

Formalization of Generic Authenticated Data Structures

Matthias Brun Dmitriy Traytel

March 8, 2026

Abstract

Authenticated data structures are a technique for outsourcing data storage and maintenance to an untrusted server. The server is required to produce an efficiently checkable and cryptographically secure proof that it carried out precisely the requested computation. Miller et al. [2] introduced λ^\bullet (pronounced *lambda auth*)—a functional programming language with a built-in primitive authentication construct, which supports a wide range of user-specified authenticated data structures while guaranteeing certain correctness and security properties for all well-typed programs. We formalize λ^\bullet and prove its correctness and security properties. With Isabelle’s help, we uncover and repair several mistakes in the informal proofs and lemma statements. Our findings are summarized in an ITP’19 paper [1].

Contents

1	Preliminaries	2
2	Syntax of λ^\bullet	5
3	Semantics of λ^\bullet	7
3.1	Equivariant Hash Function	7
3.2	Substitution	9
3.3	Weak Typing Judgement	11
3.4	Erasure of Authenticated Types	15
3.5	Strong Typing Judgement	15
3.6	Shallow Projection	16
3.7	Small-step Semantics	17
3.8	Type Progress	23
3.9	Weak Type Preservation	23
3.10	Corrected Lemma 1 from Miller et al. [2]: Weak Type Soundness	26
4	Agreement Relation	26
5	Formalization of Miller et al.’s [2] Main Results	35
5.1	Lemma 2.1	35
5.2	Counterexample to Lemma 2.2	35
5.3	Lemma 2.3	37
5.4	Lemma 2.4	37
5.5	Lemma 3	38
5.6	Lemma 4	39
5.7	Lemma 5: Single-Step Correctness	40
5.8	Lemma 6: Single-Step Security	46
5.9	Theorem 1: Correctness	51
5.10	Counterexamples to Theorem 1: Security	51
5.11	Corrected Theorem 1: Security	58
5.12	Remark 1	59

1 Preliminaries

Auxiliary freshness lemmas and simplifier setup.

declare

fresh_star_Pair[simp] *fresh_star_insert*[simp] *fresh_Nil*[simp]
pure_supp[simp] *pure_fresh*[simp]

lemma *fresh_star_Nil*[simp]: {} #* t
unfolding *fresh_star_def* **by** *auto*

lemma *supp_flip*[simp]:
fixes a b :: _ :: at
shows *supp* (a ↔ b) = (if a = b then {} else {atom a, atom b})
by (auto simp: *flip_def* *supp_swap*)

lemma *Abs_lst_eq_flipI*:
fixes a b :: _ :: at **and** t :: _ :: fs
assumes atom b # t
shows [[atom a]]lst. t = [[atom b]]lst. (a ↔ b) · t
by (metis *Abs1_eq_iff*(3) *assms* *flip_commute* *flip_def* *swap_fresh_fresh*)

lemma *atom_not_fresh_eq*:
assumes ¬ atom a # x
shows a = x
using *assms* *atom_eq_iff* *fresh_ineq_at_base* **by** *blast*

lemma *fresh_set_fresh_forall*:
shows atom y # xs = (∀ x ∈ set xs. atom y # x)
by (induct xs) (simp_all add: *fresh_Cons*)

lemma *finite_fresh_set_fresh_all*[simp]:
fixes S :: (_ :: fs) set
shows finite S ⇒ atom a # S ↔ (∀ x ∈ S. atom a # x)
unfolding *fresh_def* **by** (simp add: *supp_of_finite_sets*)

lemma *case_option_eqvt*[eqvt]:
p · case_option a b opt = case_option (p · a) (p · b) (p · opt)
by (cases opt) *auto*

Nominal setup for finite maps.

abbreviation *fmap_update* (⟨_ '(_ \$\$:= _)⟩ [1000,0,0] 1000) **where** *fmap_update* Γ x τ ≡ *fmupd* x τ Γ

notation *fmlookup* (infixl ⟨\$\$⟩ 999)

notation *fmempty* (⟨{\$\$}⟩)

instantiation *fmap* :: (pt, pt) pt
begin

unbundle *fmap.lifting*

lift_definition
permute_fmap :: perm ⇒ ('a, 'b) *fmap* ⇒ ('a, 'b) *fmap*
is
permute :: perm ⇒ ('a → 'b) ⇒ ('a → 'b)

```

proof –
  fix  $p$  and  $f :: 'a \rightarrow 'b$ 
  assume  $finite (dom\ f)$ 
  then show  $finite (dom (p \cdot f))$ 
  proof (rule finite_surj[of _ _ permute p]; unfold dom_def, safe)
    fix  $x\ y$ 
    assume  $some: (p \cdot f)\ x = Some\ y$ 
    show  $x \in permute\ p\ \{a.\ f\ a \neq None\}$ 
    proof (rule image_eqI[of _ _ - p \cdot x])
      from  $some$  show  $- p \cdot x \in \{a.\ f\ a \neq None\}$ 
      by (auto simp: permute_self pemute_minus_self
        dest: arg_cong[of _ _ permute (- p)] intro!: exI[of _ - p \cdot y])
    qed (simp only: permute_minus_cancel)
  qed
qed

```

```

instance
proof
  fix  $x :: ('a, 'b)\ fmap$ 
  show  $0 \cdot x = x$ 
  by transfer simp
next
  fix  $p\ q$  and  $x :: ('a, 'b)\ fmap$ 
  show  $(p + q) \cdot x = p \cdot q \cdot x$ 
  by transfer simp
qed

```

end

```

lemma fmempty_eqvt[eqvt]:
  shows  $(p \cdot \{\$\$ \}) = \{\$\$ \}$ 
  by transfer simp

lemma fmap_update_eqvt[eqvt]:
  shows  $(p \cdot f(a\ \$\$ := b)) = (p \cdot f)((p \cdot a)\ \$\$ := (p \cdot b))$ 
  by transfer (simp add: map_upd_def)

```

```

lemma fmap_apply_eqvt[eqvt]:
  shows  $(p \cdot (f\ \$\$\ b)) = (p \cdot f)\ \$\$ (p \cdot b)$ 
  by transfer simp

```

```

lemma fresh_fmempty[simp]:
  shows  $a \# \{\$\$ \}$ 
  unfolding fresh_def supp_def
  by transfer simp

```

```

lemma fresh_fmap_update:
  shows  $\llbracket a \# f; a \# x; a \# y \rrbracket \implies a \# f(x\ \$\$ := y)$ 
  unfolding fresh_conv_MOST
  by (elim MOST_rev_mp) simp

```

```

lemma supp_fmempty[simp]:
  shows  $supp\ \{\$\$ \} = \{\}$ 
  by (simp add: supp_def)

```

```

lemma supp_fmap_update:
  shows  $supp\ (f(x\ \$\$ := y)) \subseteq supp(f, x, y)$ 
  using fresh_fmap_update

```

by (auto simp: fresh_def supp_Pair)

instance fmap :: (fs, fs) fs
proof
 fix x :: ('a, 'b) fmap
 show finite (supp x)
 by (induct x rule: fmap_induct)
 (simp_all add: supp_Pair finite_supp finite_subset[OF supp_fmap_update])
qed

lemma fresh_transfer[transfer_rule]:
 ((=) ==> pcr_fmap (=) (=) ==> (=)) fresh fresh
unfolding fresh_def supp_def rel_fun_def pcr_fmap_def cr_fmap_def simp_thms
 option.rel_eq fun_eq_iff[symmetric]
 by (auto elim!: finite_subset[rotated] simp: fmap_ext)

lemma fmmmap_eqvt[eqvt]: $p \cdot (\text{fmmmap } f \ F) = \text{fmmmap } (p \cdot f) (p \cdot F)$
 by (induct F arbitrary: f rule: fmap_induct) (auto simp add: fmap_update_eqvt fmmmap_fmupd)

lemma fmap_freshness_lemma:
 fixes h :: ('a::at,'b::pt) fmap
 assumes a: $\exists a. \text{atom } a \ \#\ (h, h \ \$\$ a)$
 shows $\exists x. \forall a. \text{atom } a \ \#\ h \longrightarrow h \ \$\$ a = x$
using assms **unfolding** fresh_Pair
 by transfer (simp add: fresh_Pair freshness_lemma)

lemma fmap_freshness_lemma_unique:
 fixes h :: ('a::at,'b::pt) fmap
 assumes $\exists a. \text{atom } a \ \#\ (h, h \ \$\$ a)$
 shows $\exists! x. \forall a. \text{atom } a \ \#\ h \longrightarrow h \ \$\$ a = x$
using assms **unfolding** fresh_Pair
 by transfer (rule freshness_lemma_unique, auto simp: fresh_Pair)

lemma fmdrop_fset_fmupd[simp]:
 (fmdrop_fset A f)(x \$\$= y) = fmdrop_fset (A |-| {|x|}) f(x \$\$= y)
including fmap.lifting and fset.lifting
 by transfer (auto simp: map_drop_set_def map_upd_def map_filter_def)

lemma fresh_fset_fminus:
 assumes atom x $\#\ A$
 shows $A \ |-| \{|x|\} = A$
using assms **by** (induct A) (simp_all add: finsert_fminus_if fresh_finsert)

lemma fresh_fun_app:
 shows atom x $\#\ F \Longrightarrow x \neq y \Longrightarrow F \ y = \text{Some } a \Longrightarrow \text{atom } x \ \#\ a$
using supp_fun_app[OF F y]
 by (auto simp: fresh_def supp_Some atom_not_fresh_eq)

lemma fresh_fmap_fresh_Some:
 atom x $\#\ F \Longrightarrow x \neq y \Longrightarrow F \ \$\$ y = \text{Some } a \Longrightarrow \text{atom } x \ \#\ a$
including fmap.lifting
 by (transfer) (auto elim: fresh_fun_app)

lemma fmdrop_eqvt: $p \cdot \text{fmdrop } x \ F = \text{fmdrop } (p \cdot x) (p \cdot F)$
 by transfer (auto simp: map_drop_def map_filter_def)

lemma fmfilter_eqvt: $p \cdot \text{fmfilter } Q \ F = \text{fmfilter } (p \cdot Q) (p \cdot F)$
 by transfer (auto simp: map_filter_def)

lemma *fmdrop_eq_iff*:
 $fmdrop\ x\ B = fmdrop\ y\ B \iff x = y \vee (x \notin fmdom'\ B \wedge y \notin fmdom'\ B)$
by *transfer* (*auto simp: map_drop_def map_filter_def fun_eq_iff, metis*)

lemma *fresh_fun_upd*:
shows $\llbracket a \# f; a \# x; a \# y \rrbracket \implies a \# f(x := y)$
unfolding *fresh_conv_MOST* **by** (*elim MOST_rev_mp*) *simp*

lemma *supp_fun_upd*:
shows $supp\ (f(x := y)) \subseteq supp(f, x, y)$
using *fresh_fun_upd* **by** (*auto simp: fresh_def supp_Pair*)

lemma *map_drop_fun_upd*: $map_drop\ x\ F = F(x := None)$
unfolding *map_drop_def map_filter_def* **by** *auto*

lemma *fresh_fmdrop_in_fmdom*: $\llbracket x \in fmdom'\ B; y \# B; y \# x \rrbracket \implies y \# fmdrop\ x\ B$
by *transfer* (*auto simp: map_drop_fun_upd fresh_None intro!: fresh_fun_upd*)

lemma *fresh_fmdrop*:
assumes $x \# B\ x \# y$
shows $x \# fmdrop\ y\ B$
using *assms* **by** (*cases* $y \in fmdom'\ B$) (*auto dest!: fresh_fmdrop_in_fmdom simp: fmdrop_idle'*)

lemma *fresh_fmdrop_fset*:
fixes $x :: atom$ **and** $A :: (_ :: at_base)\ fset$
assumes $x \# A\ x \# B$
shows $x \# fmdrop_fset\ A\ B$
using *assms(1)* **by** (*induct* A) (*auto simp: fresh_fmdrop assms(2) fresh_finsert*)

2 Syntax of λ

typedecl *hash*
instantiation *hash* **::** *pure*
begin
definition *permute_hash* **::** $perm \Rightarrow hash \Rightarrow hash$ **where**
 $permute_hash\ \pi\ h = h$
instance **proof** **qed** (*simp_all add: permute_hash_def*)
end

atom_decl *var*

nominal_datatype *term* =
Unit |
Var *var* |
Lam $x::var\ t::term$ **binds** x **in** t |
Rec $x::var\ t::term$ **binds** x **in** t |
Inj1 *term* |
Inj2 *term* |
Pair *term term* |
Let $term\ x::var\ t::term$ **binds** x **in** t |
App *term term* |
Case *term term term* |
Prj1 *term* |
Prj2 *term* |
Roll *term* |

Unroll term |
Auth term |
Unauth term |
Hash hash |
Hashed hash term

atom_decl *tvar*

nominal_datatype *ty* =

One |
Fun ty ty |
Sum ty ty |
Prod ty ty |
Mu $\alpha::tvar$ $\tau::ty$ binds α in τ |
Alpha tvar |
AuthT ty

lemma *no_tvars_in_term[simp]*: *atom* (*x* :: *tvar*) # (*t* :: *term*)
by (*induct t rule: term.induct*) *auto*

lemma *no_vars_in_ty[simp]*: *atom* (*x* :: *var*) # (τ :: *ty*)
by (*induct τ rule: ty.induct*) *auto*

inductive *value* :: *term* \Rightarrow *bool* **where**

value Unit |
value (Var _) |
value (Lam _ _) |
value (Rec _ _) |
value v \Rightarrow value (Inj1 v) |
value v \Rightarrow value (Inj2 v) |
 \llbracket *value v₁; value v₂* $\rrbracket \Rightarrow$ *value (Pair v₁ v₂)* |
value v \Rightarrow value (Roll v) |
value (Hash _) |
value v \Rightarrow value (Hashed _ v)

declare *value.intros[simp]*

declare *value.intros[intro]*

equivariance *value*

lemma *value_inv[simp]*:

\neg *value (Let e₁ x e₂)*
 \neg *value (App v v')*
 \neg *value (Case v v₁ v₂)*
 \neg *value (Prj1 v)*
 \neg *value (Prj2 v)*
 \neg *value (Unroll v)*
 \neg *value (Auth v)*
 \neg *value (Unauth v)*
using *value.cases* **by** *fastforce+*

inductive_cases *value_Inj1_inv[elim]*: *value (Inj1 e)*

inductive_cases *value_Inj2_inv[elim]*: *value (Inj2 e)*

inductive_cases *value_Pair_inv[elim]*: *value (Pair e₁ e₂)*

inductive_cases *value_Roll_inv[elim]*: *value (Roll e)*

inductive_cases *value_Hashed_inv[elim]*: *value (Hashed h e)*

abbreviation *closed* :: *term* \Rightarrow *bool* **where**

$closed\ t \equiv (\forall x::var. atom\ x \# t)$

3 Semantics of $\lambda\bullet$

Avoid clash with substitution notation.

no_notation *inverse_divide* (**infixl** \langle' / \rangle 70)

Help automated provers with smallsteps.

declare *One_nat_def*[*simp del*]

3.1 Equivariant Hash Function

consts *hash_real* :: *term* \Rightarrow *hash*

nominal_function *map_fixed* :: *var* \Rightarrow *var list* \Rightarrow *term* \Rightarrow *term* **where**

```

map_fixed fp l Unit = Unit |
map_fixed fp l (Var y) = (if y  $\in$  set l then (Var y) else (Var fp)) |
atom y # (fp, l)  $\implies$  map_fixed fp l (Lam y t) = (Lam y ((map_fixed fp (y # l) t))) |
atom y # (fp, l)  $\implies$  map_fixed fp l (Rec y t) = (Rec y ((map_fixed fp (y # l) t))) |
map_fixed fp l (Inj1 t) = (Inj1 ((map_fixed fp l t))) |
map_fixed fp l (Inj2 t) = (Inj2 ((map_fixed fp l t))) |
map_fixed fp l (Pair t1 t2) = (Pair ((map_fixed fp l t1)) ((map_fixed fp l t2))) |
map_fixed fp l (Roll t) = (Roll ((map_fixed fp l t))) |
atom y # (fp, l)  $\implies$  map_fixed fp l (Let t1 y t2) = (Let ((map_fixed fp l t1)) y ((map_fixed fp (y # l)
t2))) |
map_fixed fp l (App t1 t2) = (App ((map_fixed fp l t1)) ((map_fixed fp l t2))) |
map_fixed fp l (Case t1 t2 t3) = (Case ((map_fixed fp l t1)) ((map_fixed fp l t2)) ((map_fixed fp l
t3))) |
map_fixed fp l (Prj1 t) = (Prj1 ((map_fixed fp l t))) |
map_fixed fp l (Prj2 t) = (Prj2 ((map_fixed fp l t))) |
map_fixed fp l (Unroll t) = (Unroll ((map_fixed fp l t))) |
map_fixed fp l (Auth t) = (Auth ((map_fixed fp l t))) |
map_fixed fp l (Unauth t) = (Unauth ((map_fixed fp l t))) |
map_fixed fp l (Hash h) = (Hash h) |
map_fixed fp l (Hashed h t) = (Hashed h ((map_fixed fp l t)))
using [[simproc del: alpha_lst defined_all]]
subgoal by (simp add: eqvt_def map_fixed_graph_aux_def)
subgoal by (erule map_fixed_graph.induct) (auto simp: fresh_star_def fresh_at_base)
  apply clarify
subgoal for P fp l t
  by (rule term.strong_exhaust[of t P (fp, l)]) (auto simp: fresh_star_def fresh_Pair)
    apply (simp_all add: fresh_star_def fresh_at_base)
subgoal for y fp l t ya fpa la ta
  apply (erule conjE)+
  apply (erule Abs_lst1_fcb2'[where c = (fp, l)])
    apply (simp_all add: eqvt_at_def)
    apply (simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair)
  done
subgoal for y fp l t ya fpa la ta
  apply (erule conjE)+
  apply (erule Abs_lst1_fcb2'[where c = (fp, l)])
    apply (simp_all add: eqvt_at_def)
    apply (simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair)
  done
subgoal for y fp l t ya fpa la ta
  apply (erule conjE)+

```

```

apply (erule Abs_lst1_fcb2'[where c = (fp, l)])
  apply (simp_all add: eqvt_at_def)
  apply (simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair)
done
done
nominal_termination (eqvt)
  by lexicographic_order

definition hash where
  hash t = hash_real (map_fixed undefined [] t)

lemma permute_map_list: p · l = map (λx. p · x) l
  by (induct l) auto

lemma map_fixed_eqvt: p · l = l ⇒ map_fixed v l (p · t) = map_fixed v l t
proof (nominal_induct t avoiding: v l p rule: term.strong_induct)
  case (Var x)
  then show ?case
    by (auto simp: term.supp supp_at_base permute_map_list list_eq_iff_nth_eq in_set_conv_nth)
next
  case (Lam y e)
  from Lam(1,2,3,5) Lam(4)[of p y # l v] show ?case
    by (auto simp: fresh_perm)
next
  case (Rec y e)
  from Rec(1,2,3,5) Rec(4)[of p y # l v] show ?case
    by (auto simp: fresh_perm)
next
  case (Let e' y e)
  from Let(1,2,3,6) Let(4)[of p l v] Let(5)[of p y # l v] show ?case
    by (auto simp: fresh_perm)
qed (auto simp: permute_pure)

lemma hash_eqvt[eqvt]: p · hash t = hash (p · t)
  unfolding permute_pure hash_def by (auto simp: map_fixed_eqvt)

lemma map_fixed_idle: {x. ¬ atom x # t} ⊆ set l ⇒ map_fixed v l t = t
proof (nominal_induct t avoiding: v l rule: term.strong_induct)
  case (Var x)
  then show ?case
    by (auto simp: subset_iff fresh_at_base)
next
  case (Lam y e)
  from Lam(1,2,4) Lam(3)[of y # l v] show ?case
    by (auto simp: fresh_Pair Abs1_eq)
next
  case (Rec y e)
  from Rec(1,2,4) Rec(3)[of y # l v] show ?case
    by (auto simp: fresh_Pair Abs1_eq)
next
  case (Let e' y e)
  from Let(1,2,5) Let(3)[of l v] Let(4)[of y # l v] show ?case
    by (auto simp: fresh_Pair Abs1_eq)
qed (auto simp: subset_iff)

lemma map_fixed_idle_closed:
  closed t ⇒ map_fixed undefined [] t = t
  by (rule map_fixed_idle) auto

```

lemma *map_fixed_inj_closed*:
closed t \implies *closed u* \implies *map_fixed undefined [] t = map_fixed undefined [] u* \implies *t = u*
by (*rule* *box_equals*[*OF _ map_fixed_idle_closed map_fixed_idle_closed*])

lemma *hash_eq_hash_real_closed*:
assumes *closed t*
shows *hash t = hash_real t*
unfolding *hash_def map_fixed_idle_closed*[*OF assms*] ..

3.2 Substitution

nominal_function *subst_term* :: *term* \Rightarrow *term* \Rightarrow *var* \Rightarrow *term* ($\langle _ _ _ / _ \rangle$ [250, 200, 200] 250) **where**

Unit[*t' / x*] = *Unit* |
(*Var y*)[*t' / x*] = (*if x = y then t' else Var y*) |
atom y $\#$ (*x, t'*) \implies (*Lam y t*)[*t' / x*] = *Lam y (t[t' / x])* |
atom y $\#$ (*x, t'*) \implies (*Rec y t*)[*t' / x*] = *Rec y (t[t' / x])* |
(*Inj1 t*)[*t' / x*] = *Inj1 (t[t' / x])* |
(*Inj2 t*)[*t' / x*] = *Inj2 (t[t' / x])* |
(*Pair t1 t2*)[*t' / x*] = *Pair (t1[t' / x]) (t2[t' / x])* |
(*Roll t*)[*t' / x*] = *Roll (t[t' / x])* |
atom y $\#$ (*x, t'*) \implies (*Let t1 y t2*)[*t' / x*] = *Let (t1[t' / x]) y (t2[t' / x])* |
(*App t1 t2*)[*t' / x*] = *App (t1[t' / x]) (t2[t' / x])* |
(*Case t1 t2 t3*)[*t' / x*] = *Case (t1[t' / x]) (t2[t' / x]) (t3[t' / x])* |
(*Prj1 t*)[*t' / x*] = *Prj1 (t[t' / x])* |
(*Prj2 t*)[*t' / x*] = *Prj2 (t[t' / x])* |
(*Unroll t*)[*t' / x*] = *Unroll (t[t' / x])* |
(*Auth t*)[*t' / x*] = *Auth (t[t' / x])* |
(*Unauth t*)[*t' / x*] = *Unauth (t[t' / x])* |
(*Hash h*)[*t' / x*] = *Hash h* |
(*Hashed h t*)[*t' / x*] = *Hashed h (t[t' / x])*

using [*simp* *del: alpha_lst defined_all*]

subgoal by (*simp add: eqvt_def subst_term_graph_aux_def*)

subgoal by (*erule subst_term_graph.induct*) (*auto simp: fresh_star_def fresh_at_base*)

apply *clarify*

subgoal for *P a b t*

by (*rule term.strong_exhaust*[*of a P (b, t)*]) (*auto simp: fresh_star_def fresh_Pair*)

apply (*simp_all add: fresh_star_def fresh_at_base*)

subgoal

apply (*erule conjE*)

apply (*erule Abs_lst1_fcb2'*)

apply (*simp_all add: eqvt_at_def*)

apply (*simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair*)

done

subgoal

apply (*erule conjE*)

apply (*erule Abs_lst1_fcb2'*)

apply (*simp_all add: eqvt_at_def*)

apply (*simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair*)

done

subgoal

apply (*erule conjE*)

apply (*erule Abs_lst1_fcb2'*)

apply (*simp_all add: eqvt_at_def*)

apply (*simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair*)

done

done

nominal_termination (*eqvt*)

by *lexicographic_order*

type_synonym *tenv* = (*var*, *term*) *fmap*

nominal_function *psubst_term* :: *term* \Rightarrow *tenv* \Rightarrow *term* **where**

psubst_term *Unit* *f* = *Unit* |
psubst_term (*Var* *y*) *f* = (case *f* \$\$\$ *y* of *Some* *t* \Rightarrow *t* | *None* \Rightarrow *Var* *y*) |
atom *y* $\#$ *f* \Rightarrow *psubst_term* (*Lam* *y* *t*) *f* = *Lam* *y* (*psubst_term* *t* *f*) |
atom *y* $\#$ *f* \Rightarrow *psubst_term* (*Rec* *y* *t*) *f* = *Rec* *y* (*psubst_term* *t* *f*) |
psubst_term (*Inj1* *t*) *f* = *Inj1* (*psubst_term* *t* *f*) |
psubst_term (*Inj2* *t*) *f* = *Inj2* (*psubst_term* *t* *f*) |
psubst_term (*Pair* *t1* *t2*) *f* = *Pair* (*psubst_term* *t1* *f*) (*psubst_term* *t2* *f*) |
psubst_term (*Roll* *t*) *f* = *Roll* (*psubst_term* *t* *f*) |
atom *y* $\#$ *f* \Rightarrow *psubst_term* (*Let* *t1* *y* *t2*) *f* = *Let* (*psubst_term* *t1* *f*) *y* (*psubst_term* *t2* *f*) |
psubst_term (*App* *t1* *t2*) *f* = *App* (*psubst_term* *t1* *f*) (*psubst_term* *t2* *f*) |
psubst_term (*Case* *t1* *t2* *t3*) *f* = *Case* (*psubst_term* *t1* *f*) (*psubst_term* *t2* *f*) (*psubst_term* *t3* *f*) |
psubst_term (*Prj1* *t*) *f* = *Prj1* (*psubst_term* *t* *f*) |
psubst_term (*Prj2* *t*) *f* = *Prj2* (*psubst_term* *t* *f*) |
psubst_term (*Unroll* *t*) *f* = *Unroll* (*psubst_term* *t* *f*) |
psubst_term (*Auth* *t*) *f* = *Auth* (*psubst_term* *t* *f*) |
psubst_term (*Unauth* *t*) *f* = *Unauth* (*psubst_term* *t* *f*) |
psubst_term (*Hash* *h*) *f* = *Hash* *h* |
psubst_term (*Hashed* *h* *t*) *f* = *Hashed* *h* (*psubst_term* *t* *f*)

