if List.exists (fn x => x = f) enums
          then if null ts
               then cEnum f
               else error ("Enum constant " ^ f ^ " should not have a parameter list")
          else
            cFun (f,map aux ts)
      | aux (cConst c) =
          if List.exists (fn x => x = c) enums
```

```
          then cEnum c
          else cConst c
      | aux (cSet (s,ts)) = cSet (s,map aux ts)
      | aux (cOccursFact bs) = cOccursFact (aux bs)
      | aux t = t
  in
    aux t
  end

fun val_to_abs_list vs =
  let
    open Trac_Term
    fun aux t = case t of cEnum b => b | _ => error "Invalid val parameter list"
  in
    case vs of
      [] => []
    | (cConst "0"::ts) => val_to_abs_list ts
    | (cFun (s,ps)::ts) => (s, map aux ps)::val_to_abs_list ts
    | (cSet (s,ps)::ts) => (s, map aux ps)::val_to_abs_list ts
    | _ => error "Invalid val parameter list"
  end

fun val_to_abs (t:Trac_Term.cMsg) =
  let
    open Trac_Term
    fun aux t = case t of cEnum b => b | _ => error "Invalid val parameter list"

    fun val_to_abs_list [] = []
    | val_to_abs_list (cConst "0"::ts) = val_to_abs_list ts
    | val_to_abs_list (cFun (s,ps)::ts) = (s, map aux ps)::val_to_abs_list ts
    | val_to_abs_list (cSet (s,ps)::ts) = (s, map aux ps)::val_to_abs_list ts
    | val_to_abs_list _ = error "Invalid val parameter list"
  in
    case t of
      cFun (f,ts) =>
        if f = valN
        then cAbs (val_to_abs_list ts)
        else cFun (f,map val_to_abs ts)
    | cSet (s,ts) =>
        cSet (s,map val_to_abs ts)
    | cOccursFact bs =>
        cOccursFact (val_to_abs bs)
    | t => t
  end

fun occurs_enc t =
  let
    open Trac_Term
    fun aux [cVar x] = cVar x
      | aux [cAbs bs] = cAbs bs
      | aux _ = error "Invalid occurs parameter list"
    fun enc (cFun (f,ts)) = (
          if f = occursN
          then cOccursFact (aux ts)
          else cFun (f,map enc ts))
      | enc (cSet (s,ts)) =
          cSet (s,map enc ts)
      | enc (cOccursFact bs) =
          cOccursFact (enc bs)
      | enc t = t
  in
    enc t
  end
```

```
fun priv_fun_enc trac (Trac_Term.cFun (f,ts)) = (
      if is_priv_fun trac f andalso
         (case ts of Trac_Term.cPrivFunSec::_ => false | _ => true)
      then Trac_Term.cFun (f,Trac_Term.cPrivFunSec::map (priv_fun_enc trac) ts)
      else Trac_Term.cFun (f,map (priv_fun_enc trac) ts))
  | priv_fun_enc _ t = t

fun transform_cMsg trac =
  priv_fun_enc trac o occurs_enc o val_to_abs o conv_enum_consts trac

fun check_no_vars_and_consts (fp:Trac_Term.cMsg list) =
  let
    open Trac_Term
    fun aux (cVar _) = false
      | aux (cConst _) = false
      | aux (cFun (_,ts)) = List.all aux ts
      | aux (cSet (_,ts)) = List.all aux ts
      | aux (cOccursFact bs) = aux bs
      | aux _ = true
  in
    if List.all aux fp
    then fp
    else error "There shouldn't be any cVars and cConsts at this point in the fixpoint translation"
  end

fun split_fp (fp:Trac_Term.cMsg list) =
  let
    open Trac_Term
    fun fa t = case t of cFun (s,_) => s <> timpliesN | _ => true
    fun fb (t,ts) = case t of cOccursFact (cAbs bs) => bs::ts | _ => ts
    fun fc (cFun (s, [cAbs bs, cAbs cs]),ts) =
        if s = timpliesN
        then (bs,cs)::ts
        else ts
      | fc (_,ts) = ts

    val eq = eq_set (fn ((s,xs),(t,ys)) => s = t andalso eq_set (op =) (xs,ys))
    fun eq_pairs ((a,b),(c,d)) = eq (a,c) andalso eq (b,d)

    val timplies_trancl =
      let
        fun trans_step ts =
          let
            fun aux (s,t) = map (fn (_,u) => (s,u)) (filter (fn (v,_) => eq (t,v)) ts)
          in
            distinct eq_pairs (filter (not o eq) (ts@List.concat (map aux ts)))
          end
        fun loop ts =
          let
            val ts' = trans_step ts
          in
            if eq_set eq_pairs (ts,ts')
            then ts
            else loop ts'
          end
      in
        loop
      end

    val ti = List.foldl fc [] fp
  in
    (filter fa fp, distinct eq (List.foldl fb [] fp@map snd ti), timplies_trancl ti)
```

```
  end

fun mk_enum_substs trac (vars:(string * Trac_Term.VarType) list) =
  let
    open Trac_Term
    val flat_type_spec = flatten_type_spec trac
    val deltas =
      let
        fun f (s,EnumType tau) = (
            case List.find (fn x => fst x = tau) flat_type_spec of
              SOME x => map (fn c => (s,c)) (snd x)
            | NONE => error ("Type " ^ tau ^ " was not found in the type specification"))
          | f (s,_) = error ("Variable " ^ s ^ " is not of enum type")
      in
        list_product (map f vars)
      end
  in
    map (fn d => map (fn (x,t) => (x,cEnum t)) d) deltas
  end

fun ground_enum_variables trac (fp:Trac_Term.cMsg list) =
  let
    open Trac_Term
    fun do_grounding t = map (fn d => subst_apply d t) (mk_enum_substs trac (fv_cMsg t))
  in
    List.concat (map do_grounding fp)
  end

fun transform_fp trac (fp:Trac_Term.cMsg list) =
  fp |> ground_enum_variables trac
     |> map (transform_cMsg trac)
     |> check_no_vars_and_consts
     |> split_fp

fun database_to_hol (db:string * Trac_Term.cMsg list) (trac:TracProtocol.protocol) lthy =
  let
    open Trac_Term
    val errmsg = "Invalid database parameter"
    fun mkN' n = mkN (#name trac, n)
    val s_prefix = full_name (mkN' setsN) lthy ^ "."
    val e_prefix = full_name (mkN' enum_constsN) lthy ^ "."
    val (s,es) = db
    val tau = enum_constsT trac lthy
    val databaseT = databaseT trac lthy
    val a = Term.Const (s_prefix ^ s, map (fn _ => tau) es ---> databaseT)
    fun param_to_hol (cVar (x,EnumType _)) = Term.Free (x, tau)
      | param_to_hol (cVar (x,Untyped)) = Term.Free (x, tau)
      | param_to_hol (cEnum e) = Term.Const (e_prefix ^ e, tau)
      | param_to_hol (cConst c) = error (errmsg ^ ": cConst " ^ c)
      | param_to_hol (cVar (x,ValueType)) = error (errmsg ^ ": cVar (" ^ x ^ ",ValueType)")
      | param_to_hol _ = error errmsg
  in
    fold (fn e => fn b => b $ param_to_hol e) es a
  end

fun abs_to_hol (bs:(string * string list) list) (trac:TracProtocol.protocol) lthy =
  let
    val databaseT = databaseT trac lthy
    fun db_params_to_cEnum (a,cs) = (a, map Trac_Term.cEnum cs)
  in
    mk_set databaseT (map (fn db => database_to_hol (db_params_to_cEnum db) trac lthy) bs)
  end
```

```
fun cMsg_to_hol (t:Trac_Term.cMsg) lbl varT var_map free_enum_var trac lthy =
  let
    open Trac_Term
    val tT = messageT' varT trac lthy
    val fT = message_funT trac lthy
    val enum_constsT = enum_constsT trac lthy
    val tsT = message_listT' varT trac lthy
    val VarT = varT --> tT
    val FunT = [fT, tsT] ---> tT
    val absT = absT trac lthy
    val databaseT = databaseT trac lthy
    val AbsT = absT --> fT
    val funT = funT trac lthy
    val FuT = funT --> fT
    val SetT = databaseT --> fT
    val enumT = enum_constsT --> funT
    val VarC = Term.Const (@{const_name "Var"}, VarT)
    val FunC = Term.Const (@{const_name "Fun"}, FunT)
    val NilC = Term.Const (@{const_name "Nil"}, tsT)
    val prot_label = mk_nat lbl
    fun full_name'' n = full_name' n trac lthy
    fun mk_enum_const' a = mk_enum_const a trac lthy
    fun mk_prot_fun_trm f tau = Term.Const ("Transactions.prot_fun." ^ f, tau)
    fun mk_enum_trm etrm =
          mk_prot_fun_trm "Fu" FuT $ (Term.Const (full_name'' funN ^ "." ^ enumN, enumT) $ etrm)
    fun mk_Fu_trm f =
          mk_prot_fun_trm "Fu" FuT $ Term.Const (full_name'' funN ^ "." ^ f, funT)
    fun c_to_h s = cMsg_to_hol s lbl varT var_map free_enum_var trac lthy
    fun c_list_to_h ts = mk_list tT (map c_to_h ts)
  in
    case t of
      cVar x =>
        if free_enum_var x
        then FunC $ mk_enum_trm (Term.Free (fst x, enum_constsT)) $ NilC
        else VarC $ var_map x
    | cConst f =>
        FunC $
        mk_Fu_trm f $
        NilC
    | cFun (f,ts) =>
        FunC $
        mk_Fu_trm f $
        c_list_to_h ts
    | cSet (s,ts) =>
        FunC $
        (mk_prot_fun_trm "Set" SetT $ database_to_hol (s,ts) trac lthy) $
        NilC
    | cAttack =>
        FunC $
        (mk_prot_fun_trm "Attack" (natT --> fT) $ prot_label) $
        NilC
    | cAbs bs =>
        FunC $
        (mk_prot_fun_trm "Abs" AbsT $ abs_to_hol bs trac lthy) $
        NilC
    | cOccursFact bs =>
        FunC $
        mk_prot_fun_trm "OccursFact" fT $
        mk_list tT [
          FunC $ mk_prot_fun_trm "OccursSec" fT $ NilC,
          c_to_h bs]
    | cPrivFunSec =>
        FunC $
```