using [[*simproc* *del*: *alpha_lst* *defined_all*]]

subgoal by (*simp* *add*: *eqvt_def* *psubst_term_graph_aux_def*)

subgoal by (*erule* *psubst_term_graph.induct*) (*auto* *simp*: *fresh_star_def* *fresh_at_base*)

apply *clarify*

subgoal for *P* a *b*

by (*rule* *term.strong_exhaust*[of a *P* *b*]) (*auto* *simp*: *fresh_star_def* *fresh_Pair*)

apply (*simp_all* *add*: *fresh_star_def* *fresh_at_base*)

subgoal by *clarify*

subgoal

apply (*erule* *conjE*)

apply (*erule* *Abs_lst1_fcb2'*)

apply (*simp_all* *add*: *eqvt_at_def*)

apply (*simp_all* *add*: *perm_supp_eq* *Abs_fresh_iff*)

done

subgoal

apply (*erule* *conjE*)

apply (*erule* *Abs_lst1_fcb2'*)

apply (*simp_all* *add*: *eqvt_at_def*)

apply (*simp_all* *add*: *perm_supp_eq* *Abs_fresh_iff*)

done

subgoal

apply (*erule* *conjE*)

apply (*erule* *Abs_lst1_fcb2'*)

apply (*simp_all* *add*: *eqvt_at_def*)

apply (*simp_all* *add*: *perm_supp_eq* *Abs_fresh_iff*)

done

done

nominal_termination (*eqvt*)

by *lexicographic_order*

nominal_function *subst_type* :: *ty* \Rightarrow *ty* \Rightarrow *tvar* \Rightarrow *ty* **where**

subst_type *One* *t' x* = *One* |
subst_type (*Fun* *t1* *t2*) *t' x* = *Fun* (*subst_type* *t1* *t' x*) (*subst_type* *t2* *t' x*) |
subst_type (*Sum* *t1* *t2*) *t' x* = *Sum* (*subst_type* *t1* *t' x*) (*subst_type* *t2* *t' x*) |
subst_type (*Prod* *t1* *t2*) *t' x* = *Prod* (*subst_type* *t1* *t' x*) (*subst_type* *t2* *t' x*) |

```

atom y # (t', x) ==> subst_type (Mu y t) t' x = Mu y (subst_type t t' x) |
subst_type (Alpha y) t' x = (if y = x then t' else Alpha y) |
subst_type (AuthT t) t' x = AuthT (subst_type t t' x)
using [[simplproc del: alpha_lst defined_all]]
subgoal by (simp add: eqvt_def subst_type_graph_aux_def)
subgoal by (erule subst_type_graph.induct) (auto simp: fresh_star_def fresh_at_base)
apply clarify
subgoal for P a
  by (rule ty.strong_exhaust[of a P]) (auto simp: fresh_star_def)
    apply (simp_all add: fresh_star_def fresh_at_base)
subgoal
  apply (erule conjE)
  apply (erule Abs_lst1_fcb2')
    apply (simp_all add: eqvt_at_def)
    apply (simp_all add: perm_supp_eq Abs_fresh_iff fresh_Pair)
  done
done
nominal_termination (eqvt)
  by lexicographic_order

lemma fresh_subst_term: atom x # t[t' / x'] <=> (x = x' ∨ atom x # t) ∧ (atom x' # t ∨ atom x # t')
  by (nominal_induct t avoiding: t' x x' rule: term.strong_induct) (auto simp add: fresh_at_base)

lemma term_fresh_subst[simp]: atom x # t ==> atom x # s ==> (atom (x::var)) # t[s / y]
  by (nominal_induct t avoiding: s y rule: term.strong_induct) (auto)

lemma term_subst_idle[simp]: atom y # t ==> t[s / y] = t
  by (nominal_induct t avoiding: s y rule: term.strong_induct) (auto simp: fresh_Pair fresh_at_base)

lemma term_subst_subst: atom y1 ≠ atom y2 ==> atom y1 # s2 ==> t[s1 / y1][s2 / y2] = t[s2 / y2][s1[s2 / y2] / y1]
  by (nominal_induct t avoiding: y1 y2 s1 s2 rule: term.strong_induct) auto

lemma fresh_psubst:
  fixes x :: var
  assumes atom x # e atom x # vs
  shows atom x # psubst_term e vs
  using assms
  by (induct e vs rule: psubst_term.induct)
    (auto simp: fresh_at_base elim: fresh_fmap_fresh_Some split: option.splits)

lemma fresh_subst_type:
  atom α # subst_type τ τ' α' <=> ((α = α' ∨ atom α # τ) ∧ (atom α' # τ ∨ atom α # τ'))
  by (nominal_induct τ avoiding: α α' τ' rule: ty.strong_induct) (auto simp add: fresh_at_base)

lemma type_fresh_subst[simp]: atom x # t ==> atom x # s ==> (atom (x::tvar)) # subst_type t s y
  by (nominal_induct t avoiding: s y rule: ty.strong_induct) (auto)

lemma type_subst_idle[simp]: atom y # t ==> subst_type t s y = t
  by (nominal_induct t avoiding: s y rule: ty.strong_induct) (auto simp: fresh_Pair fresh_at_base)

lemma type_subst_subst: atom y1 ≠ atom y2 ==> atom y1 # s2 ==>
  subst_type (subst_type t s1 y1) s2 y2 = subst_type (subst_type t s2 y2) (subst_type s1 s2 y2) y1
  by (nominal_induct t avoiding: y1 y2 s1 s2 rule: ty.strong_induct) auto

```

3.3 Weak Typing Judgement

`type_synonym tyenv = (var, ty) fmap`

inductive *judge_weak* :: *tyenv* \Rightarrow *term* \Rightarrow *ty* \Rightarrow *bool* ($\langle _ \vdash_W _ : _ \rangle$ [150,0,150] 149) **where**

jw_Unit: $\Gamma \vdash_W \text{Unit} : \text{One} \mid$
jw_Var: $\llbracket \Gamma \ \$\$ x = \text{Some } \tau \rrbracket$
 $\implies \Gamma \vdash_W \text{Var } x : \tau \mid$
jw_Lam: $\llbracket \text{atom } x \ \#\ \Gamma; \Gamma(x \ \$\$:= \tau_1) \vdash_W e : \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Lam } x e : \text{Fun } \tau_1 \ \tau_2 \mid$
jw_App: $\llbracket \Gamma \vdash_W e : \text{Fun } \tau_1 \ \tau_2; \Gamma \vdash_W e' : \tau_1 \rrbracket$
 $\implies \Gamma \vdash_W \text{App } e e' : \tau_2 \mid$
jw_Let: $\llbracket \text{atom } x \ \#\ (\Gamma, e_1); \Gamma \vdash_W e_1 : \tau_1; \Gamma(x \ \$\$:= \tau_1) \vdash_W e_2 : \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Let } e_1 x e_2 : \tau_2 \mid$
jw_Rec: $\llbracket \text{atom } x \ \#\ \Gamma; \text{atom } y \ \#\ (\Gamma, x); \Gamma(x \ \$\$:= \text{Fun } \tau_1 \ \tau_2) \vdash_W \text{Lam } y e : \text{Fun } \tau_1 \ \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Rec } x (\text{Lam } y e) : \text{Fun } \tau_1 \ \tau_2 \mid$
jw_Inj1: $\llbracket \Gamma \vdash_W e : \tau_1 \rrbracket$
 $\implies \Gamma \vdash_W \text{Inj1 } e : \text{Sum } \tau_1 \ \tau_2 \mid$
jw_Inj2: $\llbracket \Gamma \vdash_W e : \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Inj2 } e : \text{Sum } \tau_1 \ \tau_2 \mid$
jw_Case: $\llbracket \Gamma \vdash_W e : \text{Sum } \tau_1 \ \tau_2; \Gamma \vdash_W e_1 : \text{Fun } \tau_1 \ \tau; \Gamma \vdash_W e_2 : \text{Fun } \tau_2 \ \tau \rrbracket$
 $\implies \Gamma \vdash_W \text{Case } e e_1 e_2 : \tau \mid$
jw_Pair: $\llbracket \Gamma \vdash_W e_1 : \tau_1; \Gamma \vdash_W e_2 : \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Pair } e_1 e_2 : \text{Prod } \tau_1 \ \tau_2 \mid$
jw_Prj1: $\llbracket \Gamma \vdash_W e : \text{Prod } \tau_1 \ \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Prj1 } e : \tau_1 \mid$
jw_Prj2: $\llbracket \Gamma \vdash_W e : \text{Prod } \tau_1 \ \tau_2 \rrbracket$
 $\implies \Gamma \vdash_W \text{Prj2 } e : \tau_2 \mid$
jw_Roll: $\llbracket \text{atom } \alpha \ \#\ \Gamma; \Gamma \vdash_W e : \text{subst_type } \tau (\text{Mu } \alpha \ \tau) \ \alpha \rrbracket$
 $\implies \Gamma \vdash_W \text{Roll } e : \text{Mu } \alpha \ \tau \mid$
jw_Unroll: $\llbracket \text{atom } \alpha \ \#\ \Gamma; \Gamma \vdash_W e : \text{Mu } \alpha \ \tau \rrbracket$
 $\implies \Gamma \vdash_W \text{Unroll } e : \text{subst_type } \tau (\text{Mu } \alpha \ \tau) \ \alpha \mid$
jw_Auth: $\llbracket \Gamma \vdash_W e : \tau \rrbracket$
 $\implies \Gamma \vdash_W \text{Auth } e : \tau \mid$
jw_Unauth: $\llbracket \Gamma \vdash_W e : \tau \rrbracket$
 $\implies \Gamma \vdash_W \text{Unauth } e : \tau$

declare *judge_weak.intros*[*simp*]

declare *judge_weak.intros*[*intro*]

equivariance *judge_weak*

nominal_inductive *judge_weak*

avoids *jw_Lam*: *x*

| *jw_Rec*: *x* **and** *y*

| *jw_Let*: *x*

| *jw_Roll*: α

| *jw_Unroll*: α

by (*auto simp*: *fresh_subst_type fresh_Pair*)

Inversion rules for typing judgment.

inductive_cases *jw_Unit_inv*[*elim*]: $\Gamma \vdash_W \text{Unit} : \tau$

inductive_cases *jw_Var_inv*[*elim*]: $\Gamma \vdash_W \text{Var } x : \tau$

lemma *jw_Lam_inv*[*elim*]:

assumes $\Gamma \vdash_W \text{Lam } x e : \tau$

and $\text{atom } x \ \#\ \Gamma$

obtains $\tau_1 \ \tau_2$ **where** $\tau = \text{Fun } \tau_1 \ \tau_2 (\Gamma(x \ \$\$:= \tau_1)) \vdash_W e : \tau_2$

using *assms* **proof** (*atomize_elim*, *nominal_induct* $\Gamma \text{Lam } x e \ \tau$ *avoiding*: *x e* *rule*: *judge_weak.strong_induct*)

case (*jw_Lam* $x \ \Gamma \ \tau_1 \ t \ \tau_2 \ y \ u$)

then show *?case*

by (*auto simp*: *perm_supp_eq_elim*!:

iffD1[*OF Abs_lst1_fcb2'*[**where** $f = \lambda x t (\Gamma, \tau_1, \tau_2). (\Gamma(x \text{ \&\$:= } \tau_1)) \vdash_W t : \tau_2$
and $c = (\Gamma, \tau_1, \tau_2)$ **and** $a = x$ **and** $b = y$ **and** $x = t$ **and** $y = u$, *unfolded prod.case*],
rotated -1)]

qed

lemma *swap_permute_swap*: $\text{atom } x \# \pi \implies \text{atom } y \# \pi \implies (x \leftrightarrow y) \cdot \pi \cdot (x \leftrightarrow y) \cdot t = \pi \cdot t$
by (*subst permute_eqvt*) (*auto simp: flip_fresh_fresh*)

lemma *jw_Rec_inv[elim]*:

assumes $\Gamma \vdash_W \text{Rec } x t : \tau$

and $\text{atom } x \# \Gamma$

obtains $y e \tau_1 \tau_2$ **where** $\text{atom } y \# (\Gamma, x) t = \text{Lam } y e \tau = \text{Fun } \tau_1 \tau_2 \Gamma(x \text{ \&\$:= } \text{Fun } \tau_1 \tau_2) \vdash_W \text{Lam } y$
 $e : \text{Fun } \tau_1 \tau_2$

using [*simproc del: alpha_lst*] *assms*

proof (*atomize_elim*, *nominal_induct* $\Gamma \text{Rec } x t \tau$ *avoiding: x t rule: judge_weak.strong_induct*)

case (*jw_Rec* $x \Gamma y \tau_1 \tau_2 e z t$)

then show *?case*

proof (*nominal_induct* t *avoiding: y x z rule: term.strong_induct*)

case (*Lam* $y' e'$)

show *?case*

proof (*intro exI conjI*)

from *Lam.prem*s **show** $\text{atom } y \# (\Gamma, z)$ **by** *simp*

from *Lam.hyps*(1-3) *Lam.prem*s **show** $\text{Lam } y' e' = \text{Lam } y ((y' \leftrightarrow y) \cdot e')$

by (*subst term.eq_iff*(3), *intro Abs_lst_eq_flipI*) (*simp add: fresh_at_base*)

from *Lam.hyps*(1-3) *Lam.prem*s **show** $\Gamma(z \text{ \&\$:= } \text{Fun } \tau_1 \tau_2) \vdash_W \text{Lam } y ((y' \leftrightarrow y) \cdot e') : \text{Fun } \tau_1 \tau_2$

by (*elim judge_weak_eqvt*[*of* $\Gamma(x \text{ \&\$:= } \text{Fun } \tau_1 \tau_2) \text{Lam } y e \text{Fun } \tau_1 \tau_2 (x \leftrightarrow z)$, *elim_format*])

(*simp add: perm_supp_eq Abs1_eq_iff fresh_at_base swap_permute_swap fresh_perm_flip_commute*)

qed *simp*

qed (*simp_all add: Abs1_eq_iff*)

qed

inductive_cases *jw_Inj1_inv[elim]*: $\Gamma \vdash_W \text{Inj1 } e : \tau$

inductive_cases *jw_Inj2_inv[elim]*: $\Gamma \vdash_W \text{Inj2 } e : \tau$

inductive_cases *jw_Pair_inv[elim]*: $\Gamma \vdash_W \text{Pair } e_1 e_2 : \tau$

lemma *jw_Let_inv[elim]*:

assumes $\Gamma \vdash_W \text{Let } e_1 x e_2 : \tau_2$

and $\text{atom } x \# (e_1, \Gamma)$

obtains τ_1 **where** $\Gamma \vdash_W e_1 : \tau_1 \Gamma(x \text{ \&\$:= } \tau_1) \vdash_W e_2 : \tau_2$

using *assms* **proof** (*atomize_elim*, *nominal_induct* $\Gamma \text{Let } e_1 x e_2 \tau_2$ *avoiding: e_1 x e_2 rule: judge_weak.strong_induct*)

case (*jw_Let* $x \Gamma e_1 \tau_1 e_2 \tau_2 x' e_2'$)

then show *?case*

by (*auto simp: fresh_Pair perm_supp_eq elim!*:

iffD1[*OF Abs_lst1_fcb2'*[**where** $f = \lambda x t (\Gamma, \tau_1, \tau_2). \Gamma(x \text{ \&\$:= } \tau_1) \vdash_W t : \tau_2$

and $c = (\Gamma, \tau_1, \tau_2)$ **and** $a = x$ **and** $b = x'$ **and** $x = e_2$ **and** $y = e_2'$, *unfolded prod.case*],

rotated -1)]

qed

inductive_cases *jw_Prj1_inv[elim]*: $\Gamma \vdash_W \text{Prj1 } e : \tau_1$

inductive_cases *jw_Prj2_inv[elim]*: $\Gamma \vdash_W \text{Prj2 } e : \tau_2$

inductive_cases *jw_App_inv[elim]*: $\Gamma \vdash_W \text{App } e e' : \tau_2$

inductive_cases *jw_Case_inv[elim]*: $\Gamma \vdash_W \text{Case } e e_1 e_2 : \tau$

inductive_cases *jw_Auth_inv[elim]*: $\Gamma \vdash_W \text{Auth } e : \tau$

inductive_cases *jw_Unauth_inv[elim]*: $\Gamma \vdash_W \text{Unauth } e : \tau$

lemma *subst_type_perm_eq*:

assumes $\text{atom } b \# t$

shows $\text{subst_type } t (\text{Mu } a t) a = \text{subst_type } ((a \leftrightarrow b) \cdot t) (\text{Mu } b ((a \leftrightarrow b) \cdot t)) b$

```

using assms proof –
have f: atom a # subst_type t (Mu a t) a by (rule iffD2[OF fresh_subst_type]) simp
have atom b # subst_type t (Mu a t) a by (auto simp: assms)
with f have subst_type t (Mu a t) a = (a ↔ b) • subst_type t (Mu a t) a
  by (simp add: flip_fresh_fresh)
then show subst_type t (Mu a t) a = subst_type ((a ↔ b) • t) (Mu b ((a ↔ b) • t)) b
  by simp
qed

lemma jw_Roll_inv[elim]:
assumes  $\Gamma \vdash_W \text{Roll } e : \tau$ 
and atom  $\alpha$  # ( $\Gamma, \tau$ )
obtains  $\tau'$  where  $\tau = \text{Mu } \alpha \tau' \Gamma \vdash_W e : \text{subst\_type } \tau' (\text{Mu } \alpha \tau') \alpha$ 
using assms [[simplproc del: alpha_lst]]
proof (atomize_elim, nominal_induct  $\Gamma \text{Roll } e \tau$  avoiding: e  $\alpha$  rule: judge_weak.strong_induct)
case (jw_Roll  $\alpha \Gamma e \tau \alpha'$ )
then show ?case
  by (auto simp: perm_supp_eq fresh_Pair fresh_at_base subst_type.eqvt)
  intro!: exI[of  $\_ (\alpha \leftrightarrow \alpha') \cdot \tau$ ] Abs_lst_eq_flipI dest: judge_weak.eqvt[of  $\_ \_ \_ (\alpha \leftrightarrow \alpha')$ ]]
qed

lemma jw_Unroll_inv[elim]:
assumes  $\Gamma \vdash_W \text{Unroll } e : \tau$ 
and atom  $\alpha$  # ( $\Gamma, \tau$ )
obtains  $\tau'$  where  $\tau = \text{subst\_type } \tau' (\text{Mu } \alpha \tau') \alpha \Gamma \vdash_W e : \text{Mu } \alpha \tau'$ 
using assms proof (atomize_elim, nominal_induct  $\Gamma \text{Unroll } e \tau$  avoiding: e  $\alpha$  rule: judge_weak.strong_induct)
case (jw_Unroll  $\alpha \Gamma e \tau \alpha'$ )
then show ?case
  by (auto simp: perm_supp_eq fresh_Pair subst_type_perm_eq fresh_subst_type)
  intro!: exI[of  $\_ (\alpha \leftrightarrow \alpha') \cdot \tau$ ] dest: judge_weak.eqvt[of  $\_ \_ \_ (\alpha \leftrightarrow \alpha')$ ]]
qed

Additional inversion rules based on type rather than term.

inductive_cases jw_Prod_inv[elim]: { $\$\$$ }  $\vdash_W e : \text{Prod } \tau_1 \tau_2$ 
inductive_cases jw_Sum_inv[elim]: { $\$\$$ }  $\vdash_W e : \text{Sum } \tau_1 \tau_2$ 

lemma jw_Fun_inv[elim]:
assumes { $\$\$$ }  $\vdash_W v : \text{Fun } \tau_1 \tau_2 \text{ value } v$ 
obtains e x where  $v = \text{Lam } x e \vee v = \text{Rec } x e \text{ atom } x \# (c::\text{term})$ 
using assms [[simplproc del: alpha_lst]]
proof (atomize_elim, nominal_induct { $\$\$$ } :: tyenv v Fun  $\tau_1 \tau_2$  avoiding:  $\tau_1 \tau_2$  rule: judge_weak.strong_induct)
case (jw_Lam x  $\tau_1 e \tau_2$ )
then obtain x' where atom (x'::var) # (c, e) using finite_supp obtain_fresh' by blast
then have [[atom x]]lst. e = [[atom x']]lst. (x ↔ x') • e ∧ atom x' # c
  by (simp add: Abs_lst_eq_flipI fresh_Pair)
then show ?case
  by auto
next
case (jw_Rec x y  $\tau_1 \tau_2 e'$ )
obtain x' where atom (x'::var) # (c, Lam y e') using finite_supp obtain_fresh by blast
then have [[atom x]]lst. Lam y e' = [[atom x']]lst. (x ↔ x') • (Lam y e') ∧ atom x' # c
  using Abs_lst_eq_flipI fresh_Pair by blast
then show ?case
  by auto
qed simp_all

lemma jw_Mu_inv[elim]:
assumes { $\$\$$ }  $\vdash_W v : \text{Mu } \alpha \tau \text{ value } v$ 

```

obtains v' where $v = \text{Roll } v'$
using *assms* by (*atomize_elim*, *nominal_induct* { $\$\$$ } :: *tyenv* v $\text{Mu } \alpha \tau$ rule: *judge_weak.strong_induct*)
simp_all

3.4 Erasure of Authenticated Types

nominal_function *erase* :: *ty* \Rightarrow *ty* where
erase *One* = *One* |
erase (*Fun* $\tau_1 \tau_2$) = *Fun* (*erase* τ_1) (*erase* τ_2) |
erase (*Sum* $\tau_1 \tau_2$) = *Sum* (*erase* τ_1) (*erase* τ_2) |
erase (*Prod* $\tau_1 \tau_2$) = *Prod* (*erase* τ_1) (*erase* τ_2) |
erase (*Mu* $\alpha \tau$) = *Mu* α (*erase* τ) |
erase (*Alpha* α) = *Alpha* α |
erase (*AuthT* τ) = *erase* τ
using [*simplproc* del: *alpha_lst*]
subgoal by (*simp* add: *eqvt_def* *erase_graph_aux_def*)
subgoal by (*erule* *erase_graph.induct*) (*auto* *simp*: *fresh_star_def* *fresh_at_base*)
subgoal for $P\ x$
 by (*rule* *ty.strong_exhaust*[of $x\ P\ x$] (*auto* *simp*: *fresh_star_def*)
 apply (*simp_all* add: *fresh_star_def* *fresh_at_base*)
subgoal
 apply (*erule* *Abs_lst1_fcb2'*)
 apply (*simp_all* add: *eqvt_at_def*)
 apply (*simp_all* add: *perm_supp_eq* *Abs_fresh_iff*)
 done
done
nominal_termination (*eqvt*)
 by *lexicographic_order*

lemma *fresh_erase_fresh*:
assumes *atom* $x \# \tau$
shows *atom* $x \# \text{erase } \tau$
using *assms* by (*induct* τ rule: *ty.induct*) *auto*

lemma *fresh_fmmap_erase_fresh*:
assumes *atom* $x \# \Gamma$
shows *atom* $x \# \text{fmmap } \text{erase } \Gamma$
using *assms* by *transfer* *simp*

lemma *erase_subst_type_shift*[*simp*]:
erase (*subst_type* $\tau \tau' \alpha$) = *subst_type* (*erase* τ) (*erase* τ') α
by (*induct* $\tau \tau' \alpha$ rule: *subst_type.induct*) (*auto* *simp*: *fresh_Pair* *fresh_erase_fresh*)

definition *erase_env* :: *tyenv* \Rightarrow *tyenv* where
erase_env = *fmmap* *erase*