```
        mk_Fu_trm priv_fun_secN $
        NilC
    | cEnum a =>
        FunC $
        mk_enum_trm (mk_enum_const' a) $
        NilC
end

fun ground_cMsg_to_hol t lbl trac lthy =
  cMsg_to_hol t lbl (message_varT trac lthy) (fn _ => error "Term not ground")
              (fn _ => false) trac lthy

fun ana_cMsg_to_hol inc_vars t (ana_var_map:string list) =
  let
    open Trac_Term
    fun var_map (x,Untyped) = (
          case list_find (fn y => x = y) ana_var_map of
            SOME (_,n) => if inc_vars then mk_nat (1+n) else mk_nat n
          | NONE => error ("Analysis variable " ^ x ^ " not found"))
      | var_map _ = error "Analysis variables must be untyped"
    val lbl = 0 (* There's no constants in analysis messages requiring labels anyway *)
  in
    cMsg_to_hol t lbl natT var_map (fn _ => false)
  end

fun transaction_cMsg_to_hol t lbl (transaction_var_map:string list) trac lthy =
  let
    open Trac_Term
    val varT = message_varT trac lthy
    val atomT = message_atomT trac lthy
    val term_typeT = message_term_typeT trac lthy
    fun TAtom_Value_var n =
      let
        val a = Term.Const (@{const_name "Var"}, atomT --> term_typeT) $
                Term.Const ("Transactions.prot_atom.Value", atomT)
      in
        HOLogic.mk_prod (a, mk_nat n)
      end

    fun var_map_err_prefix x =
      "Transaction variable " ^ x ^ " should be value typed but is actually "

    fun var_map (x,ValueType) = (
          case list_find (fn y => x = y) transaction_var_map of
            SOME (_,n) => TAtom_Value_var n
          | NONE => error ("Transaction variable " ^ x ^ " not found"))
      | var_map (x,EnumType e) = error (var_map_err_prefix x ^ "of enum type " ^ e)
      | var_map (x,Untyped) = error (var_map_err_prefix x ^ "untyped")
  in
    cMsg_to_hol t lbl varT var_map (fn (_,t) => case t of EnumType _ => true | _ => false)
                trac lthy
  end

fun fp_triple_to_hol (fp,occ,ti) trac lthy =
  let
    val prot_label = 0
    val tau_abs = absT trac lthy
    val tau_fp_elem = messageT trac lthy
    val tau_occ_elem = tau_abs
    val tau_ti_elem = mk_prodT (tau_abs, tau_abs)
    fun a_to_h bs = abs_to_hol bs trac lthy
    fun c_to_h t = ground_cMsg_to_hol t prot_label trac lthy
    val fp' = mk_list tau_fp_elem (map c_to_h fp)
```

```
    val occ' = mk_list tau_occ_elem (map a_to_h occ)
    val ti' = mk_list tau_ti_elem (map (mk_prod o map_prod a_to_h) ti)
  in
    mk_tuple [fp', occ', ti']
  end

fun abstract_over_enum_vars enum_vars enum_ineqs trm flat_type_spec trac lthy =
  let
    val enum_constsT = enum_constsT trac lthy
    fun enumlistelemT n = mk_tupleT (replicate n enum_constsT)
    fun enumlistT n = mk_listT (enumlistelemT n)
    fun mk_enum_const' a = mk_enum_const a trac lthy

    fun absfreeprod xs trm =
      let
        val tau = enum_constsT
        val tau_out = Term.fastype_of trm
        fun absfree' x = absfree (x,enum_constsT)
        fun aux _ [] = trm
          | aux _ [x] = absfree' x trm
          | aux len (x::y::xs) =
              Term.Const (@{const_name "case_prod"},
                     [[tau,mk_tupleT (replicate (len-1) tau)] ---> tau_out,
                      mk_tupleT (replicate len tau)] ---> tau_out) $
              absfree' x (aux (len-1) (y::xs))
      in
        aux (length xs) xs
      end

    fun mk_enum_neq (a,b) = (HOLogic.mk_not o HOLogic.mk_eq)
      (Term.Free (a, enum_constsT), Term.Free (b, enum_constsT))

    fun mk_enum_neqs_list [] = Term.Const (@{const_name "True"}, HOLogic.boolT)
      | mk_enum_neqs_list [x] = mk_enum_neq x
      | mk_enum_neqs_list (x::y::xs) = HOLogic.mk_conj (mk_enum_neq x, mk_enum_neqs_list (y::xs))

    val enum_types =
      let
        fun aux t =
          if t = ""
          then get_enums trac
          else case List.find (fn (s,_) => t = s) flat_type_spec of
              SOME (_,cs) => cs
            | NONE => error ("Not an enum type: " ^ t ^ "?")
      in
        map (aux o snd) enum_vars
      end

    val enumlist_product =
      let
        fun mk_enumlist ns = mk_list enum_constsT (map mk_enum_const' ns)

        fun aux _ [] = mk_enumlist []
          | aux _ [ns] = mk_enumlist ns
          | aux len (ns::ms::elists) =
              Term.Const ("List.product", [enumlistT 1, enumlistT (len-1)] ---> enumlistT len) $
              mk_enumlist ns $ aux (len-1) (ms::elists)
      in
        aux (length enum_types) enum_types
      end

    val absfp = absfreeprod (map fst enum_vars) trm
    val eptrm = enumlist_product
```

```
    val typof = Term.fastype_of
    val evseT = enumlistelemT (length enum_vars)
    val evslT = enumlistT (length enum_vars)
    val eneqs = absfreeprod (map fst enum_vars) (mk_enum_neqs_list enum_ineqs)
  in
    if null enum_vars
    then mk_list (typof trm) [trm]
    else if null enum_ineqs
    then Term.Const(@{const_name "map"},
                    [typof absfp, typof eptrm] ---> mk_listT (typof trm)) $
         absfp $ eptrm
    else Term.Const(@{const_name "map"},
                    [typof absfp, typof eptrm] ---> mk_listT (typof trm)) $
         absfp $ (Term.Const(@{const_name "filter"},
                             [evseT --> HOLogic.boolT, evslT] ---> evslT) $
                  eneqs $ eptrm)
  end

fun mk_type_of_name lthy pname name ty_args
    = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, name))), ty_args)

fun mk_mt_list t = Term.Const (@{const_name "Nil"}, mk_listT t)

fun name_of_typ (Type (s, _)) = s
  | name_of_typ (TFree _)     = error "name_of_type: unexpected TFree"
  | name_of_typ (TVar _ )     = error "name_of_type: unexpected TVAR"

fun prove_UNIV name typ elems thmsN lthy =
  let
    val rhs = mk_set typ elems
    val lhs = Const("Set.UNIV",mk_setT typ)
    val stmt = mk_Trueprop (mk_eq (lhs,rhs))
    val fq_tname = name_of_typ typ

    fun inst_and_prove_enum thy =
      let
        val _ = writeln("Inst enum: "^name)
        val lthy = Class.instantiation ([fq_tname], [], @{sort enum}) thy
        val enum_eq = Const("Pure.eq",mk_listT typ --> mk_listT typ --> propT)
                        $Const(@{const_name "enum_class.enum"},mk_listT typ)
                        $(mk_list typ elems)

        val ((_, (_, enum_def')), lthy) = Specification.definition NONE [] []
                                            ((Binding.name ("enum_"^name),[]), enum_eq) lthy
        val ctxt_thy = Proof_Context.init_global (Proof_Context.theory_of lthy)
        val enum_def = singleton (Proof_Context.export lthy ctxt_thy) enum_def'

        val enum_all_eq = Const("Pure.eq", boolT --> boolT --> propT)
                        $(Const(@{const_name "enum_class.enum_all"},(typ --> boolT) --> boolT)
                                  $Free("P",typ --> boolT))
                        $(Const(@{const_name "list_all"},(typ --> boolT) --> (mk_listT typ) --> boolT)
                               $Free("P",typ --> boolT)$(mk_list typ elems))
        val ((_, (_, enum_all_def')), lthy) = Specification.definition NONE [] []
                                            ((Binding.name ("enum_all_"^name),[]), enum_all_eq) lthy
        val ctxt_thy = Proof_Context.init_global (Proof_Context.theory_of lthy)
        val enum_all_def = singleton (Proof_Context.export lthy ctxt_thy) enum_all_def'

        val enum_ex_eq = Const("Pure.eq", boolT --> boolT --> propT)
                        $(Const(@{const_name "enum_class.enum_ex"},(typ --> boolT) --> boolT)
                                  $Free("P",typ --> boolT))
                        $(Const(@{const_name "list_ex"},(typ --> boolT) --> (mk_listT typ) --> boolT)
                               $Free("P",typ --> boolT)$(mk_list typ elems))
        val ((_, (_, enum_ex_def')), lthy) = Specification.definition NONE [] []
```

```
                                              ((Binding.name ("enum_ex_"^name),[]), enum_ex_eq) lthy
        val ctxt_thy = Proof_Context.init_global (Proof_Context.theory_of lthy)
        val enum_ex_def = singleton (Proof_Context.export lthy ctxt_thy) enum_ex_def'
      in
        Class.prove_instantiation_exit (fn ctxt =>
          (Class.intro_classes_tac ctxt [])  THEN
             ALLGOALS (simp_tac (ctxt addsimps  [Proof_Context.get_thm ctxt (name^"_UNIV"),
                                                 enum_def, enum_all_def, enum_ex_def]) )
          )lthy
      end
    fun inst_and_prove_finite thy =
      let
        val lthy = Class.instantiation ([fq_tname], [], @{sort finite}) thy
      in
        Class.prove_instantiation_exit (fn ctxt =>
          (Class.intro_classes_tac ctxt []) THEN
           (simp_tac (ctxt addsimps[Proof_Context.get_thm ctxt (name^"_UNIV")])) 1) lthy
      end
  in
    lthy
    |> ml_isar_wrapper.prove_simple (name^"_UNIV") stmt
       (fn c =>     (safe_tac c)
               THEN (ALLGOALS(simp_tac c))
               THEN (ALLGOALS(Metis_Tactic.metis_tac ["full_types"]
                              "combs"  c
                              (map (Proof_Context.get_thm c) thmsN)))
       )
    |> Local_Theory.raw_theory inst_and_prove_finite
    |> Local_Theory.raw_theory inst_and_prove_enum
  end

fun def_types (trac:TracProtocol.protocol) lthy =
  let
    val pname = #name trac
    val defname = mkN(pname, enum_constsN)
    val _ = info("  Defining "^defname)
    val tnames = get_enums trac
    val types = map (fn x => ([],x)) tnames
  in
    ([defname], ml_isar_wrapper.define_simple_datatype ([], defname) types lthy)
  end

fun def_sets (trac:TracProtocol.protocol) lthy =
  let
    val pname = #name trac
    val defname = mkN(pname, setsN)
    val _ = info ("  Defining "^defname)

    val sspec = get_set_spec trac
    val tfqn = Local_Theory.full_name lthy (Binding.name (mkN(pname, enum_constsN)))
    val ttyp = Type(tfqn, [])
    val types = map (fn (x,n) => (replicate n ttyp,x)) sspec
  in
    lthy
    |> ml_isar_wrapper.define_simple_datatype ([], defname) types
  end

fun def_funs (trac:TracProtocol.protocol) lthy =
  let
    val pname = #name trac
    val (pub_f, pub_c, priv) = get_funs trac
    val pub = pub_f@pub_c
```

```
fun def_atom lthy =
  let
    val def_atomname = mkN(pname, atomN)
    val types =
      if null pub_c
      then types
      else types@[other_pubconsts_typeN]
    fun define_atom_dt lthy =
      let
        val _ = info("  Defining "^def_atomname)
      in
        lthy
        |> ml_isar_wrapper.define_simple_datatype ([], def_atomname) (map (fn x => ([],x)) types)
      end
    fun prove_UNIV_atom lthy =
      let
        val _ = info ("    Proving "^def_atomname^"_UNIV")
        val thmsN = [def_atomname^".exhaust"]
        val fqn = Local_Theory.full_name lthy (Binding.name (mkN(pname, atomN)))
        val typ = Type(fqn, [])
      in
        lthy
        |> prove_UNIV (def_atomname) typ (map (fn c => Const(fqn^"."^c,typ)) types) thmsN
      end
  in
     lthy
     |> define_atom_dt
     |> prove_UNIV_atom
  end

fun def_fun_dt lthy =
  let
    val def_funname = mkN(pname, funN)
    val _ = info("  Defining "^def_funname)
    val types = map (fn x => ([],x)) (map fst (pub@priv))
    val ctyp = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, enum_constsN))), [])
  in
    ml_isar_wrapper.define_simple_datatype ([], def_funname) (types@[([ctyp],enumN)]) lthy
  end

fun def_fun_arity lthy =
  let
    val fqn_name = Local_Theory.full_name lthy (Binding.name (mkN(pname, funN)))
    val ctyp = Type(fqn_name, [])

    fun mk_rec_eq name (fname,arity) = (Free(name,ctyp --> natT)
                                        $Const(fqn_name^"."^fname,ctyp),
                                         mk_nat((Option.valOf o Int.fromString) arity))
    val name = mkN(pname, arityN)
    val _ = info("  Defining "^name)
    val ctyp' = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, enum_constsN))), [])
  in
    ml_isar_wrapper.define_simple_fun name
        ((map (mk_rec_eq name) (pub@priv))@[
                (Free(name, ctyp --> natT)
                     $(Const(fqn_name^"."^enumN, ctyp' --> ctyp)$(Term.dummy_pattern ctyp')),
                       mk_nat(0))]) lthy
  end

fun def_public lthy =
  let
    val fqn_name = Local_Theory.full_name lthy (Binding.name (mkN(pname, funN)))
    val ctyp = Type(fqn_name, [])
```

```
        fun mk_rec_eq name t fname = (Free(name, ctyp --> boolT)
                                          $Const(fqn_name^"."^fname,ctyp), t)
        val name = mkN(pname, publicN)
        val _ = info("  Defining "^name)
        val ctyp' = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, enum_constsN))), [])
      in
        ml_isar_wrapper.define_simple_fun name
            ((map (mk_rec_eq name (@{term "False"})) (map fst priv))
            @(map (mk_rec_eq name (@{term "True"})) (map fst pub))
            @[(Free(name, ctyp --> boolT)
                        $(Const(fqn_name^"."^enumN, ctyp' --> ctyp)$(Term.dummy_pattern ctyp')),
                           @{term "True"})]) lthy
      end

    fun def_gamma lthy =
      let
        fun optionT t = Type (@{type_name "option"}, [t])
        fun mk_Some t = Const (@{const_name "Some"}, t --> optionT t)
        fun mk_None t = Const (@{const_name "None"},  optionT t)

        val fqn_name = Local_Theory.full_name lthy (Binding.name (mkN(pname, funN)))
        val ctyp = Type(fqn_name, [])
        val atomFQN = Local_Theory.full_name lthy (Binding.name (mkN(pname, atomN)))
        val atomT = Type(atomFQN, [])

        fun mk_rec_eq name t fname = (Free(name, ctyp --> optionT atomT)
                                          $Const(fqn_name^"."^fname,ctyp), t)
        val name = mkN(pname, gammaN)
        val _ = info("  Defining "^name)
        val ctyp' = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, enum_constsN))), [])
      in
        ml_isar_wrapper.define_simple_fun name
            ((map (mk_rec_eq name ((mk_Some atomT)$(Const(atomFQN^"."^secret_typeN, atomT)))) (map fst
priv))
             @(map (mk_rec_eq name ((mk_Some atomT)$(Const(atomFQN^"."^other_pubconsts_typeN, atomT))))
(map fst pub_c))
             @[(Free(name, ctyp --> optionT atomT)
                         $(Const(fqn_name^"."^enumN, ctyp' --> ctyp)$(Term.dummy_pattern ctyp')),
                            (mk_Some atomT)$(Const(atomFQN^"."^enum_typeN,atomT)))]
             @(map (mk_rec_eq name (mk_None atomT)) (map fst pub_f)) ) lthy
      end

    fun def_ana lthy = let
      val pname = #name trac
      val (pub_f, pub_c, priv) = get_funs trac
      val pub = pub_f@pub_c

      val keyT = messageT' natT trac lthy

      val fqn_name = Local_Theory.full_name lthy (Binding.name (mkN(pname, funN)))
      val ctyp = Type(fqn_name, [])

      val ana_outputT = mk_prodT (mk_listT keyT, mk_listT natT)

      val default_output = mk_prod (mk_list keyT [], mk_list natT [])

      fun mk_ana_output ks rs = mk_prod (mk_list keyT ks, mk_list natT rs)

      fun mk_rec_eq name t fname = (Free(name, ctyp --> ana_outputT)
                                        $Term.Const(fqn_name^"."^fname,ctyp), t)
      val name = mkN(pname, anaN)
      val _ = info("  Defining "^name)
```

```
      val ctyp' = Type(Local_Theory.full_name lthy (Binding.name (mkN(pname, enum_constsN))), [])

      val ana_spec =
        let
          val toInt = Option.valOf o Int.fromString
          fun ana_arity (f,n) = (if is_priv_fun trac f then (toInt n)-1 else toInt n)
          fun check_valid_arity ((f,ps),ks,rs) =
            case List.find (fn g => f = fst g) pub_f of
              SOME (f',n) =>
                if length ps <> ana_arity (f',n)
                then error ("Invalid number of parameters in the analysis rule for " ^ f ^
                            " (expected " ^ Int.toString (ana_arity (f',n)) ^
                            " but got " ^ Int.toString (length ps) ^ ")")
                else ((f,ps),ks,rs)
            | NONE => error (f ^ " is not a declared function symbol of arity greater than zero")
          val transform_cMsg = transform_cMsg trac
          val rm_special_funs = rm_special_funs (fn ((f,_),_,_) => f)
          fun var_to_nat f xs x =
            let
              val n = snd (Option.valOf ((list_find (fn y => y = x) xs)))
            in
              if is_priv_fun trac f then mk_nat (1+n) else mk_nat n
            end
          fun c_to_h f xs t = ana_cMsg_to_hol (is_priv_fun trac f) t xs trac lthy
          fun keys f ps ks = map (c_to_h f ps o transform_cMsg o Trac_Term.certifyMsg [] []) ks
          fun results f ps rs = map (var_to_nat f ps) rs
          fun aux ((f,ps),ks,rs) = (f, mk_ana_output (keys f ps ks) (results f ps rs))
        in
          map (aux o check_valid_arity) (rm_special_funs (#analysis_spec trac))
        end

      val other_funs =
        filter (fn f => not (List.exists (fn g => f = g) (map fst ana_spec))) (map fst (pub@priv))
    in
      ml_isar_wrapper.define_simple_fun name
          ((map (fn (f,out) => mk_rec_eq name out f) ana_spec)
          @(map (mk_rec_eq name default_output) other_funs)
          @[(Free(name, ctyp --> ana_outputT)
                    $(Term.Const(fqn_name^"."^enumN, ctyp' --> ctyp)$(Term.dummy_pattern ctyp')),
                       default_output)]) lthy
    end

  in
    lthy |> def_atom
         |> def_fun_dt
         |> def_fun_arity
         |> def_public
         |> def_gamma
         |> def_ana
  end

fun define_term_model (trac:TracProtocol.protocol) lthy =
  let
    val _ = info("Defining term model")
  in
    lthy |> snd o def_types trac
         |> def_sets trac
         |> def_funs trac
  end

fun define_fixpoint fp trac print lthy =
  let
    val fp_name = mkN (#name trac, "fixpoint")
```

```
    val _ = info("Defining fixpoint")
    val _ = info("  Defining "^fp_name)
    val fp_triple = transform_fp trac fp
    val fp_triple_trm = fp_triple_to_hol fp_triple trac lthy
    val trac = TracProtocol.update_fixed_point trac (SOME fp_triple)
  in
    (trac, #2 (ml_isar_wrapper.define_constant_definition' (fp_name, fp_triple_trm) print lthy))
  end

fun define_protocol print ((trac:TracProtocol.protocol), lthy) = let
    val _ =
      if length (#transaction_spec trac) > 1
      then info("Defining protocols")
      else info("Defining protocol")
    val pname = #name trac

    val flat_type_spec = flatten_type_spec trac

    val mk_Transaction = mk_Transaction trac lthy

    val mk_Send = mk_Send_step trac lthy
    val mk_Receive = mk_Receive_step trac lthy
    val mk_InSet = mk_InSet_step trac lthy
    val mk_NotInSet = mk_NotInSet_step trac lthy
    val mk_Inequality = mk_Inequality_step trac lthy
    val mk_Insert = mk_Insert_step trac lthy
    val mk_Delete = mk_Delete_step trac lthy

    val star_label = mk_star_label
    val prot_label = mk_prot_label

    val certify_transation = TracProtocol.certifyTransaction

    fun mk_tname i (tr:TracProtocol.transaction_name) =
      let
        val x = #1 tr
        val y = case i of NONE => x | SOME n => mkN(n, x)
        val z = mkN("transaction", y)
      in mkN(pname, z)
      end

    fun def_transaction name_prefix prot_num (transaction:TracProtocol.cTransaction) lthy = let
      val defname = mk_tname name_prefix (#transaction transaction)
      val _ = info("  Defining "^defname)

      val receives     = #receive_actions     transaction
      val checkssingle = #checksingle_actions transaction
      val checksall    = #checkall_actions    transaction
      val updates      = #update_actions      transaction
      val sends        = #send_actions        transaction
      val fresh        = get_fresh_value_variables transaction
      val attack_signals = #attack_actions transaction

      val nonfresh_value_vars = get_nonfresh_value_variables transaction
      val value_vars = get_value_variables transaction
      val enum_vars  = get_enum_variables  transaction

      val (enum_ineqs, value_ineqs) = get_variable_restrictions transaction

      val transform_cMsg = transform_cMsg trac

      fun c_to_h trm = transaction_cMsg_to_hol (transform_cMsg trm) prot_num value_vars trac lthy
```