3.5 Strong Typing Judgement

inductive *judge* :: *tyenv* \Rightarrow *term* \Rightarrow *ty* \Rightarrow *bool* ($\langle _ \vdash _ : _ \rangle$ [150,0,150] 149) where
j_Unit: $\Gamma \vdash \text{Unit} : \text{One}$ |
j_Var: $\llbracket \Gamma \ \$\$ \ x = \text{Some } \tau \rrbracket$
 $\Longrightarrow \Gamma \vdash \text{Var } x : \tau$ |
j_Lam: $\llbracket \text{atom } x \# \Gamma; \Gamma(x \ \$\$:= \tau_1) \vdash e : \tau_2 \rrbracket$
 $\Longrightarrow \Gamma \vdash \text{Lam } x\ e : \text{Fun } \tau_1 \tau_2$ |
j_App: $\llbracket \Gamma \vdash e : \text{Fun } \tau_1 \tau_2; \Gamma \vdash e' : \tau_1 \rrbracket$
 $\Longrightarrow \Gamma \vdash \text{App } e\ e' : \tau_2$ |
j_Let: $\llbracket \text{atom } x \# (\Gamma, e_1); \Gamma \vdash e_1 : \tau_1; \Gamma(x \ \$\$:= \tau_1) \vdash e_2 : \tau_2 \rrbracket$
 $\Longrightarrow \Gamma \vdash \text{Let } e_1\ x\ e_2 : \tau_2$ |
j_Rec: $\llbracket \text{atom } x \# \Gamma; \text{atom } y \# (\Gamma, x); \Gamma(x \ \$\$:= \text{Fun } \tau_1 \tau_2) \vdash \text{Lam } y\ e' : \text{Fun } \tau_1 \tau_2 \rrbracket$

$$\begin{aligned} & \Rightarrow \Gamma \vdash \text{Rec } x \text{ (Lam } y \text{ } e') : \text{Fun } \tau_1 \tau_2 \mid \\ j_Inj1: & \llbracket \Gamma \vdash e : \tau_1 \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Inj1 } e : \text{Sum } \tau_1 \tau_2 \mid \\ j_Inj2: & \llbracket \Gamma \vdash e : \tau_2 \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Inj2 } e : \text{Sum } \tau_1 \tau_2 \mid \\ j_Case: & \llbracket \Gamma \vdash e : \text{Sum } \tau_1 \tau_2; \Gamma \vdash e_1 : \text{Fun } \tau_1 \tau; \Gamma \vdash e_2 : \text{Fun } \tau_2 \tau \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Case } e \ e_1 \ e_2 : \tau \mid \\ j_Pair: & \llbracket \Gamma \vdash e_1 : \tau_1; \Gamma \vdash e_2 : \tau_2 \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Pair } e_1 \ e_2 : \text{Prod } \tau_1 \tau_2 \mid \\ j_Prj1: & \llbracket \Gamma \vdash e : \text{Prod } \tau_1 \tau_2 \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Prj1 } e : \tau_1 \mid \\ j_Prj2: & \llbracket \Gamma \vdash e : \text{Prod } \tau_1 \tau_2 \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Prj2 } e : \tau_2 \mid \\ j_Roll: & \llbracket \text{atom } \alpha \ \sharp \ \Gamma; \Gamma \vdash e : \text{subst_type } \tau \text{ (Mu } \alpha \ \tau) \ \alpha \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Roll } e : \text{Mu } \alpha \ \tau \mid \\ j_Unroll: & \llbracket \text{atom } \alpha \ \sharp \ \Gamma; \Gamma \vdash e : \text{Mu } \alpha \ \tau \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Unroll } e : \text{subst_type } \tau \text{ (Mu } \alpha \ \tau) \ \alpha \mid \\ j_Auth: & \llbracket \Gamma \vdash e : \tau \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Auth } e : \text{AuthT } \tau \mid \\ j_Unauth: & \llbracket \Gamma \vdash e : \text{AuthT } \tau \rrbracket \\ & \Rightarrow \Gamma \vdash \text{Unauth } e : \tau \end{aligned}$$

declare *judge.intros*[*intro*]

equivariance *judge*

nominal_inductive *judge*

avoids *j_Lam*: *x*

| *j_Rec*: *x* **and** *y*

| *j_Let*: *x*

| *j_Roll*: α

| *j_Unroll*: α

by (*auto simp*: *fresh_subst_type fresh_Pair*)

lemma *judge_imp_judge_weak*:

assumes $\Gamma \vdash e : \tau$

shows *erase_env* $\Gamma \vdash_W e : \text{erase } \tau$

using *assms unfolding erase_env_def*

by (*induct* $\Gamma \ e \ \tau$ *rule*: *judge.induct*) (*simp_all add*: *fresh_Pair fresh_fmmap_erase_fresh_fmmap_fmupd*)

3.6 Shallow Projection

nominal_function *shallow* :: *term* \Rightarrow *term* ($\langle \langle _ \rangle \rangle$) **where**

$\langle \langle \text{Unit} \rangle \rangle = \text{Unit} \mid$

$\langle \langle \text{Var } v \rangle \rangle = \text{Var } v \mid$

$\langle \langle \text{Lam } x \ e \rangle \rangle = \text{Lam } x \ \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Rec } x \ e \rangle \rangle = \text{Rec } x \ \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Inj1 } e \rangle \rangle = \text{Inj1 } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Inj2 } e \rangle \rangle = \text{Inj2 } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Pair } e_1 \ e_2 \rangle \rangle = \text{Pair } \langle \langle e_1 \rangle \rangle \ \langle \langle e_2 \rangle \rangle \mid$

$\langle \langle \text{Roll } e \rangle \rangle = \text{Roll } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Let } e_1 \ x \ e_2 \rangle \rangle = \text{Let } \langle \langle e_1 \rangle \rangle \ x \ \langle \langle e_2 \rangle \rangle \mid$

$\langle \langle \text{App } e_1 \ e_2 \rangle \rangle = \text{App } \langle \langle e_1 \rangle \rangle \ \langle \langle e_2 \rangle \rangle \mid$

$\langle \langle \text{Case } e \ e_1 \ e_2 \rangle \rangle = \text{Case } \langle \langle e \rangle \rangle \ \langle \langle e_1 \rangle \rangle \ \langle \langle e_2 \rangle \rangle \mid$

$\langle \langle \text{Prj1 } e \rangle \rangle = \text{Prj1 } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Prj2 } e \rangle \rangle = \text{Prj2 } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Unroll } e \rangle \rangle = \text{Unroll } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Auth } e \rangle \rangle = \text{Auth } \langle \langle e \rangle \rangle \mid$

$\langle \langle \text{Unauth } e \rangle \rangle = \text{Unauth } \langle \langle e \rangle \rangle \mid$

— No rule is defined for Hash, but: "[...] preserving that structure in every case but that of $\langle h, v \rangle$ [...]"

```

( $\!Hash\ h$ ) = Hash h |
( $\!Hashed\ h\ e$ ) = Hash h
using [ $\!simproc\ del:\ alpha\_lst$ ]
subgoal by ( $\!simp\ add:\ eqvt\_def\ shallow\_graph\_aux\_def$ )
subgoal by ( $\!erule\ shallow\_graph.induct$ ) ( $\!auto\ simp:\ fresh\_star\_def\ fresh\_at\_base$ )
subgoal for  $P\ a$ 
  by ( $\!rule\ term.strong\_exhaust$ [of a  $P\ a$ ]) ( $\!auto\ simp:\ fresh\_star\_def$ )
    apply ( $\!simp\_all\ add:\ fresh\_star\_def\ fresh\_at\_base$ )
subgoal
  apply ( $\!erule\ Abs\_lst1\_fcb2'$ )
    apply ( $\!simp\_all\ add:\ eqvt\_at\_def$ )
    apply ( $\!simp\_all\ add:\ perm\_supp\_eq\ Abs\_fresh\_iff$ )
  done
subgoal
  apply ( $\!erule\ Abs\_lst1\_fcb2'$ )
    apply ( $\!simp\_all\ add:\ eqvt\_at\_def$ )
    apply ( $\!simp\_all\ add:\ perm\_supp\_eq\ Abs\_fresh\_iff$ )
  done
subgoal
  apply ( $\!erule\ Abs\_lst1\_fcb2'$ )
    apply ( $\!simp\_all\ add:\ eqvt\_at\_def$ )
    apply ( $\!simp\_all\ add:\ perm\_supp\_eq\ Abs\_fresh\_iff$ )
  done
done
nominal\_termination ( $\!eqvt$ )
  by  $\!lexicographic\_order$ 

```

lemma $\!fresh_shallow$: $atom\ x\ \# \ e \implies atom\ x\ \# \ (e)$
by ($\!induct\ e\ rule:\ term.induct$) $\!auto$

3.7 Small-step Semantics

datatype $\!mode = I \mid P \mid V$ — Ideal, Prover and Verifier modes

instantiation $\!mode :: pure$

begin

definition $\!permute_mode :: perm \Rightarrow mode \Rightarrow mode$ **where**

$\!permute_mode\ \pi\ h = h$

instance proof qed ($\!auto\ simp:\ permute_mode_def$)

end

type_synonym $\!proofstream = term\ list$

inductive $\!smallstep :: proofstream \Rightarrow term \Rightarrow mode \Rightarrow proofstream \Rightarrow term \Rightarrow bool$ ($\!<< _ >> _ \rightarrow << _ >> _ >>$) **where**

$\!s_App1$: $\ll \pi, e_1 \gg \ m \rightarrow \ll \pi', e_1' \gg$]

$\implies \ll \pi, App\ e_1\ e_2 \gg \ m \rightarrow \ll \pi', App\ e_1'\ e_2 \gg$ |

$\!s_App2$: $\ll value\ v_1; \ll \pi, e_2 \gg \ m \rightarrow \ll \pi', e_2' \gg$]

$\implies \ll \pi, App\ v_1\ e_2 \gg \ m \rightarrow \ll \pi', App\ v_1\ e_2' \gg$ |

$\!s_AppLam$: $\ll value\ v; atom\ x\ \# \ (v, \pi)$]

$\implies \ll \pi, App\ (Lam\ x\ e)\ v \gg \ _ \rightarrow \ll \pi, e[v / x] \gg$ |

$\!s_AppRec$: $\ll value\ v; atom\ x\ \# \ (v, \pi)$]

$\implies \ll \pi, App\ (Rec\ x\ e)\ v \gg \ _ \rightarrow \ll \pi, App\ (e[(Rec\ x\ e) / x])\ v \gg$ |

$\!s_Let1$: $\ll atom\ x\ \# \ (e_1, e_1', \pi, \pi'); \ll \pi, e_1 \gg \ m \rightarrow \ll \pi', e_1' \gg$]

$\implies \ll \pi, Let\ e_1\ x\ e_2 \gg \ m \rightarrow \ll \pi', Let\ e_1'\ x\ e_2 \gg$ |

$\!s_Let2$: $\ll value\ v; atom\ x\ \# \ (v, \pi)$]

$\implies \ll \pi, Let\ v\ x\ e \gg \ _ \rightarrow \ll \pi, e[v / x] \gg$ |

$s_Inj1: \quad [\ll \pi, e \gg m \rightarrow \ll \pi', e' \gg]$
 $\implies \ll \pi, Inj1\ e \gg m \rightarrow \ll \pi', Inj1\ e' \gg \mid$
 $s_Inj2: \quad [\ll \pi, e \gg m \rightarrow \ll \pi', e' \gg]$
 $\implies \ll \pi, Inj2\ e \gg m \rightarrow \ll \pi', Inj2\ e' \gg \mid$
 $s_Case: \quad [\ll \pi, e \gg m \rightarrow \ll \pi', e' \gg]$
 $\implies \ll \pi, Case\ e\ e_1\ e_2 \gg m \rightarrow \ll \pi', Case\ e'\ e_1\ e_2 \gg \mid$
— Case rules are different from paper to account for recursive functions.
 $s_CaseInj1: [\text{value } v]$
 $\implies \ll \pi, Case\ (Inj1\ v)\ e_1\ e_2 \gg _ \rightarrow \ll \pi, App\ e_1\ v \gg \mid$
 $s_CaseInj2: [\text{value } v]$
 $\implies \ll \pi, Case\ (Inj2\ v)\ e_1\ e_2 \gg _ \rightarrow \ll \pi, App\ e_2\ v \gg \mid$
 $s_Pair1: \quad [\ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg]$
 $\implies \ll \pi, Pair\ e_1\ e_2 \gg m \rightarrow \ll \pi', Pair\ e_1'\ e_2' \gg \mid$
 $s_Pair2: \quad [\text{value } v_1; \ll \pi, e_2 \gg m \rightarrow \ll \pi', e_2' \gg]$
 $\implies \ll \pi, Pair\ v_1\ e_2 \gg m \rightarrow \ll \pi', Pair\ v_1\ e_2' \gg \mid$
 $s_Prj1: \quad [\ll \pi, e \gg m \rightarrow \ll \pi', e' \gg]$
 $\implies \ll \pi, Prj1\ e \gg m \rightarrow \ll \pi', Prj1\ e' \gg \mid$
 $s_Prj2: \quad [\ll \pi, e \gg m \rightarrow \ll \pi', e' \gg]$
 $\implies \ll \pi, Prj2\ e \gg m \rightarrow \ll \pi', Prj2\ e' \gg \mid$
 $s_PrjPair1: [\text{value } v_1; \text{value } v_2]$
 $\implies \ll \pi, Prj1\ (Pair\ v_1\ v_2) \gg _ \rightarrow \ll \pi, v_1 \gg \mid$
 $s_PrjPair2: [\text{value } v_1; \text{value } v_2]$
 $\implies \ll \pi, Prj2\ (Pair\ v_1\ v_2) \gg _ \rightarrow \ll \pi, v_2 \gg \mid$
 $s_Unroll: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\implies \ll \pi, Unroll\ e \gg m \rightarrow \ll \pi', Unroll\ e' \gg \mid$
 $s_Roll: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\implies \ll \pi, Roll\ e \gg m \rightarrow \ll \pi', Roll\ e' \gg \mid$
 $s_UnrollRoll: [\text{value } v]$
 $\implies \ll \pi, Unroll\ (Roll\ v) \gg _ \rightarrow \ll \pi, v \gg \mid$
— Mode-specific rules
 $s_Auth: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\implies \ll \pi, Auth\ e \gg m \rightarrow \ll \pi', Auth\ e' \gg \mid$
 $s_Unauth: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\implies \ll \pi, Unauth\ e \gg m \rightarrow \ll \pi', Unauth\ e' \gg \mid$
 $s_AuthI: \quad [\text{value } v]$
 $\implies \ll \pi, Auth\ v \gg I \rightarrow \ll \pi, v \gg \mid$
 $s_UnauthI: [\text{value } v]$
 $\implies \ll \pi, Unauth\ v \gg I \rightarrow \ll \pi, v \gg \mid$
 $s_AuthP: \quad [\text{closed } (\!|v\!|); \text{value } v]$
 $\implies \ll \pi, Auth\ v \gg P \rightarrow \ll \pi, Hashed\ (hash\ (\!|v\!|))\ v \gg \mid$
 $s_UnauthP: [\text{value } v]$
 $\implies \ll \pi, Unauth\ (Hashed\ h\ v) \gg P \rightarrow \ll \pi @ [(\!|v\!|)], v \gg \mid$
 $s_AuthV: \quad [\text{closed } v; \text{value } v]$
 $\implies \ll \pi, Auth\ v \gg V \rightarrow \ll \pi, Hash\ (hash\ v) \gg \mid$
 $s_UnauthV: [\text{closed } s_0; \text{hash } s_0 = h]$
 $\implies \ll s_0 \# \pi, Unauth\ (Hash\ h) \gg V \rightarrow \ll \pi, s_0 \gg$

declare *smallstep.intros*[*simp*]
declare *smallstep.intros*[*intro*]

equivariance *smallstep*

nominal_inductive *smallstep*

avoids *s_AppLam*: *x*

| *s_AppRec*: *x*

| *s_Let1*: *x*

| *s_Let2*: *x*

by (*auto simp add: fresh_Pair fresh_subst_term*)

```

inductive smallsteps :: proofstream  $\Rightarrow$  term  $\Rightarrow$  mode  $\Rightarrow$  nat  $\Rightarrow$  proofstream  $\Rightarrow$  term  $\Rightarrow$  bool ( $\langle \ll\_ , \_ \gg$ 
 $\_ \rightarrow \_ \ll\_ , \_ \gg$ ) where
  s_Id:  $\langle \ll \pi, e \gg \_ \rightarrow 0 \ll \pi, e \gg$  |
  s_Tr: [ $\langle \ll \pi_1, e_1 \gg m \rightarrow i \ll \pi_2, e_2 \gg$ ;  $\langle \ll \pi_2, e_2 \gg m \rightarrow \ll \pi_3, e_3 \gg$  ]
         $\implies \langle \ll \pi_1, e_1 \gg m \rightarrow (i+1) \ll \pi_3, e_3 \gg$ 

declare smallsteps.intros[simp]
declare smallsteps.intros[intro]

equivariance smallsteps
nominal_inductive smallsteps .

```

```

lemma steps_1_step[simp]:  $\langle \ll \pi, e \gg m \rightarrow 1 \ll \pi', e' \gg = \langle \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$  (is ?L  $\longleftrightarrow$  ?R)
proof
  assume ?L
  then show ?R
  proof (induct  $\pi$   $e$   $m$  1::nat  $\pi'$   $e'$  rule: smallsteps.induct)
    case (s_Tr  $\pi_1$   $e_1$   $m$   $i$   $\pi_2$   $e_2$   $\pi_3$   $e_3$ )
    then show ?case
    by (induct  $\pi_1$   $e_1$   $m$   $i$   $\pi_2$   $e_2$  rule: smallsteps.induct) auto
  qed simp
qed (auto intro: s_Tr[where  $i=0$ , simplified])

```

Inversion rules for smallstep(s) predicates.

```

lemma value_no_step[intro]:
  assumes  $\langle \ll \pi_1, v \gg m \rightarrow \ll \pi_2, t \gg$  value  $v$ 
  shows False
  using assms by (induct  $\pi_1$   $v$   $m$   $\pi_2$   $t$  rule: smallstep.induct) auto

```

```

lemma subst_term_perm:
  assumes atom  $x' \# (x, e)$ 
  shows  $e[v / x] = ((x \leftrightarrow x') \cdot e)[v / x']$ 
  using assms [[simproc del: alpha_lst]]
  by (nominal_induct  $e$  avoiding:  $x$   $x'$   $v$  rule: term.strong_induct)
    (auto simp: fresh_Pair fresh_at_base(2) permute_hash_def)

```

```

inductive_cases s_Unit_inv[elim]:  $\langle \ll \pi_1, \text{Unit} \gg m \rightarrow \ll \pi_2, v \gg$ 

```

```

inductive_cases s_App_inv[consumes 1, case_names App1 App2 AppLam AppRec, elim]:  $\langle \ll \pi, \text{App}$ 
 $v_1 v_2 \gg m \rightarrow \ll \pi', e \gg$ 

```

```

lemma s_Let_inv':
  assumes  $\langle \ll \pi, \text{Let } e_1 x e_2 \gg m \rightarrow \ll \pi', e' \gg$ 
  and atom  $x \# (e_1, \pi)$ 
  obtains  $e_1'$  where  $(e' = e_2[e_1 / x] \wedge \text{value } e_1 \wedge \pi = \pi') \vee (\langle \ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg \wedge e' = \text{Let}$ 
 $e_1' x e_2 \wedge \neg \text{value } e_1)$ 
  using assms [[simproc del: alpha_lst]]
  by (atomize_elim, induct  $\pi$   $\text{Let } e_1 x e_2 m$   $\pi'$   $e'$  rule: smallstep.induct)
    (auto simp: fresh_Pair fresh_subst_term_perm_supp_eq elim: Abs_lst1_fcb2')

```

```

lemma s_Let_inv[consumes 2, case_names Let1 Let2, elim]:
  assumes  $\langle \ll \pi, \text{Let } e_1 x e_2 \gg m \rightarrow \ll \pi', e' \gg$ 
  and atom  $x \# (e_1, \pi)$ 
  and  $e' = e_2[e_1 / x] \wedge \text{value } e_1 \wedge \pi = \pi' \implies Q$ 
  and  $\bigwedge e_1'. \langle \ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg \wedge e' = \text{Let } e_1' x e_2 \wedge \neg \text{value } e_1 \implies Q$ 
  shows  $Q$ 
  using assms by (auto elim: s_Let_inv')

```

inductive_cases s_Case_inv [consumes 1, case_names Case Inj1 Inj2, elim]:
 $\ll \pi, Case\ e\ e_1\ e_2 \gg\ m \rightarrow \ll \pi', e' \gg$
inductive_cases s_Prj1_inv [consumes 1, case_names Prj1 PrjPair1, elim]:
 $\ll \pi, Prj1\ e \gg\ m \rightarrow \ll \pi', v \gg$
inductive_cases s_Prj2_inv [consumes 1, case_names Prj2 PrjPair2, elim]:
 $\ll \pi, Prj2\ e \gg\ m \rightarrow \ll \pi', v \gg$
inductive_cases s_Pair_inv [consumes 1, case_names Pair1 Pair2, elim]:
 $\ll \pi, Pair\ e_1\ e_2 \gg\ m \rightarrow \ll \pi', e' \gg$
inductive_cases s_Inj1_inv [consumes 1, case_names Inj1, elim]:
 $\ll \pi, Inj1\ e \gg\ m \rightarrow \ll \pi', e' \gg$
inductive_cases s_Inj2_inv [consumes 1, case_names Inj2, elim]:
 $\ll \pi, Inj2\ e \gg\ m \rightarrow \ll \pi', e' \gg$
inductive_cases s_Roll_inv [consumes 1, case_names Roll, elim]:
 $\ll \pi, Roll\ e \gg\ m \rightarrow \ll \pi', e' \gg$
inductive_cases s_Unroll_inv [consumes 1, case_names Unroll UnrollRoll, elim]:
 $\ll \pi, Unroll\ e \gg\ m \rightarrow \ll \pi', e' \gg$
inductive_cases s_AuthI_inv [consumes 1, case_names Auth AuthI, elim]:
 $\ll \pi, Auth\ e \gg\ I \rightarrow \ll \pi', e' \gg$
inductive_cases $s_UnauthI_inv$ [consumes 1, case_names Unauth UnauthI, elim]:
 $\ll \pi, Unauth\ e \gg\ I \rightarrow \ll \pi', e' \gg$
inductive_cases s_AuthP_inv [consumes 1, case_names Auth AuthP, elim]:
 $\ll \pi, Auth\ e \gg\ P \rightarrow \ll \pi', e' \gg$
inductive_cases $s_UnauthP_inv$ [consumes 1, case_names Unauth UnauthP, elim]:
 $\ll \pi, Unauth\ e \gg\ P \rightarrow \ll \pi', e' \gg$
inductive_cases s_AuthV_inv [consumes 1, case_names Auth AuthV, elim]:
 $\ll \pi, Auth\ e \gg\ V \rightarrow \ll \pi', e' \gg$
inductive_cases $s_UnauthV_inv$ [consumes 1, case_names Unauth UnauthV, elim]:
 $\ll \pi, Unauth\ e \gg\ V \rightarrow \ll \pi', e' \gg$

inductive_cases s_Id_inv [elim]: $\ll \pi_1, e_1 \gg\ m \rightarrow 0 \ll \pi_2, e_2 \gg$
inductive_cases s_Tr_inv [elim]: $\ll \pi_1, e_1 \gg\ m \rightarrow i \ll \pi_3, e_3 \gg$

Freshness with smallstep.

lemma *fresh_smallstep_I*:

fixes $x :: var$
assumes $\ll \pi, e \gg\ I \rightarrow \ll \pi', e' \gg\ atom\ x \# e$
shows $atom\ x \# e'$
using *assms* **by** (induct $\pi\ e\ I\ \pi'\ e'$ rule: *smallstep.induct*) (auto simp: *fresh_subst_term*)

lemma *fresh_smallstep_P*:

fixes $x :: var$
assumes $\ll \pi, e \gg\ P \rightarrow \ll \pi', e' \gg\ atom\ x \# e$
shows $atom\ x \# e'$
using *assms* **by** (induct $\pi\ e\ P\ \pi'\ e'$ rule: *smallstep.induct*) (auto simp: *fresh_subst_term*)

lemma *fresh_smallsteps_I*:

fixes $x :: var$
assumes $\ll \pi, e \gg\ I \rightarrow i \ll \pi', e' \gg\ atom\ x \# e$
shows $atom\ x \# e'$
using *assms* **by** (induct $\pi\ e\ I\ i\ \pi'\ e'$ rule: *smallsteps.induct*) (simp_all add: *fresh_smallstep_I*)

lemma *fresh_ps_smallstep_P*:

fixes $x :: var$
assumes $\ll \pi, e \gg\ P \rightarrow \ll \pi', e' \gg\ atom\ x \# e\ atom\ x \# \pi$
shows $atom\ x \# \pi'$
using *assms* **proof** (induct $\pi\ e\ P\ \pi'\ e'$ rule: *smallstep.induct*)
case ($s_UnauthP\ v\ \pi\ h$)
then show ?case

by (simp add: fresh_Cons fresh_append fresh_shallow)
qed auto

Proofstream lemmas.

lemma *smallstepI_ps_eq*:

assumes $\ll \pi, e \gg I \rightarrow \ll \pi', e' \gg$

shows $\pi = \pi'$

using *assms* by (induct $\pi e I \pi' e'$ rule: *smallstep.induct*) auto

lemma *smallstepI_ps_emptyD*:

$\ll \pi, e \gg I \rightarrow \ll [], e' \gg \implies \ll [], e \gg I \rightarrow \ll [], e' \gg$

$\ll [], e \gg I \rightarrow \ll \pi, e' \gg \implies \ll [], e \gg I \rightarrow \ll [], e' \gg$

using *smallstepI_ps_eq* by force+

lemma *smallstepsI_ps_eq*:

assumes $\ll \pi, e \gg I \rightarrow i \ll \pi', e' \gg$

shows $\pi = \pi'$

using *assms* by (induct $\pi e I i \pi' e'$ rule: *smallsteps.induct*) (auto dest: *smallstepI_ps_eq*)

lemma *smallstepsI_ps_emptyD*:

$\ll \pi, e \gg I \rightarrow i \ll [], e' \gg \implies \ll [], e \gg I \rightarrow i \ll [], e' \gg$