```
val abstract_over_enum_vars = fn x => fn y => fn z =>
  abstract_over_enum_vars x y z flat_type_spec trac lthy

fun mk_transaction_term (rcvs, chcksingle, chckall, upds, snds, frsh, atcks) =
  let
    open Trac_Term
    fun action_filter f (lbl,a) = case f a of SOME x => SOME (lbl,x) | NONE => NONE

    fun lbl_to_h (TracProtocol.LabelS) = star_label
      | lbl_to_h (TracProtocol.LabelN) = prot_label prot_num

    fun lbl_trm_to_h f (lbl,t) = f (lbl_to_h lbl) (c_to_h t)

    val S1 = map (lbl_trm_to_h mk_Receive)
                 (map_filter (action_filter TracProtocol.maybe_the_Receive) rcvs)

    val S2 =
      let
        fun aux (lbl,TracProtocol.cInequality (x,y)) =
              SOME (mk_Inequality (lbl_to_h lbl) (c_to_h x) (c_to_h y))
          | aux (lbl,TracProtocol.cInSet (e,s)) =
              SOME (mk_InSet (lbl_to_h lbl) (c_to_h e) (c_to_h s))
          | aux (lbl,TracProtocol.cNotInSet (e,s)) =
              SOME (mk_NotInSet (lbl_to_h lbl) (c_to_h e) (c_to_h s))
          | aux _ = NONE
      in
        map_filter aux chcksingle
      end

    val S3 =
      let
        fun arity s = case set_arity trac s of
            SOME n => n
          | NONE => error ("Not a set family: " ^ s)

        fun mk_evs s = map (fn n => ("X" ^ Int.toString n, "")) (0 upto ((arity s) -1))

        fun mk_trm (lbl,e,s) =
          let
            val ps = map (fn x => cVar (x,Untyped)) (map fst (mk_evs s))
          in
            mk_NotInSet (lbl_to_h lbl) (c_to_h e) (c_to_h (cSet (s,ps)))
          end

        fun mk_trms (lbl,(e,s)) =
          abstract_over_enum_vars (mk_evs s) [] (mk_trm (lbl,e,s))
      in
        map mk_trms (map_filter (action_filter TracProtocol.maybe_the_NotInAny) chckall)
      end

    val S4 = map (c_to_h o mk_Value_cVar) frsh

    val S5 =
      let
        fun aux (lbl,TracProtocol.cInsert (e,s)) =
              SOME (mk_Insert (lbl_to_h lbl) (c_to_h e) (c_to_h s))
          | aux (lbl,TracProtocol.cDelete (e,s)) =
              SOME (mk_Delete (lbl_to_h lbl) (c_to_h e) (c_to_h s))
          | aux _ = NONE
      in
        map_filter aux upds
      end
```

```
      val S6 =
        let val snds' = map_filter (action_filter TracProtocol.maybe_the_Send) snds
        in map (lbl_trm_to_h mk_Send) (snds'@map (fn (lbl,_) => (lbl,cAttack)) atcks) end
    in
      abstract_over_enum_vars enum_vars enum_ineqs (mk_Transaction S1 S2 S3 S4 S5 S6)
    end

  fun def_trm trm print lthy =
    #2 (ml_isar_wrapper.define_constant_definition' (defname, trm) print lthy)

  val additional_value_ineqs =
    let
      open Trac_Term
      open TracProtocol
      val poschecks = map_filter (maybe_the_InSet o snd) checkssingle
      val negchecks_single = map_filter (maybe_the_NotInSet o snd) checkssingle
      val negchecks_all = map_filter (maybe_the_NotInAny o snd) checksall

      fun aux' (cVar (x,ValueType),s) (cVar (y,ValueType),t) =
            if s = t then SOME (x,y) else NONE
        | aux' _ _ = NONE

      fun aux (x,cSet (s,ps)) = SOME (
            map_filter (aux' (x,cSet (s,ps))) negchecks_single@
            map_filter (aux' (x,s)) negchecks_all
          )
        | aux _ = NONE
    in
      List.concat (map_filter aux poschecks)
    end

  val all_value_ineqs = mk_unique (value_ineqs@additional_value_ineqs)

  val valvarsprod =
        filter (fn p => not (List.exists (fn q => p = q orelse swap p = q) all_value_ineqs))
               (list_triangle_product (fn x => fn y => (x,y)) nonfresh_value_vars)

  val transaction_trm0 = mk_transaction_term
                (receives, checkssingle, checksall, updates, sends, fresh, attack_signals)
in
  if null valvarsprod
  then def_trm transaction_trm0 print lthy
  else let
    val partitions = list_partitions nonfresh_value_vars all_value_ineqs
    val ps = filter (not o null) (map (filter (fn x => length x > 1)) partitions)

    fun mk_subst ps =
      let
        open Trac_Term
        fun aux [] = NONE
          | aux (x::xs) = SOME (map (fn y => (y,cVar (x,ValueType))) xs)
      in
        List.concat (map_filter aux ps)
      end

    fun apply d =
      let
        val ap = TracProtocol.subst_apply_actions d
        fun f (TracProtocol.cInequality (x,y)) = x <> y
          | f _ = true
        val checksingle' = filter (f o snd) (ap checkssingle)
      in
        (ap receives, checksingle', ap checksall, ap updates, ap sends, fresh, attack_signals)
```

```
      end

    val transaction_trms = transaction_trm0::map (mk_transaction_term o apply o mk_subst) ps
    val transaction_typ = Term.fastype_of transaction_trm0

    fun mk_concat_trm tau trms =
      Term.Const (@{const_name "concat"}, mk_listT tau --> tau) $ mk_list tau trms
  in
    def_trm (mk_concat_trm transaction_typ transaction_trms) print lthy
  end
end

val def_transactions =
  let
    val prots = map (fn (n,pr) => map (fn tr => (n,tr)) pr) (#transaction_spec trac)
    val lbls = list_upto (length prots)
    val lbl_prots = List.concat (map (fn i => map (fn tr => (i,tr)) (nth prots i)) lbls)
    val f = fold (fn (i,(n,tr)) => def_transaction n i (certify_transation tr))
  in
    f lbl_prots
  end

fun def_protocols lthy = let
    fun mk_prot_def (name,trm) lthy =
      let val _ = info("  Defining "^name)
      in #2 (ml_isar_wrapper.define_constant_definition' (name,trm) print lthy)
      end

    val prots = #transaction_spec trac
    val num_prots = length prots

    val pdefname = mkN(pname, "protocol")

    fun mk_tnames i =
      let
        val trs = case nth prots i of (j,prot) => map (fn tr => (j,tr)) prot
      in map (fn (j,s) => full_name (mk_tname j (#transaction s)) lthy) trs
      end

    val tnames = List.concat (map mk_tnames (list_upto num_prots))

    val pnames =
      let
        val f = fn i => (Int.toString i,nth prots i)
        val g = fn (i,(n,_)) => case n of NONE => i | SOME m => m
        val h = fn s => mkN (pdefname,s)
      in map (h o g o f) (list_upto num_prots)
      end

    val trtyp = prot_transactionT trac lthy
    val trstyp = mk_listT trtyp

    fun mk_prot_trm names =
      Term.Const (@{const_name "concat"}, mk_listT trstyp --> trstyp) $
      mk_list trstyp (map (fn x => Term.Const (x, trstyp)) names)

    val lthy =
      if num_prots > 1
      then fold (fn (i,pname) => mk_prot_def (pname, mk_prot_trm (mk_tnames i)))
                (map (fn i => (i, nth pnames i)) (list_upto num_prots))
                lthy
      else lthy
```

```
          val pnames' = map (fn n => full_name n lthy) pnames

          fun mk_prot_trm_with_star i =
            let
              fun f j =
                if j = i
                then Term.Const (nth pnames' j, trstyp)
                else (Term.Const (@{const_name "map"}, [trtyp --> trtyp, trstyp] ---> trstyp) $
                      Term.Const ("Transactions.transaction_star_proj", trtyp --> trtyp) $
                      Term.Const (nth pnames' j, trstyp))
            in
              Term.Const (@{const_name "concat"}, mk_listT trstyp --> trstyp) $
              mk_list trstyp (map f (list_upto num_prots))
            end

          val lthy =
            if num_prots > 1
            then fold (fn (i,pname) => mk_prot_def (pname, mk_prot_trm_with_star i))
                      (map (fn i => (i, nth pnames i ^ "_with_star")) (list_upto num_prots))
                      lthy
            else lthy
        in
          mk_prot_def (pdefname, mk_prot_trm (if num_prots > 1 then pnames' else tnames)) lthy
        end
      in
        (trac, lthy |> def_transactions |> def_protocols)
      end
end
⟩
```

ML⟨