$\ll [], e \gg I \rightarrow i \ll \pi, e' \gg \implies \ll [], e \gg I \rightarrow i \ll [], e' \gg$

using *smallstepsI_ps_eq* by force+

lemma *smallstepV_consumes_proofstream*:

assumes $\ll \pi_1, eV \gg V \rightarrow \ll \pi_2, eV' \gg$

obtains π **where** $\pi_1 = \pi @ \pi_2$

using *assms* by (induct $\pi_1 eV V \pi_2 eV'$ rule: *smallstep.induct*) auto

lemma *smallstepsV_consumes_proofstream*:

assumes $\ll \pi_1, eV \gg V \rightarrow i \ll \pi_2, eV' \gg$

obtains π **where** $\pi_1 = \pi @ \pi_2$

using *assms* by (induct $\pi_1 eV V i \pi_2 eV'$ rule: *smallsteps.induct*)
(auto elim: *smallstepV_consumes_proofstream*)

lemma *smallstepP_generates_proofstream*:

assumes $\ll \pi_1, eP \gg P \rightarrow \ll \pi_2, eP' \gg$

obtains π **where** $\pi_2 = \pi_1 @ \pi$

using *assms* by (induct $\pi_1 eP P \pi_2 eP'$ rule: *smallstep.induct*) auto

lemma *smallstepsP_generates_proofstream*:

assumes $\ll \pi_1, eP \gg P \rightarrow i \ll \pi_2, eP' \gg$

obtains π **where** $\pi_2 = \pi_1 @ \pi$

using *assms* by (induct $\pi_1 eP P i \pi_2 eP'$ rule: *smallsteps.induct*)
(auto elim: *smallstepP_generates_proofstream*)

lemma *smallstepV_ps_append*:

$\ll \pi, eV \gg V \rightarrow \ll \pi', eV' \gg \longleftrightarrow \ll \pi @ X, eV \gg V \rightarrow \ll \pi' @ X, eV' \gg$ (is ?L \longleftrightarrow ?R)

proof (rule *iffI*)

assume ?L **then show** ?R

by (nominal_induct $\pi eV V \pi' eV'$ avoiding: X rule: *smallstep.strong_induct*)
(auto simp: *fresh_append fresh_Pair*)

next

assume ?R **then show** ?L

by (nominal_induct $\pi @ X eV V \pi' @ X eV'$ avoiding: X rule: *smallstep.strong_induct*)
(auto simp: *fresh_append fresh_Pair*)

qed

```

lemma smallstepV_ps_to_suffix:
  assumes  $\ll \pi, e \gg V \rightarrow \ll \pi' @ X, e' \gg$ 
  obtains  $\pi''$  where  $\pi = \pi'' @ X$ 
  using assms
  by (induct  $\pi e V \pi' @ X e'$  rule: smallstep.induct) auto

lemma smallstepsV_ps_append:
   $\ll \pi, eV \gg V \rightarrow i \ll \pi', eV' \gg \longleftrightarrow \ll \pi @ X, eV \gg V \rightarrow i \ll \pi' @ X, eV' \gg$  (is  $?L \longleftrightarrow ?R$ )
proof (rule iffI)
  assume  $?L$  then show  $?R$ 
  proof (induct  $\pi eV V i \pi' eV'$  rule: smallsteps.induct)
    case (s_Tr  $\pi_1 e_1 i \pi_2 e_2 \pi_3 e_3$ )
    then show  $?case$ 
      by (auto simp: iffD1[OF smallstepV_ps_append])
  qed simp
next
  assume  $?R$  then show  $?L$ 
  proof (induct  $\pi @ X eV V i \pi' @ X eV'$  arbitrary:  $\pi'$  rule: smallsteps.induct)
    case (s_Tr  $e_1 i \pi_2 e_2 e_3$ )
    from s_Tr( $\beta$ ) obtain  $\pi'''$  where  $\pi_2 = \pi''' @ X$ 
    by (auto elim: smallstepV_ps_to_suffix)
    with s_Tr show  $?case$ 
    by (auto dest: iffD2[OF smallstepV_ps_append])
  qed simp
qed

lemma smallstepP_ps_prepend:
   $\ll \pi, eP \gg P \rightarrow \ll \pi', eP' \gg \longleftrightarrow \ll X @ \pi, eP \gg P \rightarrow \ll X @ \pi', eP' \gg$  (is  $?L \longleftrightarrow ?R$ )
proof (rule iffI)
  assume  $?L$  then show  $?R$ 
  proof (nominal_induct  $\pi eP P \pi' eP'$  avoiding: X rule: smallstep.strong_induct)
    case (s_UnauthP  $v \pi h$ )
    then show  $?case$ 
      by (subst append_assoc[symmetric, of X  $\pi$  [ $\{v\}$ ]]) (erule smallstep.s_UnauthP)
  qed (auto simp: fresh_append fresh_Pair)
next
  assume  $?R$  then show  $?L$ 
  by (nominal_induct  $X @ \pi eP P X @ \pi' eP'$  avoiding: X rule: smallstep.strong_induct)
    (auto simp: fresh_append fresh_Pair)
qed

lemma smallstepsP_ps_prepend:
   $\ll \pi, eP \gg P \rightarrow i \ll \pi', eP' \gg \longleftrightarrow \ll X @ \pi, eP \gg P \rightarrow i \ll X @ \pi', eP' \gg$  (is  $?L \longleftrightarrow ?R$ )
proof (rule iffI)
  assume  $?L$  then show  $?R$ 
  proof (induct  $\pi eP P i \pi' eP'$  rule: smallsteps.induct)
    case (s_Tr  $\pi_1 e_1 i \pi_2 e_2 \pi_3 e_3$ )
    then show  $?case$ 
      by (auto simp: iffD1[OF smallstepP_ps_prepend])
  qed simp
next
  assume  $?R$  then show  $?L$ 
  proof (induct  $X @ \pi eP P i X @ \pi' eP'$  arbitrary:  $\pi'$  rule: smallsteps.induct)
    case (s_Tr  $e_1 i \pi_2 e_2 e_3$ )
    then obtain  $\pi''$  where  $\pi_2 = X @ \pi @ \pi''$ 
    by (auto elim: smallstepsP_generates_proofstream)
    then have  $\ll \pi, e_1 \gg P \rightarrow i \ll \pi @ \pi'', e_2 \gg$ 
    by (auto dest: s_Tr(2))
  qed

```

```

with  $\pi'' s\_Tr(1,3)$  show ?case
  by (auto dest: iffD2[OF smallstepP_ps_prepend])
qed simp
qed

```

3.8 Type Progress

```

lemma type_progress:
  assumes  $\{\$\$ \} \vdash_W e : \tau$ 
  shows  $value\ e \vee (\exists e'. \ll [], e \gg I \rightarrow \ll [], e' \gg)$ 
using assms proof (nominal_induct  $\{\$\$ \} :: tyenv\ e\ \tau$  rule: judge_weak.strong_induct)
  case (jw_Let x e1  $\tau_1$  e2  $\tau_2$ )
  then show ?case
    by (auto 0 3 simp: fresh_smallstep_I elim!: s_Let2[of e2]
      intro: exI[where P= $\lambda e. \ll \_, \_ \gg \_ \rightarrow \ll \_, e \gg$ , OF s_Let1, rotated])
next
  case (jw_Prj1 v  $\tau_1$   $\tau_2$ )
  then show ?case
    by (auto elim!: jw_Prod_inv[of v  $\tau_1$   $\tau_2$ ])
next
  case (jw_Prj2 v  $\tau_1$   $\tau_2$ )
  then show ?case
    by (auto elim!: jw_Prod_inv[of v  $\tau_1$   $\tau_2$ ])
next
  case (jw_App e  $\tau_1$   $\tau_2$  e')
  then show ?case
    by (auto 0 4 elim: jw_Fun_inv[of _ _ _ e'])
next
  case (jw_Case v v1 v2  $\tau_1$   $\tau_2$   $\tau$ )
  then show ?case
    by (auto 0 4 elim: jw_Sum_inv[of _ v1 v2])
qed fast+

```

3.9 Weak Type Preservation

```

lemma fresh_tyenv_None:
  fixes  $\Gamma :: tyenv$ 
  shows  $atom\ x \# \Gamma \longleftrightarrow \Gamma\ \$\$ x = None$  (is ?L  $\longleftrightarrow$  ?R)
proof
  assume asm: ?L show ?R
  proof (rule ccontr)
    assume  $\Gamma\ \$\$ x \neq None$ 
    then obtain  $\tau$  where  $\Gamma\ \$\$ x = Some\ \tau$  by blast
    with asm have  $\forall a :: var. atom\ a \# \Gamma \longrightarrow \Gamma\ \$\$ a = Some\ \tau$ 
      using fmap_freshness_lemma_unique[OF exI, of x  $\Gamma$ ]
      by (simp add: fresh_Pair fresh_Some) metis
    then have  $\{a :: var. atom\ a \# \Gamma\} \subseteq fmdom'\ \Gamma$ 
      by (auto simp: image_iff Ball_def fmllookup_dom'_iff)
    moreover
    {
      assume infinite  $\{a :: var. \neg atom\ a \# \Gamma\}$ 
      then have infinite  $\{a :: var. atom\ a \in supp\ \Gamma\}$ 
        unfolding fresh_def by auto
      then have infinite (supp  $\Gamma$ )
        by (rule contrapos_nn)
        (auto simp: image_iff inv_f_f[of atom] inj_on_def
          elim!: finite_surj[of _ _ inv atom] beXI[rotated])
      then have False
        using finite_supp[of  $\Gamma$ ] by blast
    }
  }

```

```

then have infinite {a :: var. atom a # Γ}
  by auto
ultimately show False
  using finite_subset[of {a. atom a # Γ} fmdom' Γ] unfolding fmdom'_alt_def
  by auto
qed
next
assume ?R then show ?L
proof (induct Γ arbitrary: x)
  case (fmupd y z Γ)
  then show ?case
    by (cases y = x) (auto intro: fresh_fmap_update)
  qed simp
qed

```

lemma judge_weak_fresh_env_fresh_term[dest]:

```

fixes a :: var
assumes Γ ⊢W e : τ atom a # Γ
shows atom a # e
using assms proof (nominal_induct Γ e τ avoiding: a rule: judge_weak.strong_induct)
  case (jw_Var Γ x τ)
  then show ?case
    by (cases a = x) (auto simp: fresh_tyenv_None)
  qed (simp_all add: fresh_Cons fresh_fmap_update)

```

lemma judge_weak_weakening_1:

```

assumes Γ ⊢W e : τ atom y # e
shows Γ(y ::= τ') ⊢W e : τ
using assms proof (nominal_induct Γ e τ avoiding: y τ' rule: judge_weak.strong_induct)
  case (jw_Lam x Γ τ1 e τ2)
  from jw_Lam(5)[of y τ'] jw_Lam(1-4,6) show ?case
    by (auto simp add: fresh_at_base fmupd_reorder_neq fresh_fmap_update)
  next
  case (jw_App v v' Γ τ1 τ2)
  then show ?case
    by (force simp add: fresh_at_base fmupd_reorder_neq fresh_fmap_update)
  next
  case (jw_Let x Γ e1 τ1 e2 τ2)
  from jw_Let(6)[of y τ'] jw_Let(8)[of y τ'] jw_Let(1-5,7,9) show ?case
    by (auto simp add: fresh_at_base fmupd_reorder_neq fresh_fmap_update)
  next
  case (jw_Rec x Γ z τ1 τ2 e')
  from jw_Rec(9)[of y τ'] jw_Rec(1-8,10) show ?case
    by (auto simp add: fresh_at_base fmupd_reorder_neq fresh_fmap_update fresh_Pair)
  next
  case (jw_Case v v1 v2 Γ τ1 τ2 τ)
  then show ?case
    by (fastforce simp add: fresh_at_base fmupd_reorder_neq fresh_fmap_update)
  next
  case (jw_Roll α Γ v τ)
  then show ?case
    by (simp add: fresh_fmap_update)
  next
  case (jw_Unroll α Γ v τ)
  then show ?case
    by (simp add: fresh_fmap_update)
  qed auto

```

```

lemma judge_weak_weakening_2:
  assumes  $\Gamma \vdash_W e : \tau$  atom  $y \# \Gamma$ 
  shows  $\Gamma(y \text{ \#\#} := \tau') \vdash_W e : \tau$ 
  proof -
    from assms have atom  $y \# e$ 
    by (rule judge_weak_fresh_env_fresh_term)
    with assms show  $\Gamma(y \text{ \#\#} := \tau') \vdash_W e : \tau$  by (simp add: judge_weak_weakening_1)
  qed

lemma judge_weak_weakening_env:
  assumes  $\{\#\#\} \vdash_W e : \tau$ 
  shows  $\Gamma \vdash_W e : \tau$ 
using assms proof (induct  $\Gamma$ )
  case fmempty
  then show ?case by assumption
next
  case (fmupd  $x\ y\ \Gamma$ )
  then show ?case
    by (simp add: fresh_tyenv_None judge_weak_weakening_2)
qed

lemma value_subst_value:
  assumes value  $e\ e'$ 
  shows value ( $e[e' / x]$ )
  using assms by (induct  $e\ e'\ x$  rule: subst_term.induct) auto

lemma judge_weak_subst[intro]:
  assumes  $\Gamma(a \text{ \#\#} := \tau') \vdash_W e : \tau$   $\{\#\#\} \vdash_W e' : \tau'$ 
  shows  $\Gamma \vdash_W e[e' / a] : \tau$ 
  using assms proof (nominal_induct  $\Gamma(a \text{ \#\#} := \tau')$   $e\ \tau$  avoiding:  $a\ e'\ \Gamma$  rule: judge_weak.strong_induct)
  case (jw_Var  $x\ \tau$ )
  then show ?case
    by (auto simp: judge_weak_weakening_env)
next
  case (jw_Lam  $x\ \tau_1\ e\ \tau_2$ )
  then show ?case
    by (fastforce simp: fmupd_reorder_neq)
next
  case (jw_Rec  $x\ y\ \tau_1\ \tau_2\ e$ )
  then show ?case
    by (fastforce simp: fmupd_reorder_neq)
next
  case (jw_Let  $x\ e_1\ \tau_1\ e_2\ \tau_2$ )
  then show ?case
    by (fastforce simp: fmupd_reorder_neq)
qed fastforce

lemma type_preservation:
  assumes  $\ll \[], e \gg I \rightarrow \ll \[], e' \gg \{\#\#\} \vdash_W e : \tau$ 
  shows  $\{\#\#\} \vdash_W e' : \tau$ 
  using assms  $[[\textit{simproc}\ \textit{del}: \textit{alpha\_lst}]]$ 
proof (nominal_induct  $[\textit{proofstream}\ e\ I\ \textit{proofstream}\ e']$  arbitrary:  $\tau$  rule: smallstep.strong_induct)
  case (s_AppLam  $v\ x\ e$ )
  then show ?case by force
next
  case (s_AppRec  $v\ x\ e$ )
  then show ?case
    by (elim jw_App_inv jw_Rec_inv) (auto  $0\ 3$  simp del: subst_term.simps)

```

```

next
  case (s_Let1 x e1 e1' e2)
  from s_Let1(1,2,7) show ?case
  by (auto intro: s_Let1(6) del: jw_Let_inv elim!: jw_Let_inv)
next
  case (s_Unroll e e')
  then obtain α::tvar where atom α # τ
  using obtain_fresh by blast
  with s_Unroll show ?case
  by (auto elim: jw_Unroll_inv[where α = α])
next
  case (s_Roll e e')
  then obtain α::tvar where atom α # τ
  using obtain_fresh by blast
  with s_Roll show ?case
  by (auto elim: jw_Roll_inv[where α = α])
next
  case (s_UnrollRoll v)
  then obtain α::tvar where atom α # τ
  using obtain_fresh by blast
  with s_UnrollRoll show ?case
  by (fastforce simp: Abs1_eq(3) elim: jw_Roll_inv[where α = α] jw_Unroll_inv[where α = α])
qed fastforce+

```

3.10 Corrected Lemma 1 from Miller et al. [2]: Weak Type Soundness

```

lemma type_soundness:
  assumes {$$} ⊢W e : τ
  shows value e ∨ (∃ e'. << [], e >> I → << [], e' >> ∧ {$$} ⊢W e' : τ)
proof (cases value e)
  case True
  then show ?thesis by simp
next
  case False
  with assms obtain e' where << [], e >> I → << [], e' >> by (auto dest: type_progress)
  with assms show ?thesis
  by (auto simp: type_preservation)
qed

```

4 Agreement Relation

inductive agree :: tyenv ⇒ term ⇒ term ⇒ term ⇒ ty ⇒ bool (⟨_ ⊢ _, _, _ : _⟩ [150,0,0,0,150] 149)

where

```

a_Unit: Γ ⊢ Unit, Unit, Unit : One |
a_Var: Γ $$ x = Some τ
  ⇒ Γ ⊢ Var x, Var x, Var x : τ |
a_Lam: [ atom x # Γ; Γ(x $$:= τ1) ⊢ e, eP, eV : τ2 ]
  ⇒ Γ ⊢ Lam x e, Lam x eP, Lam x eV : Fun τ1 τ2 |
a_App: [ Γ ⊢ e1, eP1, eV1 : Fun τ1 τ2; Γ ⊢ e2, eP2, eV2 : τ1 ]
  ⇒ Γ ⊢ App e1 e2, App eP1 eP2, App eV1 eV2 : τ2 |
a_Let: [ atom x # (Γ, e1, eP1, eV1); Γ ⊢ e1, eP1, eV1 : τ1; Γ(x $$:= τ1) ⊢ e2, eP2, eV2 : τ2 ]
  ⇒ Γ ⊢ Let e1 x e2, Let eP1 x eP2, Let eV1 x eV2 : τ2 |
a_Rec: [ atom x # Γ; atom y # (Γ,x); Γ(x $$:= Fun τ1 τ2) ⊢ Lam y e, Lam y eP, Lam y eV : Fun τ1
τ2 ]
  ⇒ Γ ⊢ Rec x (Lam y e), Rec x (Lam y eP), Rec x (Lam y eV) : Fun τ1 τ2 |
a_Inj1: [ Γ ⊢ e, eP, eV : τ1 ]

```

```

    ⇒ Γ ⊢ Inj1 e, Inj1 eP, Inj1 eV : Sum τ1 τ2 |
a_Inj2: [ [ Γ ⊢ e, eP, eV : τ2 ]
    ⇒ Γ ⊢ Inj2 e, Inj2 eP, Inj2 eV : Sum τ1 τ2 |
a_Case: [ [ Γ ⊢ e, eP, eV : Sum τ1 τ2; Γ ⊢ e1, eP1, eV1 : Fun τ1 τ; Γ ⊢ e2, eP2, eV2 : Fun τ2 τ ]
    ⇒ Γ ⊢ Case e e1 e2, Case eP eP1 eP2, Case eV eV1 eV2 : τ |
a_Pair: [ [ Γ ⊢ e1, eP1, eV1 : τ1; Γ ⊢ e2, eP2, eV2 : τ2 ]
    ⇒ Γ ⊢ Pair e1 e2, Pair eP1 eP2, Pair eV1 eV2 : Prod τ1 τ2 |
a_Prj1: [ [ Γ ⊢ e, eP, eV : Prod τ1 τ2 ]
    ⇒ Γ ⊢ Prj1 e, Prj1 eP, Prj1 eV : τ1 |
a_Prj2: [ [ Γ ⊢ e, eP, eV : Prod τ1 τ2 ]
    ⇒ Γ ⊢ Prj2 e, Prj2 eP, Prj2 eV : τ2 |
a_Roll: [ [ atom α # Γ; Γ ⊢ e, eP, eV : subst_type τ (Mu α τ) α ]
    ⇒ Γ ⊢ Roll e, Roll eP, Roll eV : Mu α τ |
a_Unroll: [ [ atom α # Γ; Γ ⊢ e, eP, eV : Mu α τ ]
    ⇒ Γ ⊢ Unroll e, Unroll eP, Unroll eV : subst_type τ (Mu α τ) α |
a_Auth: [ [ Γ ⊢ e, eP, eV : τ ]
    ⇒ Γ ⊢ Auth e, Auth eP, Auth eV : AuthT τ |
a_Unauth: [ [ Γ ⊢ e, eP, eV : AuthT τ ]
    ⇒ Γ ⊢ Unauth e, Unauth eP, Unauth eV : τ |
a_HashI: [ [ { $$ } ⊢ v, vP, (vP) : τ; hash (vP) = h; value v; value vP ]
    ⇒ Γ ⊢ v, Hashed h vP, Hash h : AuthT τ

```

declare *agree.intros*[*intro*]

equivariance *agree*

nominal_inductive *agree*

```

avoids a_Lam: x
  | a_Rec: x and y
  | a_Let: x
  | a_Roll: α
  | a_Unroll: α
by (auto simp: fresh_Pair fresh_subst_type)

```

lemma *Abs_lst_eq_3tuple*:

```

fixes x x' :: var
fixes e eP eV e' eP' eV' :: term
assumes [[atom x]]lst. e = [[atom x']]lst. e'
and [[atom x]]lst. eP = [[atom x']]lst. eP'
and [[atom x]]lst. eV = [[atom x']]lst. eV'
shows [[atom x]]lst. (e, eP, eV) = [[atom x']]lst. (e', eP', eV')
using assms by (simp add: fresh_Pair)

```

lemma *agree_fresh_env_fresh_term*:

```

fixes a :: var
assumes Γ ⊢ e, eP, eV : τ atom a # Γ
shows atom a # (e, eP, eV)
using assms proof (nominal_induct Γ e eP eV τ avoiding: a rule: agree.strong_induct)
case (a_Var Γ x τ)
then show ?case
  by (cases a = x) (auto simp: fresh_tyenv_None)
qed (simp_all add: fresh_Cons fresh_fmap_update fresh_Pair)

```

lemma *agree_empty_fresh*[*dest*]:

```

fixes a :: var
assumes { $$ } ⊢ e, eP, eV : τ
shows {atom a} #* {e, eP, eV}
using assms by (auto simp: fresh_Pair dest: agree_fresh_env_fresh_term)

```

Inversion rules for agreement.

declare $[[\text{simproc del: alpha_lst}]]$

lemma $a_Lam_inv_I[\text{elim}]$:

assumes $\Gamma \vdash (Lam\ x\ e'), eP, eV : (Fun\ \tau_1\ \tau_2)$

and $atom\ x \# \Gamma$

obtains $eP' eV'$ **where** $eP = Lam\ x\ eP' eV = Lam\ x\ eV' \Gamma(x\ \$\$:= \tau_1) \vdash e', eP', eV' : \tau_2$

using $assms$

proof $(atomize_elim, nominal_induct\ \Gamma\ Lam\ x\ e'\ eP\ eV\ Fun\ \tau_1\ \tau_2\ avoiding: x\ e'\ \tau_1\ \tau_2\ rule: agree.strong_induct)$

case $(a_Lam\ x\ \Gamma\ \tau_1\ e\ eP\ eV\ \tau_2\ y\ e')$

show $?case$

proof $(intro\ exI\ conjI)$

from a_Lam **show** $Lam\ x\ eP = Lam\ y\ ((x \leftrightarrow y) \cdot eP)$

by $(auto\ intro! : Abs_lst_eq_flipI\ dest! : agree_fresh_env_fresh_term\ simp: fresh_fmap_update\ fresh_Pair)$

from a_Lam **show** $Lam\ x\ eV = Lam\ y\ ((x \leftrightarrow y) \cdot eV)$

by $(auto\ intro! : Abs_lst_eq_flipI\ dest! : agree_fresh_env_fresh_term\ simp: fresh_fmap_update\ fresh_Pair)$

from $a_Lam(1-6,8,10)$ **show** $\Gamma(y\ \$\$:= \tau_1) \vdash e', (x \leftrightarrow y) \cdot eP, (x \leftrightarrow y) \cdot eV : \tau_2$

by $(auto\ simp: perm_supp_eq\ Abs1_eq_iff(3))$

dest! $agree.eqt[\text{where } p = (x \leftrightarrow y), \text{ of } \Gamma(x\ \$\$:= \tau_1)]$

qed

qed

lemma $a_Lam_inv_P[\text{elim}]$:

assumes $\{\$\$ \} \vdash v, (Lam\ x\ vP'), vV : (Fun\ \tau_1\ \tau_2)$

obtains $v' vV'$ **where** $v = Lam\ x\ v' vV = Lam\ x\ vV' \{\$\$ \}(x\ \$\$:= \tau_1) \vdash v', vP', vV' : \tau_2$

using $assms$

proof $(atomize_elim, nominal_induct\ \{\$\$ \}::tyenv\ v\ Lam\ x\ vP'\ vV\ Fun\ \tau_1\ \tau_2\ avoiding: x\ vP'\ rule: agree.strong_induct)$

case $(a_Lam\ x'\ e\ eP\ eV)$

show $?case$

proof $(intro\ exI\ conjI)$

from a_Lam **show** $Lam\ x'\ e = Lam\ x\ ((x' \leftrightarrow x) \cdot e)$

by $(auto\ intro! : Abs_lst_eq_flipI\ dest! : agree_fresh_env_fresh_term\ simp: fresh_fmap_update\ fresh_Pair)$

from a_Lam **show** $Lam\ x'\ eV = Lam\ x\ ((x' \leftrightarrow x) \cdot eV)$

by $(auto\ intro! : Abs_lst_eq_flipI\ dest! : agree_fresh_env_fresh_term\ simp: fresh_fmap_update\ fresh_Pair)$

from $a_Lam(1-4,6)$ **show** $\{\$\$ \}(x\ \$\$:= \tau_1) \vdash (x' \leftrightarrow x) \cdot e, vP', (x' \leftrightarrow x) \cdot eV : \tau_2$

by $(auto\ simp: perm_supp_eq\ Abs1_eq_iff(3))$

dest! $agree.eqt[\text{where } p = (x' \leftrightarrow x), \text{ of } \{\$\$ \}(x' \$\$:= \tau_1)]$

qed

qed

lemma $a_Lam_inv_V[\text{elim}]$:

assumes $\{\$\$ \} \vdash v, vP, (Lam\ x\ vV') : (Fun\ \tau_1\ \tau_2)$

obtains $v' vP'$ **where** $v = Lam\ x\ v' vP = Lam\ x\ vP' \{\$\$ \}(x\ \$\$:= \tau_1) \vdash v', vP', vV' : \tau_2$

using $assms$

proof $(atomize_elim, nominal_induct\ \{\$\$ \}::tyenv\ v\ vP\ Lam\ x\ vV'\ Fun\ \tau_1\ \tau_2\ avoiding: x\ vV'\ rule: agree.strong_induct)$

case $(a_Lam\ x'\ e\ eP\ eV)$

show $?case$

proof $(intro\ exI\ conjI)$

from a_Lam **show** $Lam\ x'\ e = Lam\ x\ ((x' \leftrightarrow x) \cdot e)$

by $(auto\ intro! : Abs_lst_eq_flipI\ dest! : agree_fresh_env_fresh_term\ simp: fresh_fmap_update\ fresh_Pair)$

from a_Lam **show** $Lam\ x'\ eP = Lam\ x\ ((x' \leftrightarrow x) \cdot eP)$

by $(auto\ intro! : Abs_lst_eq_flipI\ dest! : agree_fresh_env_fresh_term)$

```

    simp: fresh_fmap_update fresh_Pair)
  from a_Lam(1-4,6) show { $\$ \$$ }(x  $\$ \$$  :=  $\tau_1$ )  $\vdash$  (x'  $\leftrightarrow$  x)  $\cdot$  e, (x'  $\leftrightarrow$  x)  $\cdot$  eP, vV' :  $\tau_2$ 
  by (auto simp: perm_supp_eq Abs1_eq_iff(3)
      dest!: agree.eqvt[where p = (x'  $\leftrightarrow$  x), of { $\$ \$$ }(x'  $\$ \$$  :=  $\tau_1$ )])
qed
qed

lemma a_Rec_inv_I[elim]:
  assumes  $\Gamma \vdash \text{Rec } x \ e, eP, eV : \text{Fun } \tau_1 \ \tau_2$ 
  and atom x  $\#$   $\Gamma$ 
  obtains y e' eP' eV'
  where e = Lam y e' eP = Rec x (Lam y eP') eV = Rec x (Lam y eV') atom y  $\#$  ( $\Gamma, x$ )
         $\Gamma(x \ \$ \$ := \text{Fun } \tau_1 \ \tau_2) \vdash \text{Lam } y \ e', \text{Lam } y \ eP', \text{Lam } y \ eV' : \text{Fun } \tau_1 \ \tau_2$ 
  using assms
proof (atomize_elim, nominal_induct  $\Gamma \ \text{Rec } x \ e \ eP \ eV \ \text{Fun } \tau_1 \ \tau_2$  avoiding: x e rule: agree.strong_induct)
  case (a_Rec x'  $\Gamma \ y \ e' \ eP \ eV$ )
  then show ?case
  proof (nominal_induct e avoiding: x x' y rule: term.strong_induct)
    case Unit
    from Unit(9) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Var x)
    from Var(9) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Lam z ee)
    show ?case
    proof (intro conjI exI)
      from Lam(1-3,5-13,15) show Lam z ee = Lam y ((z  $\leftrightarrow$  y)  $\cdot$  ee)
      by (auto intro: Abs_lst_eq_flipI simp: fresh_fmap_update fresh_Pair)
      from Lam(1-3,5-13,15) show Rec x' (Lam y eP) = Rec x (Lam y ((x'  $\leftrightarrow$  x)  $\cdot$  eP))
      using Abs_lst_eq_flipI[of x Lam y eP x']
      by (elim agree_fresh_env_fresh_term[where a = x, elim_format])
      (simp_all add: fresh_fmap_update fresh_Pair)
      from Lam(1-3,5-13,15) show Rec x' (Lam y eV) = Rec x (Lam y ((x'  $\leftrightarrow$  x)  $\cdot$  eV))
      using Abs_lst_eq_flipI[of x Lam y eV x']
      by (elim agree_fresh_env_fresh_term[where a = x, elim_format])
      (simp_all add: fresh_fmap_update fresh_Pair)
      from Lam(7,10) show atom y  $\#$  ( $\Gamma, x$ )
      by simp
      from Lam(1-3,5-11,13) have (x'  $\leftrightarrow$  x)  $\cdot$  e' = (z  $\leftrightarrow$  y)  $\cdot$  ee
      by (simp add: Abs1_eq_iff flip_commute swap_permute_swap fresh_perm fresh_at_base)
      with Lam(1-2,7,9,11-12,15) show  $\Gamma(x \ \$ \$ := \text{Fun } \tau_1 \ \tau_2) \vdash$ 
        Lam y ((z  $\leftrightarrow$  y)  $\cdot$  ee), Lam y ((x'  $\leftrightarrow$  x)  $\cdot$  eP), Lam y ((x'  $\leftrightarrow$  x)  $\cdot$  eV) : Fun  $\tau_1 \ \tau_2$ 
      by (elim agree.eqvt[of _ Lam y e' _ _ (x'  $\leftrightarrow$  x), elim_format]) (simp add: perm_supp_eq)
    qed
  qed
next
  case (Rec x1 x2a)
  from Rec(13) show ?case by (simp add: Abs1_eq_iff)
next
  case (Inj1 x)
  from Inj1(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Inj2 x)
  from Inj2(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Pair x1 x2a)
  from Pair(11) show ?case by (simp add: Abs1_eq_iff)
next

```

```

    case (Roll x)
  from Roll(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Let x1 x2a x3)
  from Let(14) show ?case by (simp add: Abs1_eq_iff)
next
  case (App x1 x2a)
  from App(11) show ?case by (simp add: Abs1_eq_iff)
next
  case (Case x1 x2a x3)
  from Case(12) show ?case by (simp add: Abs1_eq_iff)
next
  case (Prj1 x)
  from Prj1(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Prj2 x)
  from Prj2(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Unroll x)
  from Unroll(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Auth x)
  from Auth(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Unauth x)
  from Unauth(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Hash x)
  from Hash(9) show ?case by (simp add: Abs1_eq_iff)
next
  case (Hashed x1 x2a)
  from Hashed(10) show ?case by (simp add: Abs1_eq_iff)
qed
qed

```

lemma $a_Rec_inv_P[elim]$:

```

assumes  $\Gamma \vdash e, Rec\ x\ eP, eV : Fun\ \tau_1\ \tau_2$ 
and  $atom\ x\ \# \Gamma$ 
obtains  $y\ e'\ eP'\ eV'$ 
where  $e = Rec\ x\ (Lam\ y\ e')\ eP = Lam\ y\ eP'\ eV = Rec\ x\ (Lam\ y\ eV')\ atom\ y\ \# (\Gamma, x)$ 
 $\Gamma(x\ \$\$ := Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e', Lam\ y\ eP', Lam\ y\ eV' : Fun\ \tau_1\ \tau_2$ 
using assms
proof (atomize_elim, nominal_induct  $\Gamma\ e\ Rec\ x\ eP\ eV\ Fun\ \tau_1\ \tau_2$  avoiding:  $x\ eP$  rule: agree.strong_induct)
case (a_Rec x  $\Gamma\ y\ e\ eP\ eV\ x'\ eP'$ )
then show ?case
proof (nominal_induct  $eP'$  avoiding:  $x'\ x\ y$  rule: term.strong_induct)
case Unit
from Unit(9) show ?case by (simp add: Abs1_eq_iff)
next
case (Var x)
from Var(9) show ?case by (simp add: Abs1_eq_iff)
next
case (Lam ya  $eP'$ )
show ?case
proof (intro conjI exI)
from Lam(1-3,5-13,15) show  $Rec\ x\ (Lam\ y\ e) = Rec\ x'\ (Lam\ y\ ((x \leftrightarrow x') \cdot e))$ 
using Abs_lst_eq_flipI[of  $x'\ Lam\ y\ e\ x$ ]
by (elim agree_fresh_env_fresh_term[where  $a = x', elim\_format$ ])

```

```

    (simp_all add: fresh_fmap_update fresh_Pair)
  from Lam(1-3,5-13,15) show Lam ya eP' = Lam y ((x ↔ x') · eP)
    unfolding trans[OF eq_sym_conv Abs1_eq_iff(3)]
    by (simp add: flip_commute fresh_at_base)
  from Lam(1-3,5-13,15) show Rec x (Lam y eV) = Rec x' (Lam y ((x ↔ x') · eV))
    using Abs_lst_eq_flipI[of x' Lam y eV x]
    by (elim agree_fresh_env_fresh_term[where a = x', elim_format])
    (simp_all add: fresh_fmap_update fresh_Pair)
  from Lam(7,10) show atom y # (Γ, x')
    by simp
  with Lam(1-2,7,9,11-12,15) show Γ(x' $$$ := Fun τ1 τ2) ⊢
    Lam y ((x ↔ x') · e), Lam y ((x ↔ x') · eP), Lam y ((x ↔ x') · eV) : Fun τ1 τ2
    by (elim agree_eqvt[of _ Lam y _ _ _ (x' ↔ x), elim_format])
    (simp add: perm_supp_eq flip_commute)
qed
next
  case (Rec x1 x2a)
  from Rec(13) show ?case by (simp add: Abs1_eq_iff)
next
  case (Inj1 x)
  from Inj1(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Inj2 x)
  from Inj2(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Pair x1 x2a)
  from Pair(11) show ?case by (simp add: Abs1_eq_iff)
next
  case (Roll x)
  from Roll(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Let x1 x2a x3)
  from Let(14) show ?case by (simp add: Abs1_eq_iff)
next
  case (App x1 x2a)
  from App(11) show ?case by (simp add: Abs1_eq_iff)
next
  case (Case x1 x2a x3)
  from Case(12) show ?case by (simp add: Abs1_eq_iff)
next
  case (Prj1 x)
  from Prj1(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Prj2 x)
  from Prj2(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Unroll x)
  from Unroll(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Auth x)
  from Auth(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Unauth x)
  from Unauth(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Hash x)
  from Hash(9) show ?case by (simp add: Abs1_eq_iff)
next

```

```

    case (Hashed x1 x2a)
  from Hashed(10) show ?case by (simp add: Abs1_eq_iff)
qed
qed

lemma a_Rec_inv_V[elim]:
  assumes  $\Gamma \vdash e, eP, \text{Rec } x \text{ eV} : \text{Fun } \tau_1 \tau_2$ 
  and  $\text{atom } x \# \Gamma$ 
  obtains  $y \text{ e}' \text{ eP}' \text{ eV}'$ 
  where  $e = \text{Rec } x (\text{Lam } y \text{ e}') \text{ eP} = \text{Rec } x (\text{Lam } y \text{ eP}') \text{ eV} = \text{Lam } y \text{ eV}' \text{ atom } y \# (\Gamma, x)$ 
   $\Gamma(x \text{ \#\#} := \text{Fun } \tau_1 \tau_2) \vdash \text{Lam } y \text{ e}', \text{Lam } y \text{ eP}', \text{Lam } y \text{ eV}' : \text{Fun } \tau_1 \tau_2$ 
  using assms
proof (atomize_elim, nominal_induct  $\Gamma \text{ e eP Rec } x \text{ eV Fun } \tau_1 \tau_2$  avoiding:  $x \text{ eV}$  rule: agree.strong_induct)
  case (a_Rec  $x \Gamma y \text{ e eP eV}' \text{ eV}'$ )
  then show ?case
  proof (nominal_induct  $\text{eV}'$  avoiding:  $x' \text{ x y}$  rule: term.strong_induct)
    case Unit
    from Unit(9) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Var  $x$ )
    from Var(9) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Lam  $ya \text{ eV}'$ )
    show ?case
    proof (intro conjI exI)
      from Lam(1-3,5-13,15) show  $\text{Rec } x (\text{Lam } y \text{ e}) = \text{Rec } x' (\text{Lam } y ((x \leftrightarrow x') \cdot \text{e}))$ 
      using Abs_lst_eq_flipI[of  $x' \text{ Lam } y \text{ e } x$ ]
      by (elim agree_fresh_env_fresh_term[where  $a = x', \text{elim\_format}$ ])
      (simp_all add: fresh_fmap_update fresh_Pair)
      from Lam(1-3,5-13,15) show  $\text{Lam } ya \text{ eV}' = \text{Lam } y ((x \leftrightarrow x') \cdot \text{eV})$ 
      unfolding trans[OF eq_sym_conv Abs1_eq_iff(3)]
      by (simp add: flip_commute fresh_at_base)
      from Lam(1-3,5-13,15) show  $\text{Rec } x (\text{Lam } y \text{ eP}) = \text{Rec } x' (\text{Lam } y ((x \leftrightarrow x') \cdot \text{eP}))$ 
      using Abs_lst_eq_flipI[of  $x' \text{ Lam } y \text{ eP } x$ ]
      by (elim agree_fresh_env_fresh_term[where  $a = x', \text{elim\_format}$ ])
      (simp_all add: fresh_fmap_update fresh_Pair)
      from Lam(7,10) show  $\text{atom } y \# (\Gamma, x')$ 
      by simp
      with Lam(1-2,7,9,11-12,15) show  $\Gamma(x' \text{ \#\#} := \text{Fun } \tau_1 \tau_2) \vdash$ 
       $\text{Lam } y ((x \leftrightarrow x') \cdot \text{e}), \text{Lam } y ((x \leftrightarrow x') \cdot \text{eP}), \text{Lam } y ((x \leftrightarrow x') \cdot \text{eV}) : \text{Fun } \tau_1 \tau_2$ 
      by (elim agree_eqvt[of  $\_ \text{ Lam } y \_ \_ \_ (x' \leftrightarrow x), \text{elim\_format}$ ])
      (simp add: perm_supp_eq_flip_commute)
    qed
  next
    case (Rec  $x1 \text{ x2a}$ )
    from Rec(13) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Inj1  $x$ )
    from Inj1(10) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Inj2  $x$ )
    from Inj2(10) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Pair  $x1 \text{ x2a}$ )
    from Pair(11) show ?case by (simp add: Abs1_eq_iff)
  next
    case (Roll  $x$ )
    from Roll(10) show ?case by (simp add: Abs1_eq_iff)

```

```

next
  case (Let x1 x2a x3)
  from Let(14) show ?case by (simp add: Abs1_eq_iff)
next
  case (App x1 x2a)
  from App(11) show ?case by (simp add: Abs1_eq_iff)
next
  case (Case x1 x2a x3)
  from Case(12) show ?case by (simp add: Abs1_eq_iff)
next
  case (Prj1 x)
  from Prj1(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Prj2 x)
  from Prj2(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Unroll x)
  from Unroll(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Auth x)
  from Auth(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Unauth x)
  from Unauth(10) show ?case by (simp add: Abs1_eq_iff)
next
  case (Hash x)
  from Hash(9) show ?case by (simp add: Abs1_eq_iff)
next
  case (Hashed x1 x2a)
  from Hashed(10) show ?case by (simp add: Abs1_eq_iff)
qed

```

```

inductive_cases a_Inj1_inv_I[elim]:  $\Gamma \vdash \text{Inj1 } e, eP, eV : \text{Sum } \tau_1 \tau_2$ 
inductive_cases a_Inj1_inv_P[elim]:  $\Gamma \vdash e, \text{Inj1 } eP, eV : \text{Sum } \tau_1 \tau_2$ 
inductive_cases a_Inj1_inv_V[elim]:  $\Gamma \vdash e, eP, \text{Inj1 } eV : \text{Sum } \tau_1 \tau_2$ 

```

```

inductive_cases a_Inj2_inv_I[elim]:  $\Gamma \vdash \text{Inj2 } e, eP, eV : \text{Sum } \tau_1 \tau_2$ 
inductive_cases a_Inj2_inv_P[elim]:  $\Gamma \vdash e, \text{Inj2 } eP, eV : \text{Sum } \tau_1 \tau_2$ 
inductive_cases a_Inj2_inv_V[elim]:  $\Gamma \vdash e, eP, \text{Inj2 } eV : \text{Sum } \tau_1 \tau_2$ 

```

```

inductive_cases a_Pair_inv_I[elim]:  $\Gamma \vdash \text{Pair } e_1 e_2, eP, eV : \text{Prod } \tau_1 \tau_2$ 
inductive_cases a_Pair_inv_P[elim]:  $\Gamma \vdash e, \text{Pair } eP_1 eP_2, eV : \text{Prod } \tau_1 \tau_2$ 

```

```

lemma a_Roll_inv_I[elim]:
  assumes  $\Gamma \vdash \text{Roll } e', eP, eV : \text{Mu } \alpha \tau$ 
  obtains  $eP' eV'$ 
  where  $eP = \text{Roll } eP' eV = \text{Roll } eV' \Gamma \vdash e', eP', eV' : \text{subst\_type } \tau (\text{Mu } \alpha \tau) \alpha$ 
  using assms
  by (nominal_induct  $\Gamma \text{Roll } e' eP eV \text{Mu } \alpha \tau$  avoiding:  $\alpha \tau$  rule: agree.strong_induct)
    (simp add: Abs1_eq(3) Abs_lst_eq_flipI subst_type_perm_eq)

```

```

lemma a_Roll_inv_P[elim]:
  assumes  $\Gamma \vdash e, \text{Roll } eP', eV : \text{Mu } \alpha \tau$ 
  obtains  $e' eV'$ 
  where  $e = \text{Roll } e' eV = \text{Roll } eV' \Gamma \vdash e', eP', eV' : \text{subst\_type } \tau (\text{Mu } \alpha \tau) \alpha$ 
  using assms
  by (nominal_induct  $\Gamma e \text{Roll } eP' eV \text{Mu } \alpha \tau$  avoiding:  $\alpha \tau$  rule: agree.strong_induct)

```

(simp add: Abs1_eq(3) Abs_lst_eq_flipI subst_type_perm_eq)

lemma *a_Roll_inv_V[elim]*:

assumes $\Gamma \vdash e, eP, \text{Roll } eV' : \text{Mu } \alpha \tau$

obtains $e' eP'$

where $e = \text{Roll } e' eP = \text{Roll } eP' \Gamma \vdash e', eP', eV' : \text{subst_type } \tau (\text{Mu } \alpha \tau) \alpha$

using *assms*

by (*nominal_induct* $\Gamma e eP \text{Roll } eV' \text{Mu } \alpha \tau$ *avoiding*: $\alpha \tau$ *rule*: *agree.strong_induct*)

(simp add: Abs1_eq(3) Abs_lst_eq_flipI subst_type_perm_eq)

inductive_cases *a_HashI_inv[elim]*: $\Gamma \vdash v, \text{Hashed } (\text{hash } (\downarrow vP)) vP, \text{Hash } (\text{hash } (\downarrow vP)) : \text{AuthT } \tau$

Inversion on types for agreement.

lemma *a_AuthT_value_inv*:

assumes $\{\$\$ \} \vdash v, vP, vV : \text{AuthT } \tau$

and *value v value vP value vV*

obtains vP' **where** $vP = \text{Hashed } (\text{hash } (\downarrow vP')) vP' vV = \text{Hash } (\text{hash } (\downarrow vP')) \text{value } vP'$

using *assms* **by** (*atomize_elim*, *induct* $\{\$\$ \}::\text{tyenv } vP vV \text{AuthT } \tau$ *rule*: *agree.induct*) *simp_all*

inductive_cases *a_Mu_inv[elim]*: $\Gamma \vdash e, eP, eV : \text{Mu } \alpha \tau$

inductive_cases *a_Sum_inv[elim]*: $\Gamma \vdash e, eP, eV : \text{Sum } \tau_1 \tau_2$

inductive_cases *a_Prod_inv[elim]*: $\Gamma \vdash e, eP, eV : \text{Prod } \tau_1 \tau_2$

inductive_cases *a_Fun_inv[elim]*: $\Gamma \vdash e, eP, eV : \text{Fun } \tau_1 \tau_2$

declare [*simproc* add: *alpha_lst*]

lemma *agree_weakening_1*:

assumes $\Gamma \vdash e, eP, eV : \tau \text{atom } y \# e \text{atom } y \# eP \text{atom } y \# eV$

shows $\Gamma(y \text{\$} := \tau') \vdash e, eP, eV : \tau$

using *assms* **proof** (*nominal_induct* $\Gamma e eP eV \tau$ *avoiding*: $y \tau'$ *rule*: *agree.strong_induct*)

case (*a_Lam* $x \Gamma \tau_1 e eP eV \tau_2$)

then show *?case*

by (*force simp* add: *fresh_at_base fresh_fmap_update fmupd_reorder_neq*)

next

case (*a_App* $v_1 v_2 vP_1 vP_2 vV_1 vV_2 \Gamma \tau_1 \tau_2$)

then show *?case*

using *term.fresh(9)* **by** *blast*

next

case (*a_Let* $x \Gamma e_1 eP_1 eV_1 \tau_1 e_2 eP_2 eV_2 \tau_2$)

then show *?case*

by (*auto simp* add: *fresh_at_base fresh_Pair fresh_fmap_update fmupd_reorder_neq[of x y]*
intro!: *a_Let(10) agree.a_Let[of x]*)

next

case (*a_Rec* $x \Gamma z \tau_1 \tau_2 e eP eV$)

then show *?case*

by (*auto simp* add: *fresh_at_base fresh_Pair fresh_fmap_update fmupd_reorder_neq[of x y]*
intro!: *a_Rec(9) agree.a_Rec[of x]*)

next

case (*a_Case* $v v_1 v_2 vP vP_1 vP_2 vV vV_1 vV_2 \Gamma \tau_1 \tau_2 \tau$)

then show *?case*

by (*fastforce simp*: *fresh_at_base*)

next

case (*a_Prj1* $v vP vV \Gamma \tau_1 \tau_2$)

then show *?case*

by (*fastforce simp*: *fresh_at_base*)

next

case (*a_Prj2* $v vP vV \Gamma \tau_1 \tau_2$)

then show *?case*

by (fastforce simp: fresh_at_base)
qed (auto simp: fresh_fmap_update)

lemma agree_weakening_2:

assumes $\Gamma \vdash e, eP, eV : \tau$ atom $y \# \Gamma$
shows $\Gamma(y \text{ \#\#} := \tau') \vdash e, eP, eV : \tau$

proof –

from *assms* have $\{atom\ y\} \# \{e, eP, eV\}$ by (auto simp: fresh_Pair dest: agree_fresh_env_fresh_term)
with *assms* show $\Gamma(y \text{ \#\#} := \tau') \vdash e, eP, eV : \tau$ by (simp add: agree_weakening_1)

qed

lemma agree_weakening_env:

assumes $\{\#\#\} \vdash e, eP, eV : \tau$
shows $\Gamma \vdash e, eP, eV : \tau$

using *assms* proof (induct Γ)

case *fmempty*

then show *?case* by assumption

next

case (*fmupd* $x\ y\ \Gamma$)

then show *?case*

by (simp add: fresh_tyenv_None agree_weakening_2)

qed

5 Formalization of Miller et al.'s [2] Main Results

lemma judge_imp_agree:

assumes $\Gamma \vdash e : \tau$

shows $\Gamma \vdash e, e, e : \tau$

using *assms* by (induct $\Gamma\ e\ \tau$) (auto simp: fresh_Pair)

5.1 Lemma 2.1

lemma lemma2_1:

assumes $\Gamma \vdash e, eP, eV : \tau$

shows $(\langle eP \rangle) = eV$

using *assms* by (induct $\Gamma\ e\ eP\ eV\ \tau$) (simp_all add: Abs1_eq)