```
structure trac = struct
  open Trac_Term

  val info = Output.information
  (* Define global configuration option "trac" *)
  (* val trac_fp_compute_binary_cfg =
      let
        val (trac_fp_compute_path_config, trac_fp_compute_path_setup) =
          Attrib.config_string (Binding.name "trac_fp_compute") (K "trac_fp_compute")
      in
        Context.>>(Context.map_theory trac_fp_compute_path_setup);
        trac_fp_compute_path_config
      end

  val trac_eval_cfg =
      let
        val (trac_fp_compute_eval_config, trac_fp_compute_eval) =
          Attrib.config_bool (Binding.name "trac_fp_compute_eval") (K false)
      in
        Context.>>(Context.map_theory trac_fp_compute_eval);
        trac_fp_compute_eval_config
      end *)

  type hide_tvar_tab = (TracProtocol.protocol) Symtab.table
  fun trac_eq (a, a') = (#name a) = (#name a')
  fun merge_trac_tab (tab,tab') = Symtab.merge trac_eq (tab,tab')
  structure Data = Generic_Data
  (
    type T = hide_tvar_tab
    val empty  = Symtab.empty:hide_tvar_tab
    val extend = I
    fun merge(t1,t2)  = merge_trac_tab (t1, t2)
```

```
);

fun update  p thy = Context.theory_of
                      ((Data.map (fn tab => Symtab.update (#name p, p) tab) (Context.Theory thy)))
fun lookup name thy = (Symtab.lookup ((Data.get o Context.Theory) thy) name,thy)

fun mk_abs_filename thy filename =
    let
      val filename = Path.explode filename
      val master_dir = Resources.master_directory thy
    in
      Path.implode (if (Path.is_absolute filename)
                    then filename
                    else Path.append master_dir filename)
    end

(* fun exec {trac_path, error_detail}  filename = let
      open OS.FileSys OS.Process

      val tmpname = tmpName()
      val err_tmpname = tmpName()
      fun plural 1 = "" | plural _ = "s"
      val trac = case trac_path of
                    SOME s => s
                  | NONE   => raise error ("trac_fp_compute_path not specified")
      val cmdline = trac ^ " \"" ^ filename ^ "\" > " ^ tmpname ^ " 2> " ^ err_tmpname
    in
      if isSuccess (system cmdline) then (OS.FileSys.remove err_tmpname; tmpname)
      else let val _ = OS.FileSys.remove tmpname
               val (msg, rest) = File.read_lines (Path.explode err_tmpname) |> chop error_detail
               val _ = OS.FileSys.remove err_tmpname
               val _ = warning ("trac failed on " ^ filename ^ "\nCommand: " ^ cmdline ^
                                "\n\nOutput:\n" ^
                                cat_lines (msg @ (if null rest then [] else
                                                  ["(... " ^ string_of_int (length rest) ^
                                                   " more line" ^ plural (length rest) ^ ")"])))
           in raise error ("trac failed on " ^ filename) end
    end *)

fun lookup_trac (pname:string) lthy =
  Option.valOf (fst (lookup pname (Proof_Context.theory_of lthy)))

fun def_fp fp_str print (trac, lthy) =
  let
    val fp = TracFpParser.parse_str fp_str
    val (trac,lthy) = trac_definitorial_package.define_fixpoint fp trac print lthy
    val lthy = Local_Theory.raw_theory (update trac) lthy
  in
    (trac, lthy)
  end

fun def_fp_file filename print (trac, lthy) = let
      val thy = Proof_Context.theory_of lthy
      val abs_filename = mk_abs_filename thy filename
      val fp = TracFpParser.parse_file abs_filename
      val (trac,lthy) = trac_definitorial_package.define_fixpoint fp trac print lthy
      val lthy = Local_Theory.raw_theory (update trac) lthy
    in
      (trac, lthy)
    end

fun def_fp_trac fp_filename print (trac, lthy) = let
      open OS.FileSys OS.Process
```

```
      val _ = info("Checking protocol specification with trac.")
      val thy = Proof_Context.theory_of lthy
      (* val trac =  Config.get_global thy trac_binary_cfg *)
      val abs_filename = mk_abs_filename thy fp_filename
      (* val fp_file = exec {error_detail=10, trac_path = SOME trac} abs_filename *)
      (* val fp_raw = File.read (Path.explode fp_file) *)
      val fp_raw = File.read (Path.explode abs_filename)
      val fp = TracFpParser.parse_str fp_raw
      (* val _ = OS.FileSys.remove fp_file *)
      val _ = if TracFpParser.attack fp
              then
                error ("  ATTACK found, skipping generating of Isabelle/HOL definitions.\n\n")
              else
                info("  No attack found, continue with generating Isabelle/HOL definitions.")
      val (trac,lthy) = trac_definitorial_package.define_fixpoint fp trac print lthy
      val lthy = Local_Theory.raw_theory (update trac) lthy
    in
      (trac, lthy)
    end

  fun def_trac_term_model str lthy = let
      val trac = TracProtocolParser.parse_str str
      val lthy = Local_Theory.raw_theory (update trac) lthy
      val lthy = trac_definitorial_package.define_term_model trac lthy
    in
      (trac, lthy)
    end

  val def_trac_protocol = trac_definitorial_package.define_protocol

  fun def_trac str print = def_trac_protocol print o def_trac_term_model str

  fun def_trac_file filename print lthy = let
      val trac_raw = File.read (Path.explode filename)
      val (trac,lthy) = def_trac trac_raw print lthy
      val lthy = Local_Theory.raw_theory (update trac) lthy
    in
      (trac, lthy)
    end

  fun def_trac_fp_trac trac_str print lthy = let
      open OS.FileSys OS.Process
      val (trac,lthy) = def_trac trac_str print lthy
      val tmpname = tmpName()
      val _ = File.write (Path.explode tmpname) trac_str
      val (trac,lthy) = def_fp_trac tmpname print (trac, lthy)
      val _ = OS.FileSys.remove tmpname
      val lthy = Local_Theory.raw_theory (update trac) lthy
    in
      lthy
    end

end
)
```

ML(

```
  val fileNameP = Parse.name -- Parse.name

  val _ = Outer_Syntax.local_theory' @{command_keyword "trac_import"}
        "Import protocol and fixpoint from trac files."
        (fileNameP >> (fn (trac_filename, fp_filename) => fn print =>
           trac.def_trac_file trac_filename print #>
           trac.def_fp_file fp_filename print #> snd));
```

```
  val _ = Outer_Syntax.local_theory' @{command_keyword "trac_import_trac"}
          "Import protocol from trac file and compute fixpoint with trac."
          (fileNameP >> (fn (trac_filename, fp_filename) => fn print =>
              trac.def_trac trac_filename print #> trac.def_fp_trac fp_filename print #> snd));

  val _ = Outer_Syntax.local_theory' @{command_keyword "trac_trac"}
          "Define protocol using trac format and compute fixpoint with trac."
          (Parse.cartouche >> (fn trac => fn print => trac.def_trac_fp_trac trac print));

  val _ = Outer_Syntax.local_theory' @{command_keyword "trac"}
          "Define protocol and (optionally) fixpoint using trac format."
          (Parse.cartouche -- Scan.optional Parse.cartouche "" >> (fn (trac,fp) => fn print =>
            if fp = ""
            then trac.def_trac trac print #> snd
            else trac.def_trac trac print #> trac.def_fp fp print #> snd));
⟩
```

ML⟨

```
val name_prefix_parser = Parse.!!! (Parse.name --| Parse.$$$ ":" -- Parse.name)

(* Original definition (opt_evaluator) copied from value_command.ml *)
val opt_proof_method_choice =
  Scan.optional (keyword⟨[⟩ |-- Parse.name --| keyword⟨]⟩) "safe";

(* Original definition (locale_expression) copied from parse_spec.ML *)
val opt_defs_list = Scan.optional
  (keyword⟨for⟩ |-- Scan.repeat1 Parse.name >>
      (fn xs => if length xs > 3 then error "Too many optional arguments" else xs))
  [];

val security_proof_locale_parser =
  name_prefix_parser -- opt_defs_list

val security_proof_locale_parser_with_method_choice =
  opt_proof_method_choice -- name_prefix_parser -- opt_defs_list


fun protocol_model_setup_proof_state name prefix lthy =
  let
    fun f x y z = ([((x,Position.none),((y,true),(Expression.Positional z,[])))],[])
    val _ = if name = "" then error "No name given" else ()
    val pexpr = f "stateful_protocol_model" name (protocol_model_interpretation_params prefix)
    val pdefs = protocol_model_interpretation_defs name
    val proof_state = Interpretation.global_interpretation_cmd pexpr pdefs lthy
  in
    proof_state
  end

fun protocol_security_proof_proof_state manual_proof name prefix opt_defs print lthy =
  let
    fun f x y z = ([((x,Position.none),((y,true),(Expression.Positional z,[])))],[])
    val _ = if name = "" then error "No name given" else ()
    val num_defs = length opt_defs
    val pparams = protocol_model_interpretation_params prefix
    val default_defs = [prefix ^ "_" ^ "protocol", prefix ^ "_" ^ "fixpoint"]
    fun g locale_name extra_params = f locale_name name (pparams@map SOME extra_params)
    val (prot_fp_smp_names, pexpr) = if manual_proof
      then (case num_defs of
        0 => (default_defs, g "secure_stateful_protocol'" default_defs)
      | 1 => (opt_defs, g "secure_stateful_protocol''" opt_defs)
      | 2 => (opt_defs, g "secure_stateful_protocol'" opt_defs)
      | _ => (opt_defs, g "secure_stateful_protocol" opt_defs))
```

```
      else (case num_defs of
        0 => (default_defs, g "secure_stateful_protocol''''" default_defs)
      | 1 => (opt_defs, g "secure_stateful_protocol''" opt_defs)
      | 2 => (opt_defs, g "secure_stateful_protocol''''" opt_defs)
      | _ => (opt_defs, g "secure_stateful_protocol'''" opt_defs))
    val proof_state = lthy |> declare_protocol_checks print
                           |> Interpretation.global_interpretation_cmd pexpr []
  in
    (prot_fp_smp_names, proof_state)
  end

val _ =
  Outer_Syntax.local_theory command_keyword⟨protocol_model_setup⟩
    "prove interpretation of protocol model locale into global theory"
    (name_prefix_parser >> (fn (name,prefix) => fn lthy =>
    let
      val proof_state = protocol_model_setup_proof_state name prefix lthy
      val meth =
        let
          val m = "protocol_model_interpretation"
          val _ = Output.information (
                    "Proving protocol model locale instance with proof method " ^ m)
        in
          Method.Source (Token.make_src (m, Position.none) [])
        end
    in
      ml_isar_wrapper.prove_state_simple meth proof_state
  end));

val _ =
  Outer_Syntax.local_theory_to_proof command_keyword⟨manual_protocol_model_setup⟩
    "prove interpretation of protocol model locale into global theory"
    (name_prefix_parser >> (fn (name,prefix) => fn lthy =>
    let
      val proof_state = protocol_model_setup_proof_state name prefix lthy
      val subgoal_proof = "  subgoal by protocol_model_subgoal\n"
      val _ = Output.information ("Example proof:\n" ^
                Active.sendback_markup_command ("  apply unfold_locales\n"^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                subgoal_proof^
                                                "  done\n"))
    in
      proof_state
  end));

val _ =
  Outer_Syntax.local_theory' command_keyword⟨protocol_security_proof⟩
    "prove interpretation of secure protocol locale into global theory"
    (security_proof_locale_parser_with_method_choice >> (fn params => fn print => fn lthy =>
    let
      val ((opt_meth_level,(name,prefix)),opt_defs) = params
      val (defs, proof_state) =
        protocol_security_proof_proof_state false name prefix opt_defs print lthy
      val num_defs = length defs
      val meth =
        let
          val m = case opt_meth_level of
              "safe" => "check_protocol" ^ "'" (* (if num_defs = 1 then "'" else "") *)
            | "unsafe" => "check_protocol_unsafe" ^ "'" (* (if num_defs = 1 then "'" else "") *)
            | _ => error ("Invalid option: " ^ opt_meth_level)
```