5.2 Counterexample to Lemma 2.2

lemma lemma2_2_false:

fixes $x :: var$

assumes $\bigwedge \Gamma\ e\ eP\ eV\ \tau\ eP'\ eV'. \llbracket \Gamma \vdash e, eP, eV : \tau; \Gamma \vdash e, eP', eV' : \tau \rrbracket \implies eP = eP' \wedge eV = eV'$

shows *False*

proof –

have $a1: \{\#\#\} \vdash Prj1\ (Pair\ Unit\ Unit), Prj1\ (Pair\ Unit\ Unit), Prj1\ (Pair\ Unit\ Unit) : One$

by *fastforce*

also have $a2: \{\#\#\} \vdash Prj1\ (Pair\ Unit\ Unit), Prj1\ (Pair\ Unit\ (Hashed\ (hash\ Unit)\ Unit)), Prj1\ (Pair\ Unit\ (Hash\ (hash\ Unit))) : One$

by *fastforce*

from $a1\ a2$ have $Prj1\ (Pair\ Unit\ Unit) = Prj1\ (Pair\ Unit\ (Hash\ (hash\ Unit)))$

by (auto dest: *assms*)

then show *False*

by *simp*

qed

lemma smallstep_ideal_deterministic:

```

  <<[], t>> I → <<[], u>> ⇒ <<[], t>> I → <<[], u'>> ⇒ u = u'
proof (nominal_induct []::proofstream t I []::proofstream u avoiding: u' rule: smallstep.strong_induct)
  case (s_App1 e1 e1' e2)
  from s_App1(3) value_no_step[OF s_App1(1)] show ?case
  by (auto dest!: s_App1(2))
next
  case (s_App2 v1 e2 e2')
  from s_App2(4) value_no_step[OF s_App2(2)] value_no_step[OF _ s_App2(1)] show ?case
  by (force dest!: s_App2(3))
next
  case (s_AppLam v x e)
  from s_AppLam(5,1,3) value_no_step[OF _ s_AppLam(2)] show ?case
  proof (cases rule: s_App_inv)
  case (AppLam y e')
  obtain z :: var where atom z ‡ (x, e, y, e')
  by (metis obtain_fresh)
  with AppLam s_AppLam(1,3) show ?thesis
  by (auto simp: fresh_Pair intro: box_equals[OF _ subst_term_perm[symmetric, of z] subst_term_perm[symmetric, of z]])
  qed (auto dest: value_no_step)
next
  case (s_AppRec v x e)
  from s_AppRec(5,1,3) value_no_step[OF _ s_AppRec(2)] show ?case
  proof (cases rule: s_App_inv)
  case (AppRec y e')
  obtain z :: var where atom z ‡ (x, e, y, e')
  by (metis obtain_fresh)
  with AppRec(1-4) AppRec(5)[simplified] s_AppRec(1,3) show ?thesis
  apply (auto simp: fresh_Pair AppRec(1))
  apply (rule box_equals[OF _ subst_term_perm[symmetric, of z] subst_term_perm[symmetric, of z]])
  using AppRec(1) apply auto
  done
  qed (auto dest: value_no_step)
next
  case (s_Let1 x e1 e1' e2)
  from s_Let1(1,2,3,8) value_no_step[OF s_Let1(6)] show ?case
  by (auto dest: s_Let1(7))
next
  case (s_Let2 v x e)
  from s_Let2(1,3,5) value_no_step[OF _ s_Let2(2)] show ?case
  by force
next
  case (s_Inj1 e e')
  from s_Inj1(2,3) show ?case
  by auto
next
  case (s_Inj2 e e')
  from s_Inj2(2,3) show ?case
  by auto
next
  case (s_Case e e' e1 e2)
  from s_Case(2,3) value_no_step[OF s_Case(1)] show ?case
  by auto
next
  case (s_Pair1 e1 e1' e2)
  from s_Pair1(2,3) value_no_step[OF s_Pair1(1)] show ?case
  by auto

```

```

next
  case (s_Pair2 v1 e2 e2')
  from s_Pair2(3,4) value_no_step[OF s_Pair2(1)] value_no_step[OF s_Pair2(2)] show ?case
  by force
next
  case (s_Prj1 e e')
  from s_Prj1(2,3) value_no_step[OF s_Prj1(1)] show ?case
  by auto
next
  case (s_Prj2 e e')
  from s_Prj2(2,3) value_no_step[OF s_Prj2(1)] show ?case
  by auto
next
  case (s_Unroll e e')
  from s_Unroll(2,3) value_no_step[OF s_Unroll(1)] show ?case
  by auto
next
  case (s_Roll e e')
  from s_Roll(2,3) show ?case
  by auto
next
  case (s_Auth e e')
  from s_Auth(2,3) value_no_step[OF s_Auth(1)] show ?case
  by auto
next
  case (s_Unauth e e')
  from s_Unauth(2,3) value_no_step[OF s_Unauth(1)] show ?case
  by auto
qed (auto 0 3 dest: value_no_step)

```

lemma *smallsteps_ideal_deterministic*:

$\ll [], t \gg I \rightarrow i \ll [], u \gg \implies \ll [], t \gg I \rightarrow i \ll [], u' \gg \implies u = u'$

proof (*induct* []::proofstream t I i []::proofstream u arbitrary: u' rule: smallsteps.induct)

case (s_Tr e1 i π2 e2 e3)

from s_Tr(4) show ?case

proof (*cases rule: smallsteps.cases*)

case _: (s_Tr i π4 e4)

with s_Tr(1,3) s_Tr(2)[of e4] show ?thesis

using smallstepsI_ps_emptyD(2)[of e1 i π4 e4] smallstepI_ps_eq[of π2 e2 [] e3]

by (auto intro!: smallstep_ideal_deterministic elim!: smallstepI_ps_emptyD)

qed simp

qed auto

5.3 Lemma 2.3

lemma *lemma2_3*:

assumes $\Gamma \vdash e, eP, eV : \tau$

shows $\text{erase_env } \Gamma \vdash_W e : \text{erase } \tau$

using *assms unfolding erase_env_def*

proof (*nominal_induct* $\Gamma e eP eV \tau$ rule: *agree.strong_induct*)

case (a_HashI v vP τ h Γ)

then show ?case

by (*induct* Γ) (auto simp: judge_weak_weakening_2 fmmmap_fmupd judge_weak_fresh_env_fresh_term fresh_tyenv_None)

qed (simp_all add: fresh_fmmap_erase_fresh_fmmap_fmupd judge_weak_fresh_env_fresh_term)

5.4 Lemma 2.4

lemma *lemma2_4[dest]*:

assumes $\Gamma \vdash e, eP, eV : \tau$
shows $\text{value } e \wedge \text{value } eP \wedge \text{value } eV \vee \neg \text{value } e \wedge \neg \text{value } eP \wedge \neg \text{value } eV$
using *assms* **by** (*nominal_induct* $\Gamma \ e \ eP \ eV \ \tau$ *rule: agree.strong_induct*) *auto*

5.5 Lemma 3

lemma *lemma3_general*:

fixes $\Gamma :: \text{tyenv}$ **and** $vs \ vPs \ vVs :: \text{tenv}$

assumes $\Gamma \vdash e : \tau \ A \ |\subseteq| \ \text{fmdom } \Gamma$

and $\text{fmdom } vs = A \ \text{fmdom } vPs = A \ \text{fmdom } vVs = A$

and $\forall x. x \ |\in| \ A \ \longrightarrow \ (\exists \tau' \ v \ vP \ h.$

$\Gamma \ \S\ \$ \ x = \text{Some } (\text{AuthT } \tau') \wedge$

$vs \ \S\ \$ \ x = \text{Some } v \wedge$

$vPs \ \S\ \$ \ x = \text{Some } (\text{Hashed } h \ vP) \wedge$

$vVs \ \S\ \$ \ x = \text{Some } (\text{Hash } h) \wedge$

$\{\S\ \$\} \vdash v, \text{Hashed } h \ vP, \text{Hash } h : (\text{AuthT } \tau')$)

shows $\text{fmdrop_fset } A \ \Gamma \vdash \text{psubst_term } e \ vs, \text{psubst_term } e \ vPs, \text{psubst_term } e \ vVs : \tau$

using *assms*

proof (*nominal_induct* $\Gamma \ e \ \tau$ *avoiding: vs vPs vVs A rule: judge.strong_induct*)

case (*j_Unit* Γ)

then show *?case*

by *auto*

next

case (*j_Var* $\Gamma \ x \ \tau$)

with *j_Var* **show** *?case*

proof (*cases* $x \ |\in| \ A$)

case *True*

with *j_Var* **show** *?thesis*

by (*fastforce* *dest!*: *spec*[*of* x] *intro: agree_weakening_env*)

next

case *False*

with *j_Var* **show** *?thesis*

by (*auto simp* *add: a_Var dest!: fmdomI split: option.splits*)

qed

next

case (*j_Lam* $x \ \Gamma \ \tau_1 \ e \ \tau_2$)

from *j_Lam*(4) **have** [*simp*]: $A \ |\dashv| \ \{|x|\} = A$

by (*simp* *add: fresh_fset_fminus*)

from *j_Lam*(5,8-) **have** $\text{fmdrop_fset } A \ \Gamma(x \ \S\ \$:= \tau_1) \vdash \text{psubst_term } e \ vs, \text{psubst_term } e \ vPs,$
 $\text{psubst_term } e \ vVs : \tau_2$

by (*intro* *j_Lam*(7)[*of* $A \ vs \ vPs \ vVs$] (*auto simp: fresh_tyenv_None*))

with *j_Lam*(1-5) **show** *?case*

by (*auto simp: fresh_fmdrop_fset*)

next

case (*j_App* $\Gamma \ e \ \tau_1 \ \tau_2 \ e'$)

then have $\text{fmdrop_fset } A \ \Gamma \vdash \text{psubst_term } e \ vs, \text{psubst_term } e \ vPs, \text{psubst_term } e \ vVs : \text{Fun } \tau_1 \ \tau_2$

and $\text{fmdrop_fset } A \ \Gamma \vdash \text{psubst_term } e' \ vs, \text{psubst_term } e' \ vPs, \text{psubst_term } e' \ vVs : \tau_1$

by *simp_all*

then show *?case*

by *auto*

next

case (*j_Let* $x \ \Gamma \ e_1 \ \tau_1 \ e_2 \ \tau_2$)

from *j_Let*(4) **have** [*simp*]: $A \ |\dashv| \ \{|x|\} = A$

by (*simp* *add: fresh_fset_fminus*)

from *j_Let*(8,11-) **have** $\text{fmdrop_fset } A \ \Gamma \vdash \text{psubst_term } e_1 \ vs, \text{psubst_term } e_1 \ vPs, \text{psubst_term } e_1$
 $vVs : \tau_1$

by *simp*

moreover from *j_Let*(4,5,11-) **have** $\text{fmdrop_fset } A \ \Gamma(x \ \S\ \$:= \tau_1) \vdash \text{psubst_term } e_2 \ vs, \text{psubst_term}$

```

e2 vPs, psubst_term e2 vVs : τ2
  by (intro j_Let(10)) (auto simp: fresh_tyenv_None)
ultimately show ?case using j_Let(1-6)
  by (auto simp: fresh_fmdrop_fset fresh_Pair fresh_psubst)
next
case (j_Rec x Γ y τ1 τ2 e')
from j_Rec(4) have [simp]: A |-| {|x|} = A
  by (simp add: fresh_fset_fminus)
from j_Rec(9,14-) have fmdrop_fset A Γ(x $$$:= Fun τ1 τ2) ⊢ psubst_term (Lam y e') vs, psubst_term
(Lam y e') vPs, psubst_term (Lam y e') vVs : Fun τ1 τ2
  by (intro j_Rec(13)) (auto simp: fresh_tyenv_None)
with j_Rec(1-11) show ?case
  by (auto simp: fresh_fmdrop_fset)
next
case (j_Case Γ e τ1 τ2 e1 τ e2)
then have fmdrop_fset A Γ ⊢ psubst_term e vs, psubst_term e vPs, psubst_term e vVs : Sum τ1 τ2
  and fmdrop_fset A Γ ⊢ psubst_term e1 vs, psubst_term e1 vPs, psubst_term e1 vVs : Fun τ1 τ
  and fmdrop_fset A Γ ⊢ psubst_term e2 vs, psubst_term e2 vPs, psubst_term e2 vVs : Fun τ2 τ
  by simp_all
then show ?case
  by auto
next
case (j_Prj1 Γ e τ1 τ2)
then have fmdrop_fset A Γ ⊢ psubst_term e vs, psubst_term e vPs, psubst_term e vVs : Prod τ1 τ2
  by simp
then show ?case
  by auto
next
case (j_Prj2 Γ e τ1 τ2)
then have fmdrop_fset A Γ ⊢ psubst_term e vs, psubst_term e vPs, psubst_term e vVs : Prod τ1 τ2
  by simp
then show ?case
  by auto
next
case (j_Roll α Γ e τ)
then have fmdrop_fset A Γ ⊢ psubst_term e vs, psubst_term e vPs, psubst_term e vVs : subst_type τ
(Mu α τ) α
  by simp
with j_Roll(4,5) show ?case
  by (auto simp: fresh_fmdrop_fset)
next
case (j_Unroll α Γ e τ)
then have fmdrop_fset A Γ ⊢ psubst_term e vs, psubst_term e vPs, psubst_term e vVs : Mu α τ
  by simp
with j_Unroll(4,5) show ?case
  by (auto simp: fresh_fmdrop_fset)
qed auto

```

lemmas lemma3 = lemma3_general[where A = fmdom Γ and Γ = Γ, simplified] for Γ

5.6 Lemma 4

lemma lemma4:

```

assumes Γ(x $$$:= τ') ⊢ e, eP, eV : τ
and     {$$$} ⊢ v, vP, vV : τ'
and     value v value vP value vV
shows   Γ ⊢ e[v / x], eP[vP / x], eV[vV / x] : τ
using  assms

```

```

proof (nominal_induct  $\Gamma(x \text{ \#\#} := \tau')$   $e \in P \ eV \ \tau$  avoiding:  $x \ \Gamma$  rule: agree.strong_induct)
  case  $a\_Unit$ 
  then show ?case by auto
next
  case ( $a\_Var \ y \ \tau$ )
  then show ?case
  proof (induct  $\Gamma$ )
    case  $fmempty$ 
    then show ?case by (metis agree.a_Var  $fmupd\_lookup \ option.sel \ subst\_term.simps(2)$ )
  next
  case ( $fmupd \ x \ y \ \Gamma$ )
  then show ?case
    using agree_weakening_2 fresh_tyenv_None by auto
  qed
next
  case ( $a\_Lam \ y \ \tau_1 \ e \in P \ eV \ \tau_2$ )
  from  $a\_Lam(1,2,5,6,7-)$  show ?case
    using agree_empty_fresh by (auto simp:  $fmupd\_reorder\_neq$ )
  next
  case ( $a\_App \ v_1 \ v_2 \ v_1P \ v_2P \ v_1V \ v_2V \ \tau_1 \ \tau_2$ )
  from  $a\_App(5-)$  show ?case
    by (auto intro:  $a\_App(2,4)$ )
  next
  case ( $a\_Let \ y \ e_1 \ eP_1 \ eV_1 \ \tau_1 \ e_2 \ eP_2 \ eV_2 \ \tau_2$ )
  then show ?case
    using agree_empty_fresh by (auto simp:  $fmupd\_reorder\_neq \ intro!$ : agree.a_Let[where  $x=y$ ])
  next
  case ( $a\_Rec \ y \ z \ \tau_1 \ \tau_2 \ e \in P \ eV$ )
  from  $a\_Rec(10)$  have  $\forall a::var. \ atom \ a \ \# \ v \ \forall a::var. \ atom \ a \ \# \ vP \ \forall a::var. \ atom \ a \ \# \ vV$ 
    by auto
  with  $a\_Rec(1-8,10-)$  show ?case
    using  $a\_Rec(9)[OF \ fmupd\_reorder\_neq]$ 
    by (auto simp:  $fmupd\_reorder\_neq \ intro!$ : agree.a_Rec[where  $x=y$ ])
  next
  case ( $a\_Case \ v \ v_1 \ v_2 \ vP \ v_1P \ v_2P \ vV \ v_1V \ v_2V \ \tau_1 \ \tau_2 \ \tau$ )
  from  $a\_Case(7-)$  show ?case
    by (auto intro:  $a\_Case(2,4,6)$ )
  next
  case ( $a\_HashI \ v \ vP \ \tau \ h$ )
  then have  $atom \ x \ \# \ v \ atom \ x \ \# \ vP$  by auto
  with  $a\_HashI$  show ?case by auto
qed auto

```

5.7 Lemma 5: Single-Step Correctness

```

lemma lemma5:
  assumes  $\{\#\#\} \vdash e, eP, eV : \tau$ 
  and  $\ll [], e \gg I \rightarrow \ll [], e' \gg$ 
  obtains  $eP' \ eV' \ \pi$ 
  where  $\{\#\#\} \vdash e', eP', eV' : \tau \ \forall \pi_P. \ll \pi_P, eP \gg P \rightarrow \ll \pi_P @ \pi, eP' \gg \forall \pi'. \ll \pi @ \pi', eV \gg V \rightarrow$ 
 $\ll \pi', eV' \gg$ 
proof (atomize_elim, insert assms, nominal_induct  $\{\#\#\}::tyenv \ e \in P \ eV \ \tau$  avoiding:  $e'$  rule: agree.strong_induct)
  case ( $a\_App \ e_1 \ eP_1 \ eV_1 \ \tau_1 \ \tau_2 \ e_2 \ eP_2 \ eV_2$ )
  from  $a\_App(5)$  show ?case
  proof (cases rule:  $s\_App\_inv$ )
    case ( $App1 \ e_1'$ )
    with  $a\_App(2)$  obtain  $eP_1' \ eV_1' \ \pi$  where *:  $\{\#\#\} \vdash e_1', eP_1', eV_1' : Fun \ \tau_1 \ \tau_2$ 
       $\forall \pi_P. \ll \pi_P, eP_1 \gg P \rightarrow \ll \pi_P @ \pi, eP_1' \gg \forall \pi'. \ll \pi @ \pi', eV_1 \gg V \rightarrow \ll \pi', eV_1' \gg$ 

```

```

  by blast
show ?thesis
proof (intro exI conjI)
  from * App1 a_App(1,3,5-)
  show { $\$\$$ }  $\vdash e', App eP_1' eP_2, App eV_1' eV_2 : \tau_2$ 
     $\forall \pi_P. \ll \pi_P, App eP_1 eP_2 \gg P \rightarrow \ll \pi_P @ \pi, App eP_1' eP_2 \gg$ 
     $\forall \pi'. \ll \pi @ \pi', App eV_1 eV_2 \gg V \rightarrow \ll \pi', App eV_1' eV_2 \gg$ 
  by auto
qed
next
case (App2 e2')
with a_App(4) obtain eP_2' eV_2'  $\pi$  where *: { $\$\$$ }  $\vdash e_2', eP_2', eV_2' : \tau_1$ 
 $\forall \pi_P. \ll \pi_P, eP_2 \gg P \rightarrow \ll \pi_P @ \pi, eP_2' \gg \forall \pi'. \ll \pi @ \pi', eV_2 \gg V \rightarrow \ll \pi', eV_2' \gg$ 
  by blast
show ?thesis
proof (intro exI conjI)
  from * App2 a_App(1,3,5-)
  show { $\$\$$ }  $\vdash e', App eP_1 eP_2', App eV_1 eV_2' : \tau_2$ 
     $\forall \pi_P. \ll \pi_P, App eP_1 eP_2 \gg P \rightarrow \ll \pi_P @ \pi, App eP_1 eP_2' \gg$ 
     $\forall \pi'. \ll \pi @ \pi', App eV_1 eV_2 \gg V \rightarrow \ll \pi', App eV_1 eV_2' \gg$ 
  by auto
qed
next
case (AppLam x e)
from a_App(1)[unfolded  $\langle e_1 = Lam x e \rangle$ ] show ?thesis
proof (cases rule: a_Lam_inv_I[case_names _ Lam])
  case (Lam eP eV)
  show ?thesis
  proof (intro exI conjI)
    from Lam a_App(3,5) AppLam show { $\$\$$ }  $\vdash e', eP[eP_2 / x], eV[eV_2 / x] : \tau_2$ 
      by (auto intro: lemma4)
    from Lam a_App(3,5) AppLam show  $\forall \pi_P. \ll \pi_P, App eP_1 eP_2 \gg P \rightarrow \ll \pi_P @ [], eP[eP_2 / x] \gg$ 
      by (intro allI iffD1[OF smallstepP_ps_prepend[where  $\pi = [], simplified$ ]])
        (auto simp: fresh_Pair intro!: s_AppLam[where  $v=eP_2$ ])
    from Lam a_App(3,5) AppLam show  $\forall \pi'. \ll [] @ \pi', App eV_1 eV_2 \gg V \rightarrow \ll \pi', eV[eV_2 / x] \gg$ 
      by (intro allI iffD1[OF smallstepV_ps_append[where  $\pi' = [], simplified$ ]])
        (auto simp: fresh_Pair intro!: s_AppLam[where  $v=eV_2$ ])
  qed
qed simp
next
case (AppRec x e)
from a_App(1)[unfolded  $\langle e_1 = Rec x e \rangle$ ] show ?thesis
proof (cases rule: a_Rec_inv_I[consumes 1, case_names _ Rec])
  case (Rec y e'' eP' eV')
  from Rec(5,4) show ?thesis
  proof (cases rule: a_Lam_inv_I[consumes 1, case_names _ Lam])
    case (Lam eP'' eV'')
    show ?thesis
    proof (intro conjI exI[of _ []] exI)
      let ?eP = App (Lam y eP''[Rec x (Lam y eP'') / x]) eP_2
      let ?eV = App (Lam y eV''[Rec x (Lam y eV'') / x]) eV_2
      from a_App(3) AppRec have [simp]: value eP_2 atom x  $\#$  eP_2 value eV_2 atom x  $\#$  eV_2
      by (auto simp: fresh_Pair)
      from Lam a_App(3,5-) AppRec Rec show { $\$\$$ }  $\vdash e', ?eP, ?eV : \tau_2$ 
      unfolding term.eq_iff Abs1_eq(3)
      by (auto simp: fnupd_reorder_neq
        intro!: agree.a_App[where  $\Gamma=\{\$\$$ }] a_Lam[where  $x=y$ ] lemma4)
      from Lam a_App(3,5-) AppRec Rec show  $\forall \pi_P. \ll \pi_P, App eP_1 eP_2 \gg P \rightarrow \ll \pi_P @ [], ?eP \gg$ 

```

```

    unfolding term.eq_iff Abs1_eq(3)
    using s_AppRec[where v=eP2 and x=x and π=[] and e=Lam y eP'' and uv=P]
    by (intro allI iffD1[OF smallstepP_ps_prepend[of [], simplified]])
      (auto simp: fresh_Pair simp del: s_AppRec)
  from Lam a_App(3,5-) AppRec Rec show ∀π'. <<[] @ π', App eV1 eV2>> V → <<π', ?eV>>
    unfolding term.eq_iff Abs1_eq(3)
    using s_AppRec[where v=eV2 and x=x and π=[] and e=Lam y eV'' and uv=V]
    by (intro allI iffD1[OF smallstepV_ps_append[of _ _ [], simplified]])
      (auto simp: fresh_Pair simp del: s_AppRec)
  qed
  qed (simp add: fresh_fmap_update)
  qed simp
  qed
next
case (a_Let x e1 eP1 eV1 τ1 e2 eP2 eV2 τ2)
then have atom x ‡ (e1, []) by auto
with a_Let(10) show ?case
proof (cases rule: s_Let_inv)
  case Let1
  show ?thesis
  proof (intro conjI exI[of _ []] exI)
    from a_Let(6,8) Let1 show {$$} ⊢ e', eP2[eP1 / x], eV2[eV1 / x] : τ2
    by (auto intro: lemma4)
    from a_Let(4,6) Let1 show ∀πP. <<πP, Let eP1 x eP2>> P → <<πP @ [], eP2[eP1 / x]>>
    by (intro allI iffD1[OF smallstepP_ps_prepend[of [], simplified]] s_Let2) auto
    from a_Let(5,6) Let1 show ∀π'. <<[] @ π', Let eV1 x eV2>> V → <<π', eV2[eV1 / x]>>
    by (intro allI iffD1[OF smallstepV_ps_append[of _ _ [], simplified]] s_Let2) auto
  qed
next
case (Let2 e1')
moreover
from Let2 a_Let(7) obtain eP1' eV1' π
  where ih: {$$} ⊢ e1', eP1', eV1' : τ1
    ∀πP. <<πP, eP1>> P → <<πP @ π, eP1'>>
    ∀π'. <<π @ π', eV1>> V → <<π', eV1'>>
  by (blast dest: spec[of _ []])
then have [simp]: atom x ‡ ({$$}, e1', eP1', eV1')
  using agree_empty_fresh by auto
from ih a_Let(4) have [simp]: atom x ‡ π
  using fresh_Nil fresh_append fresh_ps_smallstep_P by blast
from a_Let ih have agree: {$$} ⊢ Let e1' x e2, Let eP1' x eP2, Let eV1' x eV2 : τ2
  by auto
moreover from a_Let(4,5) ih(1) spec[OF ih(2), of [], simplified]
have <<π', Let eP1 x eP2>> P → <<π' @ π, Let eP1' x eP2>> for π'
  by (intro iffD1[OF smallstepP_ps_prepend[of [], simplified]] s_Let1) (auto simp: fresh_Pair)
moreover from a_Let(4,5) ih(1) spec[OF ih(3), of [], simplified]
have <<π @ π', Let eV1 x eV2>> V → <<π', Let eV1' x eV2>> for π'
  by (intro iffD1[OF smallstepV_ps_append[of π _ [], simplified]] s_Let1) (auto simp: fresh_Pair)
ultimately show ?thesis
  by blast
qed
next
case (a_Case e eP eV τ1 τ2 e1 eP1 eV1 τ e2 eP2 eV2)
from a_Case(7) show ?case
proof (cases rule: s_Case_inv)
  case (Case e')
  with a_Case(2)[of e'] obtain eP'' eV'' π where ih: {$$} ⊢ e'', eP'', eV'' : Syntax.Sum τ1 τ2
    ∀πP. <<πP, eP>> P → <<πP @ π, eP''>> ∀π'. <<π @ π', eV>> V → <<π', eV''>>

```

```

    by blast
  show ?thesis
  proof (intro conjI exI[of _  $\pi$ ] exI)
    from ih(1) a_Case(3,5) Case show  $\{\$\$ \} \vdash e', \text{Case } eP'' eP_1 eP_2, \text{Case } eV'' eV_1 eV_2 : \tau$ 
      by auto
    from a_Case(5) spec[OF ih(2), of [], simplified]
    show  $\forall \pi_P. \ll \pi_P, \text{Case } eP eP_1 eP_2 \gg P \rightarrow \ll \pi_P @ \pi, \text{Case } eP'' eP_1 eP_2 \gg$ 
      by (intro allI iffD1[OF smallstepP_ps_prepend[of [], simplified]] s_Case) auto
    from a_Case(5) spec[OF ih(3), of [], simplified]
    show  $\forall \pi'. \ll \pi @ \pi', \text{Case } eV eV_1 eV_2 \gg V \rightarrow \ll \pi', \text{Case } eV'' eV_1 eV_2 \gg$ 
      by (intro allI iffD1[OF smallstepV_ps_append[of _ _ [], simplified]] s_Case) auto
  qed
next
case (Inj1 v)
from a_Case(1)[unfolded  $\langle e = \text{Inj1 } v \rangle$ ] show ?thesis
proof (cases rule: a_Inj1_inv_I[consumes 1, case_names Case])
  case (Case vP vV)
  with a_Case(3,5) Inj1 show ?thesis
  proof (intro conjI exI[of _ []] exI)
    from Case a_Case(3,5) Inj1 show  $\{\$\$ \} \vdash e', \text{App } eP_1 vP, \text{App } eV_1 vV : \tau$ 
      by auto
  qed auto
qed
next
case (Inj2 v)
from a_Case(1)[unfolded  $\langle e = \text{Inj2 } v \rangle$ ] show ?thesis
proof (cases rule: a_Inj2_inv_I[consumes 1, case_names Case])
  case (Case vP vV)
  with a_Case(3,5) Inj2 show ?thesis
  proof (intro conjI exI[of _ []] exI)
    from Case a_Case(3,5) Inj2 show  $\{\$\$ \} \vdash e', \text{App } eP_2 vP, \text{App } eV_2 vV : \tau$ 
      by auto
  qed auto
qed
next
case (a_Prj1 e eP eV  $\tau_1 \tau_2$ )
from a_Prj1(3) show ?case
proof (cases rule: s_Prj1_inv)
  case (Prj1 e')
  then show ?thesis
  by (blast dest!: a_Prj1(2))
next
case (PrjPair1 v2)
from a_Prj1(1)[unfolded  $\langle e = \text{Syntax.Pair } e' v_2 \rangle$ ] show ?thesis
proof (cases rule: a_Pair_inv_I[consumes 1, case_names Pair])
  case (Pair eP1 eV1 eP2 eV2)
  with PrjPair1 show ?thesis
  proof (intro conjI exI[of _ []] exI)
    show  $\{\$\$ \} \vdash e', eP_1, eV_1 : \tau_1$ 
      by (rule Pair)
  qed auto
qed
next
case (a_Prj2 e eP eV  $\tau_1 \tau_2$ )
from a_Prj2(3) show ?case
proof (cases rule: s_Prj2_inv)

```

```

case (Prj2 e'')
then show ?thesis
  by (blast dest!: a_Prj2(2))
next
case (PrjPair2 v1)
from a_Prj2(1)[unfolded ⟨e = Syntax.Pair v1 e'⟩] show ?thesis
proof (cases rule: a_Pair_inv_I[consumes 1, case_names Pair])
  case (Pair eP1 eV1 eP2 eV2)
  with PrjPair2 show ?thesis
  proof (intro conjI exI[of _ []] exI)
    show {$$} ⊢ e', eP2, eV2 : τ2
    by (rule Pair)
  qed auto
qed
qed
next
case (a_Roll α e eP eV τ)
from a_Roll(5) show ?case
proof (cases rule: s_Roll_inv)
  case (Roll e'')
  with a_Roll(4) obtain eP'' eV'' π where *: {$$} ⊢ e'', eP'', eV'' : subst_type τ (Mu α τ) α
    ∀ πP. ⟨⟨πP, eP⟩⟩ P → ⟨⟨πP @ π, eP''⟩⟩ ∀ π'. ⟨⟨π @ π', eV⟩⟩ V → ⟨⟨π', eV''⟩⟩
    by blast
  show ?thesis
  proof (intro exI conjI)
    from * Roll
    show {$$} ⊢ e', Roll eP'', Roll eV'' : Mu α τ
      ∀ πP. ⟨⟨πP, Roll eP⟩⟩ P → ⟨⟨πP @ π, Roll eP''⟩⟩
      ∀ π'. ⟨⟨π @ π', Roll eV⟩⟩ V → ⟨⟨π', Roll eV''⟩⟩
    by auto
  qed
qed
next
case (a_Unroll α e eP eV τ)
from a_Unroll(5) show ?case
proof (cases rule: s_Unroll_inv)
  case (Unroll e'')
  with a_Unroll(4) obtain eP'' eV'' π where *: {$$} ⊢ e'', eP'', eV'' : Mu α τ
    ∀ πP. ⟨⟨πP, eP⟩⟩ P → ⟨⟨πP @ π, eP''⟩⟩ ∀ π'. ⟨⟨π @ π', eV⟩⟩ V → ⟨⟨π', eV''⟩⟩
    by blast
  show ?thesis
  proof (intro exI conjI)
    from * Unroll
    show {$$} ⊢ e', Unroll eP'', Unroll eV'' : subst_type τ (Mu α τ) α
      ∀ πP. ⟨⟨πP, Unroll eP⟩⟩ P → ⟨⟨πP @ π, Unroll eP''⟩⟩
      ∀ π'. ⟨⟨π @ π', Unroll eV⟩⟩ V → ⟨⟨π', Unroll eV''⟩⟩
    by auto
  qed
qed
next
case UnrollRoll
with a_Unroll(3)[unfolded ⟨e = Roll e'⟩] show ?thesis
proof (cases rule: a_Roll_inv_I[case_names Roll])
  case (Roll eP' eV')
  with UnrollRoll show ?thesis
  proof (intro conjI exI[of _ []] exI)
    show {$$} ⊢ e', eP', eV' : subst_type τ (Mu α τ) α by fact
  qed auto
qed

```

```

qed
next
case (a_Auth e eP eV τ)
from a_Auth(1) have [simp]: atom x # eP for x :: var
using agree_empty_fresh by simp
from a_Auth(3) show ?case
proof (cases rule: s_AuthI_inv)
case (Auth e'')
then show ?thesis
by (blast dest!: a_Auth(2))
next
case AuthI
with a_Auth(1) have value eP value eV by auto
with a_Auth(1) AuthI(2) show ?thesis
proof (intro conjI exI[of _ []] exI)
from a_Auth(1) AuthI(2) ⟨value eP⟩
show {$$} ⊢ e', Hashed (hash (⟦eP⟧)) eP, Hash (hash (⟦eP⟧)) : AuthT τ
by (auto dest: lemma2_1 simp: fresh_shallow)
qed (auto dest: lemma2_1 simp: fresh_shallow)
qed
next
case (a_Unauth e eP eV τ)
from a_Unauth(1) have eP_closed: closed eP
using agree_empty_fresh by simp
from a_Unauth(3) show ?case
proof (cases rule: s_UnauthI_inv)
case (Unauth e')
then show ?thesis
by (blast dest!: a_Unauth(2))
next
case UnauthI
with a_Unauth(1) have value eP value eV by auto
from a_Unauth(1) show ?thesis
proof (cases rule: a_AuthT_value_inv[case_names _ _ _ Unauth])
case (Unauth vP')
show ?thesis
proof (intro conjI exI[of _ [⟦vP'⟧]] exI)
from Unauth(1,2) UnauthI(2) a_Unauth(1)
show {$$} ⊢ e', vP', (⟦vP'⟧) : τ
by (auto simp: fresh_shallow)
then have closed vP'
by auto
with Unauth(1,2) a_Unauth(1) show
∀ π_P. <<π_P, Unauth eP>> P → <<π_P @ [⟦vP'⟧], vP'>>
∀ π'. <<[⟦vP'⟧] @ π', Unauth eV>> V → <<π', (⟦vP'⟧)>>
by (auto simp: fresh_shallow)
qed
qed (auto simp: ⟨value e⟩ ⟨value eP⟩ ⟨value eV⟩)
qed
next
case (a_Pair e1 eP1 eV1 τ1 e2 eP2 eV2 τ2)
from a_Pair(5) show ?case
proof (cases rule: s_Pair_inv)
case (Pair1 e1')
with a_Pair(1,3) show ?thesis
by (blast dest!: a_Pair(2))
next
case (Pair2 e2')

```

```

with a_Pair(1,3) show ?thesis
  by (blast dest!: a_Pair(4))
qed
next
case (a_Inj1 e eP eV τ1 τ2)
from a_Inj1(3) show ?case
proof (cases rule: s_Inj1_inv)
  case (Inj1 e')
  with a_Inj1(1) show ?thesis
  by (blast dest!: a_Inj1(2))
qed
next
case (a_Inj2 e eP eV τ2 τ1)
from a_Inj2(3) show ?case
proof (cases rule: s_Inj2_inv)
  case (Inj2 e'')
  with a_Inj2(1) show ?thesis
  by (blast dest!: a_Inj2(2))
qed
qed (simp, meson value.intros value_no_step)+

```

5.8 Lemma 6: Single-Step Security

```

lemma lemma6:
  assumes {$$} ⊢ e, eP, eV : τ
  and    << πA, eV >> V → << π', eV' >>
  obtains e' eP' π
  where << [], e >> I → << [], e' >> ∀ πP. << πP, eP >> P → << πP @ π, eP' >>
  and    {$$} ⊢ e', eP', eV' : τ ∧ πA = π @ π' ∨
        (∃ s s'. closed s ∧ closed s' ∧ π = [s] @ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s')
proof (atomize_elim, insert assms, nominal_induct {$$}::tyenv e eP eV τ avoiding: πA π' eV' rule:
agree.strong_induct)
  case (a_App e1 eP1 eV1 τ1 τ2 e2 eP2 eV2)
  from a_App(5) show ?case
  proof (cases rule: s_App_inv)
    case (App1 eV1')
    with a_App(1,3) show ?thesis
    by (blast dest!: a_App(2))
  next
  case (App2 e2')
  with a_App(1,3) show ?thesis
  by (blast dest!: a_App(4))
  next
  case (AppLam x eV'')
  from a_App(1)[unfolded ⟨eV1 = Lam x eV''⟩] show ?thesis
  proof (cases rule: a_Lam_inv_V[case_names Lam])
    case (Lam e'' eP'')
    with a_App(3) AppLam show ?thesis
    proof (intro conjI exI[of _ []] exI disjI1)
      from Lam a_App(3) AppLam show {$$} ⊢ e''[e2 / x], eP''[eP2 / x], eV' : τ2
      by (auto intro: lemma4)
    qed (auto 0 3 simp: fresh_Pair intro!: s_AppLam[where π=[]]
intro: iffD1[OF smallstepP_ps_prepend[of [] _ [], simplified]])
  qed
  next
  case (AppRec x eV'')
  from a_App(1)[unfolded ⟨eV1 = Rec x eV''⟩] show ?thesis
  proof (cases rule: a_Rec_inv_V[case_names _ Rec])

```

```

case (Rec y e''' eP''' eV''')
with a_App(1,3) AppRec show ?thesis
proof (intro conjI exI[of _ []] exI disjI1)
  let ?e = App (Lam y e'''[Rec x (Lam y e''') / x]) e2
  let ?eP = App (Lam y eP'''[Rec x (Lam y eP''') / x]) eP2
  from Rec a_App(3) AppRec show {$$} ⊢ ?e, ?eP, eV' : τ2
  by (auto simp del: subst_term.simps(3) intro!: agree.a_App[where Γ={$$} lemma4])
qed (auto 0 3 simp del: subst_term.simps(3) simp: fresh_Pair intro!: s_AppRec[where π=[]]
  intro: iffD1[OF smallstepP_ps_prepend[of [] _ [], simplified]])
qed simp
qed
next
case (a_Let x e1 eP1 eV1 τ1 e2 eP2 eV2 τ2)
then have atom x ‡ (eV1, πA) by auto
with a_Let(12) show ?case
proof (cases rule: s_Let_inv)
  case Let1
  with a_Let(5,6,8,10) show ?thesis
  proof (intro conjI exI[of _ []] exI disjI1)
    from Let1 a_Let(5,6,8,10) show {$$} ⊢ e2[e1 / x], eP2[eP1 / x], eV' : τ2
    by (auto intro: lemma4)
  qed (auto 0 3 intro: iffD1[OF smallstepP_ps_prepend[of [] _ [], simplified]])
next
case (Let2 eV1')
with a_Let(9)[of πA π' eV1'] obtain e1' π eP1' s s' where
  ih: <<[], e1>> I → <<[], e1'>> ∀ πP. <<πP, eP1>> P → <<πP @ π, eP1'>>
  {$$} ⊢ e1', eP1', eV1' : τ1 ∧ πA = π @ π' ∨
  closed s ∧ closed s' ∧ π = [s] ∧ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s'
  by blast
with a_Let(5,6) have fresh: atom x ‡ e1' atom x ‡ eP1'
  using fresh_smallstep_I fresh_smallstep_P by blast+
from ih a_Let(2,6) have atom x ‡ π
  using fresh_append fresh_ps_smallstep_P by blast
with Let2 a_Let(1-8,10,12-) fresh ih show ?thesis
proof (intro conjI exI[of _ π] exI)
  from ⟨atom x ‡ π⟩ Let2 a_Let(1-8,10,12-) fresh ih
  show {$$} ⊢ Let e1' x e2, Let eP1' x eP2, eV' : τ2 ∧ πA = π @ π' ∨
  (∃ s s'. closed s ∧ closed s' ∧ π = [s] ∧ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s')
  by auto
qed (auto dest: spec[of _ []] intro!: iffD1[OF smallstepP_ps_prepend, of [], simplified])
qed
next
case (a_Case e eP eV τ1 τ2 e1 eP1 eV1 τ e2 eP2 eV2)
with a_Case(7) show ?case
proof (cases rule: s_Case_inv)
  case (Case eV'')
  from a_Case(2)[OF Case(2)] show ?thesis
  proof (elim exE disjE conjE, goal_cases ok collision)
    case (ok e'' π eP'')
    with Case a_Case(3,5) show ?case by blast
  next
  case (collision e'' π eP'' s s')
  with Case a_Case(3,5) show ?case
  proof (intro exI[of _ [s]] exI conjI disjI2)
    from Case a_Case(3,5) collision show <<[], Case e e1 e2>> I → <<[], Case e'' e1 e2>>
      ∀ πP. <<πP, Case eP eP1 eP2>> P → <<πP @ [s], Case eP'' eP1 eP2>>
      by auto
    from collision show closed s closed s' s ≠ s' hash s = hash s' by auto
  qed
end
end

```

```

    qed simp
  qed
next
case (Inj1 vV)
from a_Case(1)[unfolded ⟨eV = Inj1 vV⟩] show ?thesis
proof (cases rule: a_Inj1_inv_V[consumes 1, case_names Inj])
  case (Inj v' vP')
  with Inj1 show ?thesis
  proof (intro conjI exI[of _ []] exI disjI1)
    from a_Case(3) Inj Inj1 show {$$} ⊢ App e1 v', App eP1 vP', eV' : τ
    by auto
  qed auto
  qed
next
case (Inj2 vV)
from a_Case(1)[unfolded ⟨eV = Inj2 vV⟩] show ?thesis
proof (cases rule: a_Inj2_inv_V[consumes 1, case_names Inj])
  case (Inj v' vP')
  with Inj2 show ?thesis
  proof (intro conjI exI[of _ []] exI disjI1)
    from a_Case(5) Inj Inj2 show {$$} ⊢ App e2 v', App eP2 vP', eV' : τ
    by auto
  qed auto
  qed
next
case (a_Prj1 e eP eV τ1 τ2)
from a_Prj1(3) show ?case
proof (cases rule: s_Prj1_inv)
  case (Prj1 eV'')
  then show ?thesis
  by (blast dest!: a_Prj1(2))
next
case (PrjPair1 v2)
with a_Prj1(1) show ?thesis
proof (cases rule: a_Prod_inv[consumes 1, case_names _ _ _ _ Pair])
  case (Pair e1 eP1 eV1 e2 eP2 eV2)
  with PrjPair1 a_Prj1(1) show ?thesis
  proof (intro conjI exI[of _ []] exI disjI1)
    from Pair PrjPair1 a_Prj1(1) show {$$} ⊢ e1, eP1, eV' : τ1
    by auto
  qed auto
  qed simp_all
next
case (a_Prj2 e eP eV τ1 τ2)
from a_Prj2(3) show ?case
proof (cases rule: s_Prj2_inv)
  case (Prj2 eV'')
  then show ?thesis
  by (blast dest!: a_Prj2(2))
next
case (PrjPair2 v2)
with a_Prj2(1) show ?thesis
proof (cases rule: a_Prod_inv[consumes 1, case_names _ _ _ _ Pair])
  case (Pair e1 eP1 eV1 e2 eP2 eV2)
  with PrjPair2 a_Prj2(1) show ?thesis
  proof (intro conjI exI[of _ []] exI disjI1)

```

```

    from Pair PrjPair2 a_Prj2(1) show {$$} ⊢ e2, eP2, eV' : τ2
      by auto
    qed auto
  qed simp_all
qed
next
case (a_Roll α e eP eV τ)
from a_Roll(7) show ?case
proof (cases rule: s_Roll_inv)
  case (Roll eV'')
  from a_Roll(6)[OF Roll(2)] obtain e'' π eP'' where ih:
    <<[], e>> I → <<[], e''>> ∀ πP. <<πP, eP>> P → <<πP @ π, eP''>>
    {$$} ⊢ e'', eP'', eV'' : subst_type τ (Mu α τ) α ∧ πA = π @ π' ∨
    (∃ s s'. closed s ∧ closed s' ∧ π = [s] ∧ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s')
  by blast
  with Roll show ?thesis
  proof (intro exI[of _ π] exI conjI)
    from ih Roll show {$$} ⊢ Roll e'', Roll eP'', eV' : Mu α τ ∧ πA = π @ π' ∨
      (∃ s s'. closed s ∧ closed s' ∧ π = [s] ∧ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s')
    by auto
  qed auto
  qed
next
case (a_Unroll α e eP eV τ)
from a_Unroll(7) show ?case
proof (cases rule: s_Unroll_inv)
  case (Unroll eV'')
  from a_Unroll(6)[OF Unroll(2)] obtain e'' π eP'' where ih:
    <<[], e>> I → <<[], e''>> ∀ πP. <<πP, eP>> P → <<πP @ π, eP''>>
    {$$} ⊢ e'', eP'', eV'' : Mu α τ ∧ πA = π @ π' ∨
    (∃ s s'. closed s ∧ closed s' ∧ π = [s] ∧ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s')
  by blast
  with Unroll show ?thesis
  proof (intro exI[of _ π] exI conjI)
    from ih Unroll show {$$} ⊢ Unroll e'', Unroll eP'', eV' : subst_type τ (Mu α τ) α ∧ πA = π @
    π' ∨
    (∃ s s'. closed s ∧ closed s' ∧ π = [s] ∧ πA = [s'] @ π' ∧ s ≠ s' ∧ hash s = hash s')
    by auto
  qed auto
  qed
next
case UnrollRoll
  with a_Unroll(5) show ?thesis
  by fastforce
qed
next
case (a_Auth e eP eV τ)
from a_Auth(1) have [simp]: atom x ‡ eP for x :: var
  using agree_empty_fresh by simp
from a_Auth(3) show ?case
proof (cases rule: s_AuthV_inv)
  case (Auth eV'')
  from a_Auth(2)[OF Auth(2)] show ?thesis
  proof (elim exE disjE conjE, goal_cases ok collision)
    case (ok e'' π eP'')
    with Auth show ?case
    proof (intro conjI exI[of _ π] exI disjI1)
      from ok Auth show {$$} ⊢ Auth e'', Auth eP'', eV' : AuthT τ
      by auto
    end
  end
end
end
end

```

```

    qed auto
  next
    case (collision e''  $\pi$  eP'' s s')
    then show ?case by blast
  qed
next
case AuthV
with a_Auth(1) show ?thesis
proof (intro exI[of _ []] exI conjI disjI1)
  from a_Auth(1) AuthV show { $\$\$$ }  $\vdash$  e, Hashed (hash ( $\downarrow$ eP)) eP, eV' : AuthT  $\tau$ 
  by (auto dest: lemma2_1)
qed (auto simp: fresh_shallow)
qed
next
case (a_Unauth e eP eV  $\tau$ )
from a_Unauth(3) show ?case
proof (cases rule: s_UnauthV_inv)
  case (Unauth e')
  then show ?thesis
  by (blast dest!: a_Unauth(2))
next
case UnauthV
from a_Unauth(1)[unfolded  $\langle$ eV = Hash (hash eV') $\rangle$ ] UnauthV a_Unauth(1) show ?thesis
proof (cases rule: a_AuthT_value_inv[consumes 1, case_names _ _ _ Hashed])
  case (Hashed vP')
  with UnauthV a_Unauth(1) show ?thesis
  proof (intro exI[of _ [ $\downarrow$ vP']]) exI conjI
    from Hashed UnauthV a_Unauth(1) show { $\$\$$ }  $\vdash$  e, vP', eV' :  $\tau \wedge \pi_A = [ $\downarrow$ vP']$ ] @  $\pi' \vee$ 
      ( $\exists$  s s'. closed s  $\wedge$  closed s'  $\wedge$  [ $\downarrow$ vP']) = [s]  $\wedge$   $\pi_A = [s']$  @  $\pi' \wedge s \neq s' \wedge$  hash s = hash s')
    by (fastforce elim: a_HashI_inv[where  $\Gamma = \{\$\$ \}$ ])
  qed auto
qed auto
qed
next
case (a_Pair e1 eP1 eV1  $\tau_1$  e2 eP2 eV2  $\tau_2$ )
from a_Pair(5) show ?case
proof (cases rule: s_Pair_inv)
  case (Pair1 eV1')
  with a_Pair(3) show ?thesis
  using a_Pair(2)[of  $\pi_A$   $\pi'$  eV1'] by blast
next
case (Pair2 eV2')
  with a_Pair(1) show ?thesis
  using a_Pair(4)[of  $\pi_A$   $\pi'$  eV2'] by blast
qed
next
case (a_Inj1 e eP eV  $\tau_1$   $\tau_2$ )
from a_Inj1(3) show ?case
proof (cases rule: s_Inj1_inv)
  case (Inj1 eV'')
  then show ?thesis
  using a_Inj1(2)[of  $\pi_A$   $\pi'$  eV''] by blast
qed
next
case (a_Inj2 e eP eV  $\tau_2$   $\tau_1$ )
from a_Inj2(3) show ?case
proof (cases rule: s_Inj2_inv)
  case (Inj2 eV'')

```

```

with a_Inj2(1) show ?thesis
  using a_Inj2(2)[of  $\pi_A \pi' eV'$ ] by blast
qed
qed (simp, meson value.intros value_no_step)+

```

5.9 Theorem 1: Correctness

```

lemma theorem1_correctness:
  assumes  $\{\$\$ \} \vdash e, eP, eV : \tau$ 
  and  $\ll [], e \gg I \rightarrow i \ll [], e' \gg$ 
  obtains  $eP' eV' \pi$ 
  where  $\ll [], eP \gg P \rightarrow i \ll \pi, eP' \gg$ 
     $\ll \pi, eV \gg V \rightarrow i \ll [], eV' \gg$ 
     $\{\$\$ \} \vdash e', eP', eV' : \tau$ 
  using assms(2,1)
proof (atomize_elim, induct  $[\cdot]::\text{proofstream } e \text{ I } i \ [\cdot]::\text{proofstream } e' \text{ rule: smallsteps.induct}$ )
  case (s_Id e)
  then show ?case by auto
next
  case (s_Tr e1 i  $\pi_2$  e2 e3)
  then have  $\pi_2 = []$  by (auto dest: smallstepI_ps_eq)
  with s_Tr obtain  $eP_2 \pi eV_2$  where ih:
     $\ll [], eP \gg P \rightarrow i \ll \pi, eP_2 \gg \ll \pi, eV \gg V \rightarrow i \ll [], eV_2 \gg \{\$\$ \} \vdash e_2, eP_2, eV_2 : \tau$ 
  by (atomize_elim, intro s_Tr(2)) auto
  moreover obtain  $eP_3 eV_3 \pi'$  where agree:  $\{\$\$ \} \vdash e_3, eP_3, eV_3 : \tau$ 
  and  $\ll \pi_P, eP_2 \gg P \rightarrow \ll \pi_P @ \pi', eP_3 \gg \ll \pi' @ \pi'', eV_2 \gg V \rightarrow \ll \pi'', eV_3 \gg$ 
  for  $\pi_P \pi''$  using lemma5[OF ih(3) s_Tr(3)[unfolded  $\langle \pi_2 = [] \rangle$ ], of thesis] by blast
  ultimately have  $\ll [], eP \gg P \rightarrow i + 1 \ll \pi @ \pi', eP_3 \gg \ll \pi @ \pi', eV \gg V \rightarrow i + 1 \ll [], eV_3 \gg$ 
  by (auto simp: smallstepsV_ps_append[of  $-\ - \ - \ []$ , simplified, symmetric]
    intro!: smallsteps.s_Tr[of  $\pi @ \pi'$ ])
  with agree show ?case by blast
qed