```
        val _ = Output.information (
                  "Proving security of protocol " ^ nth defs 0 ^ " with proof method " ^ m)
        val _ = if num_defs > 1 then Output.information ("Using fixpoint " ^ nth defs 1) else ()
        val _ = if num_defs > 2 then Output.information ("Using SMP set " ^ nth defs 2) else ()
      in
        Method.Source (Token.make_src (m, Position.none) [])
      end
  in
    ml_isar_wrapper.prove_state_simple meth proof_state
  end
  ));

val _ =
  Outer_Syntax.local_theory_to_proof' command_keyword⟨manual_protocol_security_proof⟩
    "prove interpretation of secure protocol locale into global theory"
    (security_proof_locale_parser >> (fn params => fn print => fn lthy =>
    let
      val ((name,prefix),opt_defs) = params
      val (defs, proof_state) =
        protocol_security_proof_proof_state true name prefix opt_defs print lthy
      val subgoal_proof =
        let
          val m = "code_simp" (* case opt_meth_level of
              "safe" => "code_simp"
            | "unsafe" => "eval"
            | _ => error ("Invalid option: " ^ opt_meth_level) *)
        in
          "  subgoal by " ^ m ^ "\n"
        end
      val _ = Output.information ("Example proof:\n" ^
                Active.sendback_markup_command ("  apply check_protocol_intro\n"^
                                                subgoal_proof^
                                                (if length defs = 1 then ""
                                                 else subgoal_proof^
                                                      subgoal_proof^
                                                      subgoal_proof^
                                                      subgoal_proof)^
                                                "  done\n"))
    in
      proof_state
    end
    ));
⟩

end
```

4 Examples

4.1 The Keyserver Protocol (Keyserver)

```
theory Keyserver
  imports "../PSPSP"
begin

declare [[code_timing]]

trac(
Protocol: keyserver

Types:
honest = {a,b,c}
server = {s}
agents = honest ++ server

Sets:
ring/1 valid/2 revoked/2

Functions:
Public sign/2 crypt/2 pair/2
Private inv/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
pair(X,Y) -> X,Y

Transactions:
# Out-of-band registration
outOfBand(A:honest,S:server)
  new NPK
  insert NPK ring(A)
  insert NPK valid(A,S)
  send NPK.

# User update key
keyUpdateUser(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

# Server update key
keyUpdateServer(A:honest,S:server,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
  PK in valid(A,S)
  NPK notin valid(_)
  NPK notin revoked(_)
  delete PK valid(A,S)
  insert PK revoked(A,S)
  insert NPK valid(A,S)
  send inv(PK).
```

```
# Attack definition
authAttack(A:honest,S:server,PK:value)
  receive inv(PK)
  PK in valid(A,S)
  attack.
)(
val(ring(A)) where A:honest
sign(inv(val(0)),pair(A,val(ring(A)))) where A:honest
inv(val(revoked(A,S))) where A:honest S:server
pair(A,val(ring(A))) where A:honest

occurs(val(ring(A))) where A:honest

timplies(val(ring(A)),val(ring(A),valid(A,S))) where A:honest S:server
timplies(val(ring(A)),val(0)) where A:honest
timplies(val(ring(A),valid(A,S)),val(valid(A,S))) where A:honest S:server
timplies(val(0),val(valid(A,S))) where A:honest S:server
timplies(val(valid(A,S)),val(revoked(A,S))) where A:honest S:server
)
```

4.1.1 Proof of security

protocol_model_setup *spm: keyserver*
compute_SMP *[optimized] keyserver_protocol keyserver_SMP*
manual_protocol_security_proof *ssp: keyserver*
  for *keyserver_protocol keyserver_fixpoint keyserver_SMP*
  apply *check_protocol_intro*
  subgoal by *code_simp*
  subgoal by *code_simp*
  subgoal by *code_simp*
  subgoal by *code_simp*
  subgoal by *code_simp*
  done

end

4.2 A Variant of the Keyserver Protocol (Keyserver2)

theory *Keyserver2*
  imports *"../PSPSP"*
begin

declare *[[code_timing]]*

trac(
Protocol: keyserver2

Types:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring'/1 seen/1 pubkeys/0 valid/1

Functions:
Public h/1 sign/2 crypt/2 scrypt/2 pair/2 update/3
Private inv/1 pw/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y

```
scrypt(X,Y) ? X -> Y
pair(X,Y) -> X,Y
update(X,Y,Z) -> X,Y,Z

Transactions:
passwordGenD(A:dishonest)
  send pw(A).

pubkeysGen()
  new PK
  insert PK pubkeys
  send PK.

updateKeyPw(A:honest,PK:value)
  PK in pubkeys
  new NPK
  insert NPK ring'(A)
  send NPK
  send crypt(PK,update(A,NPK,pw(A))).

updateKeyServerPw(A:agent,PK:value,NPK:value)
  receive crypt(PK,update(A,NPK,pw(A)))
  PK in pubkeys
  NPK notin pubkeys
  NPK notin seen(_)
  insert NPK valid(A)
  insert NPK seen(A).

authAttack2(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
)
```

4.2.1 Proof of security

protocol_model_setup *spm: keyserver2*
compute_fixpoint *keyserver2_protocol keyserver2_fixpoint*
protocol_security_proof *ssp: keyserver2*

4.2.2 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *keyserver2_enum_consts.nchotomy*
thm *keyserver2_sets.nchotomy*
thm *keyserver2_fun.nchotomy*
thm *keyserver2_atom.nchotomy*
thm *keyserver2_arity.simps*
thm *keyserver2_public.simps*
thm *keyserver2_*Γ*.simps*
thm *keyserver2_Ana.simps*

thm *keyserver2_transaction_passwordGenD_def*
thm *keyserver2_transaction_pubkeysGen_def*
thm *keyserver2_transaction_updateKeyPw_def*
thm *keyserver2_transaction_updateKeyServerPw_def*
thm *keyserver2_transaction_authAttack2_def*
thm *keyserver2_protocol_def*

thm *keyserver2_fixpoint_def*

end

4.3 The Composition of the Two Keyserver Protocols (Keyserver_Composition)

```
theory Keyserver_Composition
  imports "../PSPSP"
begin

declare [[code_timing]]

trac(
Protocol: kscomp

Types:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring/1 valid/1 revoked/1 deleted/1
ring'/1 seen/1 pubkeys/0

Functions:
Public h/1 sign/2 crypt/2 scrypt/2 pair/2 update/3
Private inv/1 pw/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
scrypt(X,Y) ? X -> Y
pair(X,Y) -> X,Y
update(X,Y,Z) -> X,Y,Z

Transactions:
### The signature-based keyserver protocol
p1_outOfBand(A:honest)
  new PK
  insert PK ring(A)
* insert PK valid(A)
  send PK.

p1_oufOfBandD(A:dishonest)
  new PK
* insert PK valid(A)
  send PK
  send inv(PK).

p1_updateKey(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert PK deleted(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

p1_updateKeyServer(A:agent,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
* PK in valid(A)
* NPK notin valid(_)
  NPK notin revoked(_)
* delete PK valid(A)
  insert PK revoked(A)
* insert NPK valid(A)
```

```
  send inv(PK).

p1_authAttack(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
  attack.

### The password-based keyserver protocol
p2_passwordGenD(A:dishonest)
  send pw(A).

p2_pubkeysGen()
  new PK
  insert PK pubkeys
  send PK.

p2_updateKeyPw(A:honest,PK:value)
  PK in pubkeys
  new NPK
# NOTE: The ring' sets are not used elsewhere, but we have to avoid that the fresh keys generated
#       by this rule are abstracted to the empty abstraction, and so we insert them into a ring'
#       set. Otherwise the two protocols would have too many abstractions in common (in particular,
#       the empty abstraction) which leads to false attacks in the composed protocol (probably
#       because the term implication graphs of the two protocols then become 'linked' through the
#       empty abstraction)
  insert NPK ring'(A)
  send NPK
  send crypt(PK,update(A,NPK,pw(A))).

#Transactions of p2:
p2_updateKeyServerPw(A:agent,PK:value,NPK:value)
receive crypt(PK,update(A,NPK,pw(A)))
  PK in pubkeys
  NPK notin pubkeys
  NPK notin seen(_)
* insert NPK valid(A)
  insert NPK seen(A).

p2_authAttack2(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
  attack.
〉〈
sign(inv(val(deleted(A))),pair(A,val(ring(A)))) where A:honest
sign(inv(val(deleted(A),valid(B))),pair(A,val(ring(A)))) where A:honest B:dishonest
sign(inv(val(deleted(A),seen(B),valid(B))),pair(A,val(ring(A)))) where A:honest B:dishonest
sign(inv(val(deleted(A),valid(A))),pair(A,val(ring(A)))) where A:honest B:dishonest
sign(inv(val(deleted(A),seen(B),valid(B),valid(A))),pair(A,val(ring(A)))) where A:honest B:dishonest
pair(A,val(ring(A))) where A:honest
inv(val(deleted(A),revoked(A))) where A:honest
inv(val(valid(A))) where A:dishonest
inv(val(revoked(A))) where A:dishonest
inv(val(revoked(A),seen(A))) where A:dishonest
inv(val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
inv(val(revoked(A),deleted(A),seen(B),valid(B))) where A:honest B:dishonest
occurs(val(ring(A))) where A:honest
occurs(val(valid(A))) where A:dishonest
occurs(val(ring'(A))) where A:honest
occurs(val(pubkeys))
occurs(val(valid(A),ring(A))) where A:honest
pw(A) where A:dishonest
crypt(val(pubkeys),update(A,val(ring'(A)),pw(A))) where A:honest
val(ring(A)) where A:honest
```

```
val(valid(A)) where A:dishonest
val(ring'(A)) where A:honest
val(pubkeys)
val(valid(A),ring(A)) where A:honest

timplies(val(pubkeys),val(valid(A),pubkeys)) where A:dishonest

timplies(val(ring'(A)),val(ring'(A),valid(B))) where A:honest B:dishonest
timplies(val(ring'(A)),val(ring'(A),valid(A),seen(A))) where A:honest
timplies(val(ring'(A)),val(ring'(A),valid(A),seen(A),valid(B))) where A:honest B:dishonest
timplies(val(ring'(A)),val(seen(B),valid(B),ring'(A))) where A:honest B:dishonest

timplies(val(ring'(A),valid(B)),val(ring'(A),valid(A),seen(A),valid(B))) where A:honest B:dishonest
timplies(val(ring'(A),valid(B)),val(seen(B),valid(B),ring'(A))) where A:honest B:dishonest

timplies(val(ring(A)),val(ring(A),valid(A))) where A:honest
timplies(val(ring(A)),val(ring(A),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(deleted(A))) where A:honest
timplies(val(ring(A)),val(revoked(A),deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(revoked(A),deleted(A),seen(B),revoked(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(ring(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(valid(A),deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(valid(A),ring(A),seen(B),valid(B))) where A:honest B:dishonest

timplies(val(ring(A),valid(A)),val(deleted(A),valid(A))) where A:honest
timplies(val(ring(A),valid(B)),val(deleted(A),valid(B))) where A:honest B:dishonest
timplies(val(ring(A),valid(A)),val(deleted(A),revoked(A))) where A:honest

timplies(val(deleted(A)),val(deleted(A),valid(A))) where A:honest
timplies(val(deleted(A)),val(deleted(A),valid(B))) where A:honest B:dishonest
timplies(val(deleted(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(deleted(A)),val(seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(deleted(A)),val(seen(B),valid(B),valid(A),deleted(A))) where A:honest B:dishonest

timplies(val(revoked(A)),val(seen(A),revoked(A))) where A:dishonest
timplies(val(revoked(A)),val(seen(A),revoked(A),valid(A))) where A:dishonest

timplies(val(revoked(A),deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(revoked(A),deleted(A)),val(seen(B),valid(B),revoked(A),deleted(A))) where A:honest B:dishonest

timplies(val(seen(B),valid(B),deleted(A),valid(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest
B:dishonest
timplies(val(seen(B),valid(B),deleted(A),valid(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where
A:honest B:dishonest
timplies(val(seen(B),valid(B),revoked(A),deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where
A:honest B:dishonest
timplies(val(seen(A),valid(A)),val(revoked(A),seen(A))) where A:dishonest
timplies(val(seen(A),valid(A),revoked(A)),val(seen(A),revoked(A))) where A:dishonest
timplies(val(seen(B),valid(B),ring(A)),val(deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(seen(B),valid(B),valid(A),ring(A)),val(deleted(A),seen(B),valid(B),valid(A))) where A:honest
B:dishonest
timplies(val(seen(B),valid(B),valid(A),ring(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest
B:dishonest
timplies(val(seen(B),valid(B),valid(A),ring(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest
B:dishonest

timplies(val(valid(A)),val(revoked(A))) where A:dishonest

timplies(val(valid(A),deleted(A)),val(deleted(A),revoked(A))) where A:honest
timplies(val(valid(A),deleted(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(valid(A),deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
```

```
timplies(val(valid(A),deleted(A)),val(seen(B),valid(B),valid(A),deleted(A))) where A:honest B:dishonest

timplies(val(ring(A),valid(A)),val(deleted(A),seen(B),valid(B),valid(A))) where A:honest B:dishonest
timplies(val(ring(A),valid(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(ring(A),valid(A)),val(seen(B),valid(B),valid(A),ring(A))) where A:honest B:dishonest
timplies(val(valid(B),deleted(A)),val(seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(ring(A),valid(B)),val(deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A),valid(B)),val(seen(B),valid(B),ring(A))) where A:honest B:dishonest

timplies(val(valid(A)),val(seen(A),valid(A))) where A:dishonest
⟩
```

4.3.1 Proof: The composition of the two keyserver protocols is secure

```
protocol_model_setup spm: kscomp
setup_protocol_checks spm kscomp_protocol
manual_protocol_security_proof ssp: kscomp
  apply check_protocol_intro
  subgoal by code_simp
  subgoal
    apply coverage_check_intro
    subgoal by code_simp
    subgoal by code_simp
    subgoal by eval
    subgoal by eval
    subgoal by eval
    subgoal by code_simp
    subgoal by code_simp
    subgoal by eval
    subgoal by eval
    subgoal by eval
    done
  subgoal by eval
  subgoal by eval
  subgoal
    apply (unfold spm.wellformed_fixpoint_def Let_def case_prod_unfold; intro conjI)
    subgoal by code_simp
    subgoal by code_simp
    subgoal by eval
    subgoal by code_simp
    subgoal by code_simp
    done
  done
```

4.3.2 The generated theorems and definitions

```
thm ssp.protocol_secure

thm kscomp_enum_consts.nchotomy
thm kscomp_sets.nchotomy
thm kscomp_fun.nchotomy
thm kscomp_atom.nchotomy
thm kscomp_arity.simps
thm kscomp_public.simps
thm kscomp_Γ.simps
thm kscomp_Ana.simps

thm kscomp_transaction_p1_outOfBand_def
thm kscomp_transaction_p1_oufOfBandD_def
thm kscomp_transaction_p1_updateKey_def
thm kscomp_transaction_p1_updateKeyServer_def
thm kscomp_transaction_p1_authAttack_def
thm kscomp_transaction_p2_passwordGenD_def
```

thm *kscomp_transaction_p2_pubkeysGen_def*
thm *kscomp_transaction_p2_updateKeyPw_def*
thm *kscomp_transaction_p2_updateKeyServerPw_def*
thm *kscomp_transaction_p2_authAttack2_def*
thm *kscomp_protocol_def*

thm *kscomp_fixpoint_def*

end

4.4 The PKCS Model, Scenario 3 (PKCS_Model03)

theory *PKCS_Model03*
  imports *"../../PSPSP"*

begin

declare *[[code_timing]]*

trac(
```
Protocol: ATTACK_UNSET

Types:
token   = {token1}

Sets:
extract/1 wrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/1
Private inv/1

Analysis:
senc(M,K2) ? K2 -> M  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega specification.
                      #M was type untyped

Transactions:

iik1()
new K1
insert K1 sensitive(token1)
insert K1 extract(token1)
send h(K1).

iik2()
new K2
insert K2 wrap(token1)
send h(K2).

# ======================wrap================
wrap(K1:value,K2:value)
receive h(K1)
receive h(K2)
K1 in extract(token1)
K2 in wrap(token1)
send senc(K1,K2).

# ======================set wrap================
setwrap(K2:value)
receive h(K2)
K2 notin decrypt(token1)
insert K2 wrap(token1).
```

```
# ======================set decrypt================
setdecrypt(K2:value)
receive h(K2)
K2 notin wrap(token1)
insert K2 decrypt(token1).

# ======================decrypt================
decrypt1(K2:value,M:value)  #M was untyped in the AIF-omega specification.
receive h(K2)
receive senc(M,K2)
K2 in decrypt(token1)
send M.

# ======================attacks================
attack1(K1:value)
receive K1
K1 in sensitive(token1)
attack.
)
```

4.4.1 Protocol model setup

protocol_model_setup *spm: ATTACK_UNSET*

4.4.2 Fixpoint computation

compute_fixpoint *ATTACK_UNSET_protocol ATTACK_UNSET_fixpoint*
compute_SMP *[optimized] ATTACK_UNSET_protocol ATTACK_UNSET_SMP*

4.4.3 Proof of security

manual_protocol_security_proof *ssp: ATTACK_UNSET*
  for *ATTACK_UNSET_protocol ATTACK_UNSET_fixpoint ATTACK_UNSET_SMP*
  apply *check_protocol_intro*
  subgoal by *code_simp*
  subgoal by *code_simp*
  subgoal by *code_simp*
  subgoal by *code_simp*
  subgoal by *code_simp*
  done

4.4.4 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *ATTACK_UNSET_enum_consts.nchotomy*
thm *ATTACK_UNSET_sets.nchotomy*
thm *ATTACK_UNSET_fun.nchotomy*
thm *ATTACK_UNSET_atom.nchotomy*
thm *ATTACK_UNSET_arity.simps*
thm *ATTACK_UNSET_public.simps*
thm *ATTACK_UNSET_*Γ*.simps*
thm *ATTACK_UNSET_Ana.simps*

thm *ATTACK_UNSET_transaction_iik1_def*
thm *ATTACK_UNSET_transaction_iik2_def*
thm *ATTACK_UNSET_transaction_wrap_def*
thm *ATTACK_UNSET_transaction_setwrap_def*
thm *ATTACK_UNSET_transaction_setdecrypt_def*
thm *ATTACK_UNSET_transaction_decrypt1_def*
thm *ATTACK_UNSET_transaction_attack1_def*

thm *ATTACK_UNSET_protocol_def*

thm *ATTACK_UNSET_fixpoint_def*
thm *ATTACK_UNSET_SMP_def*

end

4.5 The PKCS Protocol, Scenario 7 (PKCS_Model07)

theory *PKCS_Model07*
imports *"../../PSPSP"*

begin

declare *[[code_timing]]*

trac(
```
Protocol: RE_IMPORT_ATT

Types:
token   = {token1}

Sets:
extract/1 wrap/1 unwrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/2 bind/2
Private inv/1

Analysis:
senc(M1,K2) ? K2 -> M1  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega specification.
                        #M1 was type untyped

Transactions:

iik1()
new K1
new N1
insert N1 sensitive(token1)
insert N1 extract(token1)
insert K1 sensitive(token1)
send h(N1,K1).

iik2()
new K2
new N2
insert N2 wrap(token1)
insert N2 extract(token1)
send h(N2,K2).