```

5.10 Counterexamples to Theorem 1: Security

Counterexample using administrative normal form.

```

lemma security_false:
  assumes agree:  $\bigwedge e eP eV \tau \pi_A i \pi' eV'. [\{\$\$ \} \vdash e, eP, eV : \tau; \ll \pi_A, eV \gg V \rightarrow i \ll \pi', eV' \gg]$ 
   $\implies$ 
     $\exists e' eP' \pi j \pi_0 s s'. (\ll [], e \gg I \rightarrow i \ll [], e' \gg \wedge \ll [], eP \gg P \rightarrow i \ll \pi, eP' \gg \wedge (\pi_A = \pi @ \pi'))$ 
   $\wedge \{\$\$ \} \vdash e', eP', eV' : \tau) \vee$ 
     $(j \leq i \wedge \ll [], eP \gg P \rightarrow j \ll \pi_0 @ [s], eP' \gg \wedge (\pi_A = \pi_0 @ [s'] @ \pi') \wedge s \neq s' \wedge \text{hash } s = \text{hash } s')$ 
  and collision:  $\text{hash } (\text{Inj1 Unit}) = \text{hash } (\text{Inj2 Unit})$ 
  and no_collision_with_Unit:  $\bigwedge t. \text{hash } \text{Unit} = \text{hash } t \implies t = \text{Unit}$ 
  shows False
proof -
  define i where  $i = \text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } 0))))))$ 
  obtain  $x y z :: \text{var}$  where fresh:  $\text{atom } y \# x \text{ atom } z \# (x, y)$ 
  by (metis obtain_fresh)
  define t where  $t = \text{Let } (\text{Let } (\text{Auth } (\text{Inj1 Unit})) y (\text{Unauth } (\text{Var } y))) x (\text{Let } (\text{Let } (\text{Auth } \text{Unit}) z (\text{Unauth } (\text{Var } z))) y (\text{Var } x))$ 
  note fresh_Cons[simp]
  have prover:  $\ll [], t \gg P \rightarrow i \ll [\text{Inj1 Unit}, \text{Unit}], \text{Inj1 Unit} \gg$  — prover execution
  unfolding i_def t_def Suc_eq_plus1 using fresh
  apply -
  apply (rule smallsteps.intros)+
  apply (rule s_Let1[rotated])

```

```

    apply (rule s_Let1[rotated])
    apply (rule s_AuthP[rotated])
    apply simp
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_Let2)
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_UnauthP)
    apply simp
    apply simp
    apply simp
    apply (rule s_Let2)
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_Let1[rotated])
    apply (rule s_AuthP[rotated])
    apply simp
    apply simp
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_Let2)
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_UnauthP)
    apply simp
    apply simp
    apply simp
    apply (rule s_Let2[of Unit y _ Inj1 Unit, simplified])
    apply simp
done
have verifier1: << [Inj1 Unit, Unit], t >> V→i << [], Inj1 Unit >> — verifier execution
unfolding i_def t_def Suc_eq_plus1 using fresh
apply -
apply (rule smallsteps.intros)+
  apply (rule s_Let1[rotated])
  apply (rule s_Let1[rotated])
  apply (rule s_AuthV[rotated])
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_Let2)
  apply simp
  apply simp

```

```

    apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthV)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let2)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_Let1[rotated])
  apply (rule s_AuthV[rotated])
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_Let2)
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthV)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let2[of Unit y _ Inj1 Unit, simplified])
  apply simp
done

```

have verifier2: $\ll [Inj2\ Unit, Unit], t \gg V \rightarrow i \ll [], Inj2\ Unit \gg$ — verifier execution with proofstream containing collision

```

unfolding i_def t_def Suc_eq_plus1 using fresh
apply -
apply (rule smallsteps.intros)+
  apply (rule s_Let1[rotated])
  apply (rule s_Let1[rotated])
  apply (rule s_AuthV[rotated])
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_Let2)
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthV)
  apply simp
  apply (simp add: collision)
  apply simp
  apply (rule s_Let2)
  apply simp
  apply simp

```

```

apply simp
apply (rule s_Let1[rotated])
apply (rule s_Let1[rotated])
apply (rule s_AuthV[rotated])
apply simp
apply simp
apply simp
apply simp
apply (rule s_Let1[rotated])
apply (rule s_Let2)
apply simp
apply simp
apply simp
apply simp
apply simp
apply (rule s_Let1[rotated])
apply (rule s_UnauthV)
apply simp
apply simp
apply simp
apply (rule s_Let2[of Unit y _ Inj2 Unit, simplified])
apply simp
done
have judge: { $\$$ }  $\vdash$  t : Sum One One
unfolding t_def using fresh
by (force simp: fresh_Pair fresh_fmap_update)
have ideal_deterministic: e = Inj1 Unit if  $\ll[]$ ,  $t \gg I \rightarrow i \ll[]$ ,  $e \gg$  for e
apply (rule smallsteps_ideal_deterministic[OF that])
unfolding i_def t_def Suc_eq_plus1 using fresh
apply -
apply (rule smallsteps.intros)+
apply (rule s_Let1[rotated])
apply (rule s_Let1[rotated])
apply (rule s_AuthI[rotated])
apply simp
apply simp
apply simp
apply (rule s_Let1[rotated])
apply (rule s_Let2)
apply simp
apply simp
apply simp
apply (rule s_Let1[rotated])
apply (rule s_UnauthI)
apply simp
apply simp
apply (rule s_Let2)
apply simp
apply simp
apply simp
apply (rule s_Let1[rotated])
apply (rule s_Let1[rotated])
apply (rule s_AuthI[rotated])
apply simp
apply simp
apply (rule s_Let1[rotated])
apply (rule s_Let2)

```

```

    apply simp
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_UnauthI)
    apply simp
    apply simp
    apply (rule s_Let2[of Unit y _ Inj1 Unit, simplified])
    apply simp
  done
from agree[OF judge_imp_agree[OF judge] verifier2] collision prover verifier1 show False
proof safe
  fix e' eP'
  assume agree: { $\$$ }  $\vdash$  e', eP', Inj2 Unit : Sum One One
  assume assm:  $\ll \[], t \gg I \rightarrow i \ll \[], e' \gg$ 
  then have e' = Inj1 Unit
    by (simp add: ideal_deterministic)
  with agree show False
    by auto
qed (auto dest!: no_collision_with_Unit[OF sym])
qed

```

Alternative, shorter counterexample not in administrative normal form.

```

lemma security_false_alt:
  assumes agree:  $\bigwedge e \ eP \ eV \ \tau \ \pi A \ i \ \pi' \ eV'. \ [ \{ \$ \} \vdash e, eP, eV : \tau; \ll \pi A, eV \gg V \rightarrow i \ll \pi', eV' \gg ]$ 
   $\implies$ 
   $\exists e' \ eP' \ \pi \ j \ \pi 0 \ s \ s'. (\ll \[], e \gg I \rightarrow i \ll \[], e' \gg \wedge \ll \[], eP \gg P \rightarrow i \ll \pi, eP' \gg \wedge (\pi A = \pi @ \pi')$ 
 $\wedge \{ \$ \} \vdash e', eP', eV' : \tau) \vee$ 
   $(j \leq i \wedge \ll \[], eP \gg P \rightarrow j \ll \pi 0 @ [s], eP' \gg \wedge (\pi A = \pi 0 @ [s'] @ \pi') \wedge s \neq s' \wedge \text{hash } s = \text{hash } s')$ 
  and collision:  $\text{hash } (\text{Inj1 Unit}) = \text{hash } (\text{Inj2 Unit})$ 
  and no_collision_with_Unit:  $\bigwedge t. \text{hash } \text{Unit} = \text{hash } t \implies t = \text{Unit}$ 
  shows False
proof -
  define i where i = Suc (Suc (Suc (Suc (Suc (Suc 0))))))
  obtain x y z :: var where fresh: atom y  $\#$  x atom z  $\#$  (x, y)
  by (metis obtain_fresh)
  define t where t = Let (Unauth (Auth (Inj1 Unit))) x (Let (Unauth (Auth Unit)) y (Var x))
  note fresh_Cons[simp]
  have prover:  $\ll \[], t \gg P \rightarrow i \ll [\text{Inj1 Unit}, \text{Unit}], \text{Inj1 Unit} \gg$  — prover execution
  unfolding i_def t_def Suc_eq_plus1 using fresh
  apply -
  apply (rule smallsteps.intros)+
  apply (rule s_Let1[rotated])
  apply (rule s_Unauth)
  apply (rule s_AuthP[rotated])
  apply simp
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthP)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let2)
  apply simp

```

```

    apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_Unauth)
  apply (rule s_AuthP[rotated])
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthP)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let2[of Unit y _ Inj1 Unit, simplified])
  apply simp
done
have verifier1: << [Inj1 Unit, Unit], t >> V→i << [], Inj1 Unit >> — verifier execution
unfolding i_def t_def Suc_eq_plus1 using fresh
apply -
apply (rule smallsteps.intros)+
  apply (rule s_Let1[rotated])
  apply (rule s_Unauth)
  apply (rule s_AuthV[rotated])
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthV)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let2)
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_Unauth)
  apply (rule s_AuthV[rotated])
  apply simp
  apply simp
  apply simp
  apply (rule s_Let1[rotated])
  apply (rule s_UnauthV)
  apply simp
  apply simp
  apply (rule s_Let2[of Unit y _ Inj1 Unit, simplified])
  apply simp
done
have verifier2: << [Inj2 Unit, Unit], t >> V→i << [], Inj2 Unit >> — verifier execution with proofstream
containing collision
unfolding i_def t_def Suc_eq_plus1 using fresh
apply -
apply (rule smallsteps.intros)+
  apply (rule s_Let1[rotated])
  apply (rule s_Unauth)
  apply (rule s_AuthV[rotated])
  apply simp

```

```

    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_UnauthV)
    apply simp
    apply (simp add: collision)
    apply simp
    apply (rule s_Let2)
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_Unauth)
    apply (rule s_AuthV[rotated])
    apply simp
    apply simp
    apply simp
    apply (rule s_Let1[rotated])
    apply (rule s_UnauthV)
    apply simp
    apply simp
    apply (rule s_Let2[of Unit y _ Inj2 Unit, simplified])
    apply simp
done
have judge: {$$} ⊢ t : Sum One One
  unfolding t_def using fresh
  by (force simp: fresh_Pair fresh_fmap_update)
have ideal_deterministic: e = Inj1 Unit if <<[], t>> I→i <<[], e>> for e
  apply (rule smallsteps_ideal_deterministic[OF that])
  unfolding i_def t_def Suc_eq_plus1 using fresh
  apply -
  apply (rule smallsteps.intros)+
    apply (rule s_Let1[rotated])
    apply (rule s_Unauth)
    apply (rule s_AuthI[rotated])
    apply simp
    apply simp
  apply (rule s_Let1[rotated])
    apply (rule s_UnauthI)
    apply simp
  apply simp
  apply (rule s_Let2)
    apply simp
    apply simp
  apply simp
  apply (rule s_Let1[rotated])
    apply (rule s_Unauth)
    apply (rule s_AuthI[rotated])
    apply simp
  apply simp
  apply (rule s_Let1[rotated])
    apply (rule s_UnauthI)
    apply simp
  apply simp
  apply (rule s_Let2[of Unit y _ Inj1 Unit, simplified])
  apply simp
done

```

```

from agree[OF judge_imp_agree[OF judge] verifier2] collision prover verifier1 show False
proof safe
  fix e' eP'
  assume agree: {$$} ⊢ e', eP', Inj2 Unit : Sum One One
  assume assm: <<[], t>> I→i <<[], e'>>
  then have e' = Inj1 Unit
    by (simp add: ideal_deterministic)
  with agree show False
  by auto
qed (auto dest!: no_collision_with_Unit[OF sym])
qed

```

5.11 Corrected Theorem 1: Security

lemma theorem1_security:

```

assumes {$$} ⊢ e, eP, eV : τ
and <<πA, eV>> V→i <<π', eV'>>
shows (∃ e' eP' π. <<[], e>> I→i <<[], e'>> ∧ <<[], eP>> P→i <<π, eP'>> ∧ πA = π @ π' ∧ {$$}
⊢ e', eP', eV' : τ) ∨
  (∃ eP' j π0 π0' s s'. j ≤ i ∧ <<[], eP>> P→j <<π0 @ [s], eP'>> ∧ πA = π0 @ [s'] @ π0' @ π' ∧ s
≠ s' ∧ hash s = hash s' ∧ closed s ∧ closed s')
using assms(2,1) proof (induct πA eV V i π' eV' rule: smallsteps.induct)
  case (s_Id π e)
  then show ?case by blast
next
  case (s_Tr π1 eV1 i π2 eV2 π3 eV3)
  then obtain e2 π eP2 j π0 π0' s s'
  where <<[], e>> I→i <<[], e2>> ∧ <<[], eP>> P→i <<π, eP2>> ∧ π1 = π @ π2 ∧ {$$} ⊢ e2, eP2, eV2
: τ ∨
    j ≤ i ∧ <<[], eP>> P→j <<π0 @ [s], eP2>> ∧ closed s ∧ closed s' ∧ π1 = π0 @ [s'] @ π0' @ π2
∧ s ≠ s' ∧ hash s = hash s'
  by blast
  then show ?case
proof (elim disjE conjE, goal_cases ok collision)
  case ok
  obtain e3 eP3 π' where
    <<[], e2>> I→ <<[], e3>> <<πP, eP2>> P→ <<πP @ π', eP3>>
    {$$} ⊢ e3, eP3, eV3 : τ ∧ π2 = π' @ π3 ∨
    (∃ s s'. closed s ∧ closed s' ∧ π' = [s] ∧ π2 = [s'] @ π3 ∧ s ≠ s' ∧ hash s = hash s')
  for πP using lemma6[OF ok(4) s_Tr(3), of thesis] by blast
  then show ?case
proof (elim disjE conjE exE, goal_cases ok2 collision)
  case ok2
  with s_Tr(1,3-) ok show ?case
  by auto
next
  case (collision s s')
  then show ?case
proof (intro disjI2 exI conjI)
  from ok collision show <<[], eP>> P→i + 1 <<π @ [s], eP3>>
  by (elim smallsteps.s_Tr) auto
  from ok collision show π1 = π @ [s'] @ [] @ π3
  by simp
qed simp_all
qed
next
  case collision
from s_Tr(3) collision show ?case

```

```

proof (elim smallstepV_consumes_proofstream, intro disjI2 exI conjI)
  fix  $\pi_0''$ 
  assume *:  $\pi_2 = \pi_0'' @ \pi_3$ 
  from collision * show  $\pi_1 = \pi_0 @ [s'] @ (\pi_0' @ \pi_0'') @ \pi_3$ 
  by simp
qed simp_all
qed
qed

```

5.12 Remark 1

lemma remark1_single:

```

assumes { $\$ \$$ }  $\vdash e, eP, eV : \tau$ 
and  $\ll \pi P, eP \gg P \rightarrow \ll \pi P @ \pi, eP' \gg$ 
obtains  $e' eV'$  where { $\$ \$$ }  $\vdash e', eP', eV' : \tau \wedge \ll [], e \gg I \rightarrow \ll [], e' \gg \wedge \ll \pi, eV \gg V \rightarrow \ll [], eV' \gg$ 

```

proof (atomize_elim, insert assms, nominal_induct { $\$ \$$ }:tyenv e eP eV τ avoiding: $\pi P \pi eP'$ rule: agree.strong_induct)

```

case (a_App e1 eP1 eV1  $\tau_1 \tau_2$  e2 eP2 eV2)
from a_App(5) show ?case
proof (cases rule: s_App_inv)
  case (App1 eP1')
  with a_App(2,3) show ?thesis by blast
next
  case (App2 eP2')
  with a_App(1,4) show ?thesis by blast
next
  case (AppLam x eP)
from a_App(1)[unfolded  $\langle eP_1 = \text{Lam } x \text{ eP} \rangle$ ] show ?thesis
proof (cases rule: a_Lam_inv_P[case_names Lam])
  case (Lam v' vV')
  with a_App(3) AppLam show ?thesis
  by (auto 0 4 simp: fresh_Pair del: s_AppLam intro!: s_AppLam lemma4)
qed
next
  case (AppRec x e)
from a_App(1)[unfolded  $\langle eP_1 = \text{Rec } x \text{ e} \rangle$ ] show ?thesis
proof (cases rule: a_Rec_inv_P[case_names _ Rec])
  case (Rec y e'' eP'' eV'')
  show ?thesis
proof (intro exI conjI)
  let ?e = App (Lam y (e''[Rec x (Lam y e'')] / x)) e2
  let ?eV = App (Lam y (eV''[Rec x (Lam y eV'')] / x)) eV2
from a_App(3) Rec AppRec show { $\$ \$$ }  $\vdash ?e, eP', ?eV : \tau_2$ 
  by (auto intro!: agree.a_App[where  $\Gamma = \{\$ \$\}$ ] lemma4
    simp del: subst_term.simps(3) simp: subst_term.simps(3)[symmetric])
from a_App(3) Rec AppRec show  $\ll [], \text{App } e_1 \text{ e}_2 \gg I \rightarrow \ll [], ?e \gg$ 
  by (auto intro!: s_AppRec[where  $v = e_2$ ]
    simp del: subst_term.simps(3) simp: subst_term.simps(3)[symmetric] fresh_Pair)
from a_App(3) Rec AppRec show  $\ll \pi, \text{App } eV_1 \text{ eV}_2 \gg V \rightarrow \ll [], ?eV \gg$ 
  by (auto intro!: s_AppRec[where  $v = eV_2$ ]
    simp del: subst_term.simps(3) simp: subst_term.simps(3)[symmetric] fresh_Pair)
qed
qed simp
qed
next
case (a_Let x e1 eP1 eV1  $\tau_1$  e2 eP2 eV2  $\tau_2$ )
then have atom x  $\# (eP_1, \pi P)$  by auto

```

```

with a_Let(12) show ?case
proof (cases rule: s_Let_inv)
  case Let1
  with a_Let show ?thesis
  by (intro exI[where P =  $\lambda x. \exists y. (Q x y)$  for Q, OF exI, of _ e2[e1 / x] eV2[eV1 / x]])
    (auto intro!: lemma4)
next
case (Let2 eP1')
with a_Let(9) obtain e1' eV1'
  where ih: { $\$\$$ }  $\vdash e1', eP1', eV1' : \tau_1 \ll [], e1 \gg I \rightarrow \ll [], e1' \gg \ll \pi, eV1 \gg V \rightarrow \ll [], eV1' \gg$ 
  by blast
from a_Let Let2 have  $\neg \text{value } e1 \neg \text{value } eP1 \neg \text{value } eV1$  by auto
with Let2 a_Let(2,5,7,10) ih show ?thesis
  by (intro exI[where P =  $\lambda x. \exists y. (Q x y)$  for Q, OF exI, of _ Let e1' x e2 Let eV1' x eV2])
    (fastforce simp: fresh_Pair del: agree.a_Let intro!: agree.a_Let)
qed
next
case (a_Case e eP eV  $\tau_1 \tau_2 e1 eP1 eV1 \tau e2 eP2 eV2$ )
from a_Case(7) show ?case
proof (cases rule: s_Case_inv)
  case (Case eP'')
  with a_Case(2,3,5) show ?thesis by blast
next
case (Inj1 v)
with a_Case(1,3,5) show ?thesis by blast
next
case (Inj2 v)
with a_Case(1,3,5) show ?thesis by blast
qed
next
case (a_Prj1 e eP eV  $\tau_1 \tau_2 \pi P \pi eP'$ )
from a_Prj1(3) show ?case
proof (cases rule: s_Prj1_inv)
  case (Prj1 eP'')
  with a_Prj1(2) show ?thesis by blast
next
case (PrjPair1 v2)
with a_Prj1(1) show ?thesis by fastforce
qed
next
case (a_Prj2 v vP vV  $\tau_1 \tau_2$ )
from a_Prj2(3) show ?case
proof (cases rule: s_Prj2_inv)
  case (Prj2 eP'')
  with a_Prj2(2) show ?thesis by blast
next
case (PrjPair2 v2)
with a_Prj2(1) show ?thesis by fastforce
qed
next
case (a_Roll  $\alpha e eP eV \tau$ )
from a_Roll(7) show ?case
proof (cases rule: s_Roll_inv)
  case (Roll eP'')
  with a_Roll(4,5,6) show ?thesis by blast
qed
next
case (a_Unroll  $\alpha e eP eV \tau$ )

```

```

from  $a\_Unroll(7)$  show ?case
proof (cases rule: s_Unroll_inv)
  case ( $Unroll\ eP''$ )
    with  $a\_Unroll(5,6)$  show ?thesis by fastforce
next
  case  $UnrollRoll$ 
    with  $a\_Unroll(5)$  show ?thesis by blast
qed
next
case ( $a\_Auth\ e\ eP\ eV\ \tau$ )
from  $a\_Auth(3)$  show ?case
proof (cases rule: s_AuthP_inv)
  case ( $Auth\ eP''$ )
    with  $a\_Auth(3)$  show ?thesis
    by (auto dest!: a_Auth(2)[of  $\pi P\ \pi\ eP''$ ])
next
  case  $AuthP$ 
    with  $a\_Auth(1)$  show ?thesis
    by (auto 0 4 simp: lemma2_1 intro: exI[of  $\_ Hash\ (hash\ (eP))$ ] exI[of  $\_ e$ ])
qed
next
case ( $a\_Unauth\ e\ eP\ eV\ \tau$ )
from  $a\_Unauth(1)$  have  $eP\_closed: closed\ eP$ 
  using agree_empty_fresh by simp
from  $a\_Unauth(3)$  show ?case
proof (cases rule: s_UnauthP_inv)
  case ( $Unauth\ e'$ )
    with  $a\_Unauth(2)$  show ?thesis
    by blast
next
  case ( $UnauthP\ h$ )
    with  $a\_Unauth(1,3)$   $eP\_closed$  show ?thesis
    by (force intro: a_AuthT_value_inv[OF a_Unauth(1)] simp: fresh_shallow)
qed
next
case ( $a\_Inj1\ e\ eP\ eV\ \tau_1\ \tau_2$ )
from  $a\_Inj1(3)$  show ?case
proof (cases rule: s_Inj1_inv)
  case ( $Inj1\ eP''$ )
    with  $a\_Inj1(1,2)$  show ?thesis by blast
qed
next
case ( $a\_Inj2\ e\ eP\ eV\ \tau_2\ \tau_1$ )
from  $a\_Inj2(3)$  show ?case
proof (cases rule: s_Inj2_inv)
  case ( $Inj2\ eP''$ )
    with  $a\_Inj2(1,2)$  show ?thesis by blast
qed
next
case ( $a\_Pair\ e_1\ eP_1\ eV_1\ \tau_1\ e_2\ eP_2\ eV_2\ \tau_2$ )
from  $a\_Pair(5)$  show ?case
proof (cases rule: s_Pair_inv)
  case ( $Pair1\ eP_1'$ )
    with  $a\_Pair(1,2,3)$  show ?thesis by blast
next
  case ( $Pair2\ eP_2'$ )
    with  $a\_Pair(1,3,4)$  show ?thesis by blast
qed

```

```

qed (auto dest: value_no_step)

lemma remark1:
  assumes { $\$\$$ }  $\vdash e, eP, eV : \tau$ 
  and  $\ll \pi_P, eP \gg P \rightarrow i \ll \pi_P @ \pi, eP' \gg$ 
  obtains  $e' eV'$ 
  where { $\$\$$ }  $\vdash e', eP', eV' : \tau \ll [], e \gg I \rightarrow i \ll [], e' \gg \ll \pi, eV \gg V \rightarrow i \ll [], eV' \gg$ 
  using assms(2,1)
proof (atomize_elim, nominal_induct  $\pi_P eP P i \pi_P @ \pi eP'$  arbitrary:  $\pi$  rule: smallsteps.strong_induct)
  case (s_Id e  $\pi P$ )
  then show ?case
    using s_Id_inv by blast
next
  case (s_Tr  $\pi_1 eP_1 i \pi_2 eP_2 eP_3$ )
  from s_Tr obtain  $\pi' \pi''$  where ps:  $\pi_2 = \pi_1 @ \pi' \pi = \pi' @ \pi''$ 
  by (force elim: smallstepP_generates_proofstream smallstepsP_generates_proofstream)
  with s_Tr obtain  $e_2 eV_2$  where ih: { $\$\$$ }  $\vdash e_2, eP_2, eV_2 : \tau$ 
   $\ll [], e \gg I \rightarrow i \ll [], e_2 \gg \ll \pi', eV \gg V \rightarrow i \ll [], eV_2 \gg$ 
  by atomize_elim (auto elim: s_Tr(2)[of  $\pi'$ ])
  moreover
  obtain  $e_3 eV_3$  where agree: { $\$\$$ }  $\vdash e_3, eP_3, eV_3 : \tau$  and
   $\ll [], e_2 \gg I \rightarrow \ll [], e_3 \gg \ll \pi', eV_2 \gg V \rightarrow \ll [], eV_3 \gg$ 
  by (rule remark1_single[OF ih(1) iffD2[OF smallstepP_ps_prepend s_Tr(3)[unfolded ps]]]) blast
  ultimately have  $\ll [], e \gg I \rightarrow i + 1 \ll [], e_3 \gg \ll \pi, eV \gg V \rightarrow i + 1 \ll [], eV_3 \gg$ 
  by (auto simp: smallstepsV_ps_append[of _ _ _ [], simplified, symmetric] ps
    intro!: smallsteps.s_Tr[where m=V and  $\pi_1=\pi' @ \pi''$  and  $\pi_2=\pi''$ ])
  with agree show ?case
    by blast
qed

```

References

- [1] M. Brun and D. Traytel. Generic authenticated data structures, formally. In J. Harrison, J. O’Leary, and A. Tolmach, editors, *ITP 2019*, volume 141 of *LIPICs*, pages 10:1–10:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [2] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In S. Jagannathan and P. Sewell, editors, *POPL 2014*, pages 411–424. ACM, 2014.