# =====set wrap=====
setwrap(N2:value,K2:value)
receive h(N2,K2)
N2 notin sensitive(token1)
N2 notin decrypt(token1)
insert N2 wrap(token1).

# =====set unwrap===
setunwrap(N2:value,K2:value)
receive h(N2,K2)
N2 notin sensitive(token1)
insert N2 unwrap(token1).
```

```
# =====unwrap, generate new handler======
#-----------the senstive attr copy-------------
unwrapsensitive(M2:value, K2:value, N1:value, N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 in sensitive(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew sensitive(token1)
send h(Nnew,M2).

#-----------the wrap attr copy-------------
wrapattr(M2:value, K2:value, N1:value, N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 in wrap(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew wrap(token1)
send h(Nnew,M2).

#-----------the decrypt attr copy-------------
decrypt1attr(M2:value,K2:value,N1:value,N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 in decrypt(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew decrypt(token1)
send h(Nnew,M2).

decrypt2attr(M2:value,K2:value,N1:value,N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 notin sensitive(token1)
N1 notin wrap(token1)
N1 notin decrypt(token1)
N2 in unwrap(token1)
new Nnew
send h(Nnew,M2).

# ======================wrap================
wrap(N1:value,K1:value,N2:value,K2:value)
receive h(N1,K1)
receive h(N2,K2)
N1 in extract(token1)
N2 in wrap(token1)
send senc(K1,K2)
send bind(N1,K1).

# =====set decrypt===
setdecrypt(Nnew:value, K2:value)
receive h(Nnew,K2)
Nnew notin wrap(token1)
insert Nnew decrypt(token1).

# ======================decrypt================
decrypt1(Nnew:value, K2:value,M1:value) #M1 was untyped in the AIF-omega specification.
```

```
receive h(Nnew,K2)
receive senc(M1,K2)
Nnew in decrypt(token1)
delete Nnew decrypt(token1)
send M1.

# ======================attacks================
attack1(K1:value)
receive K1
K1 in sensitive(token1)
attack.
)
```

4.5.1 Protocol model setup

protocol_model_setup *spm: RE_IMPORT_ATT*

4.5.2 Fixpoint computation

compute_fixpoint *RE_IMPORT_ATT_protocol RE_IMPORT_ATT_fixpoint*
compute_SMP *[optimized] RE_IMPORT_ATT_protocol RE_IMPORT_ATT_SMP*

4.5.3 Proof of security

protocol_security_proof *[unsafe] ssp: RE_IMPORT_ATT*
  for *RE_IMPORT_ATT_protocol RE_IMPORT_ATT_fixpoint RE_IMPORT_ATT_SMP*

4.5.4 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *RE_IMPORT_ATT_enum_consts.nchotomy*
thm *RE_IMPORT_ATT_sets.nchotomy*
thm *RE_IMPORT_ATT_fun.nchotomy*
thm *RE_IMPORT_ATT_atom.nchotomy*
thm *RE_IMPORT_ATT_arity.simps*
thm *RE_IMPORT_ATT_public.simps*
thm *RE_IMPORT_ATT_Γ.simps*
thm *RE_IMPORT_ATT_Ana.simps*

thm *RE_IMPORT_ATT_transaction_iik1_def*
thm *RE_IMPORT_ATT_transaction_iik2_def*
thm *RE_IMPORT_ATT_transaction_setwrap_def*
thm *RE_IMPORT_ATT_transaction_setunwrap_def*
thm *RE_IMPORT_ATT_transaction_unwrapsensitive_def*
thm *RE_IMPORT_ATT_transaction_wrapattr_def*
thm *RE_IMPORT_ATT_transaction_decrypt1attr_def*
thm *RE_IMPORT_ATT_transaction_decrypt2attr_def*
thm *RE_IMPORT_ATT_transaction_wrap_def*
thm *RE_IMPORT_ATT_transaction_setdecrypt_def*
thm *RE_IMPORT_ATT_transaction_decrypt1_def*
thm *RE_IMPORT_ATT_transaction_attack1_def*

thm *RE_IMPORT_ATT_protocol_def*

thm *RE_IMPORT_ATT_fixpoint_def*
thm *RE_IMPORT_ATT_SMP_def*

end

4.6 The PKCS Protocol, Scenario 9 (PKCS_Model09)

theory *PKCS_Model09*

```
  imports "../../PSPSP"

begin

declare [[code_timing]]

trac(
Protocol: LOSS_KEY_ATT

Types:
token   = {token1}

Sets:
extract/1 wrap/1 unwrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/2 bind/3
Private inv/1

Analysis:
senc(M1,K2) ? K2 -> M1  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega specification.
                        #M1 was type untyped

Transactions:
iik1()
new K1
new N1
insert N1 sensitive(token1)
insert N1 extract(token1)
insert K1 sensitive(token1)
send h(N1,K1).

iik2()
new K2
new N2
insert N2 wrap(token1)
insert N2 extract(token1)
send h(N2,K2).

iik3()
new K3
new N3
insert N3 extract(token1)
insert N3 decrypt(token1)
insert K3 decrypt(token1)
send h(N3,K3)
send K3.

# =====set wrap=====
setwrap(N2:value,K2:value) where N2 != K2
receive h(N2,K2)
N2 notin sensitive(token1)
N2 notin decrypt(token1)
insert N2 wrap(token1).

# =====set unwrap===
setunwrap(N2:value,K2:value) where N2 != K2
receive h(N2,K2)
N2 notin sensitive(token1)
insert N2 unwrap(token1).

# =====unwrap, generate new handler======
#-----------add the wrap attr copy-------------
```

```
unwrapWrap(M2:value,K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 in wrap(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew wrap(token1)
send h(Nnew,M2).

#-----------add the senstive attr copy-------------
unwrapSens(M2:value,K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 in sensitive(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew sensitive(token1)
send h(Nnew,M2).

#-----------add the decrypt attr copy-------------
decrypt1Attr(M2:value, K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 in decrypt(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew decrypt(token1)
send h(Nnew,M2).

decrypt2Attr(M2:value, K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 notin wrap(token1)
N1 notin sensitive(token1)
N1 notin decrypt(token1)
N2 in unwrap(token1)
new Nnew
send h(Nnew,M2).

# ======================wrap================
wrap(N1:value,K1:value, N2:value, K2:value) where N1 != N2, N1 != K2, N1 != K1, N2 != K2, N2 != K1, K2 !=
K1
receive h(N1,K1)
receive h(N2,K2)
N1 in extract(token1)
N2 in wrap(token1)
send senc(K1,K2)
send bind(N1,K1,K2).

# ======================bind generation================
bind1(K3:value,N2:value,K2:value, K1:value) where K3 != N2, K3 != K2, K3 != K1, N2 != K2, N2 != K1, K2 !=
K1
receive K3
receive h(N2,K2)
send bind(N2,K3,K3).
```

```
bind2(K3:value,N2:value,K2:value, K1:value) where K3 != N2, K3 != K2, K3 != K1, N2 != K2, N2 != K1, K2 !=
K1
receive K3
receive K1
receive h(N2,K2)
send bind(N2,K1,K3)
send bind(N2,K3,K1).

# =====set decrypt===
setdecrypt(Nnew:value,K2:value) where Nnew != K2
receive h(Nnew,K2)
Nnew notin wrap(token1)
insert Nnew decrypt(token1).

# ======================decrypt================
decrypt1(Nnew:value,K2:value,M1:value) where Nnew != K2, Nnew != M1, K2 != M1 #M1 was untyped in the AIF-omega
specification.
receive h(Nnew,K2)
receive senc(M1,K2)
Nnew in decrypt(token1)
send M1.

# ======================attacks================
attack1(K1:value)
receive K1
K1 in sensitive(token1)
attack.

)
```

4.6.1 Protocol model setup

protocol_model_setup *spm: LOSS_KEY_ATT*

4.6.2 Fixpoint computation

compute_fixpoint *LOSS_KEY_ATT_protocol LOSS_KEY_ATT_fixpoint*

The fixpoint contains an attack signal

value *"attack_notin_fixpoint LOSS_KEY_ATT_fixpoint"*

4.6.3 The generated theorems and definitions

thm *LOSS_KEY_ATT_enum_consts.nchotomy*
thm *LOSS_KEY_ATT_sets.nchotomy*
thm *LOSS_KEY_ATT_fun.nchotomy*
thm *LOSS_KEY_ATT_atom.nchotomy*
thm *LOSS_KEY_ATT_arity.simps*
thm *LOSS_KEY_ATT_public.simps*
thm *LOSS_KEY_ATT_*Γ*.simps*
thm *LOSS_KEY_ATT_Ana.simps*

thm *LOSS_KEY_ATT_transaction_iik1_def*
thm *LOSS_KEY_ATT_transaction_iik2_def*
thm *LOSS_KEY_ATT_transaction_iik3_def*
thm *LOSS_KEY_ATT_transaction_setwrap_def*
thm *LOSS_KEY_ATT_transaction_setunwrap_def*
thm *LOSS_KEY_ATT_transaction_unwrapWrap_def*
thm *LOSS_KEY_ATT_transaction_unwrapSens_def*
thm *LOSS_KEY_ATT_transaction_decrypt1Attr_def*
thm *LOSS_KEY_ATT_transaction_decrypt2Attr_def*
thm *LOSS_KEY_ATT_transaction_wrap_def*

thm *LOSS_KEY_ATT_transaction_bind1_def*
thm *LOSS_KEY_ATT_transaction_bind2_def*
thm *LOSS_KEY_ATT_transaction_setdecrypt_def*
thm *LOSS_KEY_ATT_transaction_decrypt1_def*
thm *LOSS_KEY_ATT_transaction_attack1_def*

thm *LOSS_KEY_ATT_protocol_def*
thm *LOSS_KEY_ATT_fixpoint_def*

end

Bibliography

[1] A. D. Brucker and S. Mödersheim. Integrating Automated and Interactive Protocol Verification. In P. Degano and J. D. Guttman, editors, *Formal Aspects in Security and Trust, 6th International Workshop, FAST 2009, Eindhoven, The Netherlands, November 5-6, 2009, Revised Selected Papers*, volume 5983 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2009. doi: 10.1007/978-3-642-12459-4_18.

[2] 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`.

[3] 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.

[4] 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.

[5] 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.

[6] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful Protocol Composition and Typing. *Archive of Formal Proofs*, Apr. 2020. ISSN 2150-914x. `http://isa-afp.org/entries/Stateful_Protocol_Composition_and_Typing.html`, Formal proof development.