Formalization of Generic Authenticated Data Structures
Matthias Brun       Dmitriy Traytel
June 11, 2019

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 \( \lambda \) (pronounced \textit{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 \( \lambda \) 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 a paper draft [1].

Contents
1 Preliminaries 2
2 Syntax of \( \lambda \) 5
3 Semantics of \( \lambda \) 6
  3.1 Equivariant Hash Function 6
  3.2 Substitution 7
  3.3 Weak Typing Judgement 9
  3.4 Erasure of Authenticated Types 11
  3.5 Strong Typing Judgement 11
  3.6 Shallow Projection 12
  3.7 Small-step Semantics 13
  3.8 Type Progress 17
  3.9 Weak Type Preservation 17
  3.10 Corrected Lemma 1 from Miller et al. [2]: Weak Type Soundness 18
4 Agreement Relation 18
5 Formalization of Miller et al.’s [2] Main Results 21
  5.1 Lemma 2.1 21
  5.2 Counterexample to Lemma 2.2 21
  5.3 Lemma 2.3 22
  5.4 Lemma 2.4 22
  5.5 Lemma 3 22
  5.6 Lemma 4 22
  5.7 Lemma 5: Single-Step Correctness 22
  5.8 Lemma 6: Single-Step Security 23
  5.9 Theorem 1: Correctness 23
  5.10 Counterexamples to Theorem 1: Security 23
  5.11 Corrected Theorem 1: Security 23
  5.12 Remark 1 24
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]: \{ \} \star t
⟨proof⟩

lemma supp_flip[simp]:
  fixes a b :: at
  shows supp (a ↔ b) = (if a = b then \{\} else \{atom a, atom b\})
⟨proof⟩

lemma Abs_lst_eq_flipI:
  fixes a b :: at and t :: fs
  assumes atom b \star t
  shows [[atom a]]lst. t = [[atom b]]lst. (a ↔ b) · t
⟨proof⟩

lemma atom not fresh.eq:
  assumes \neg atom a \star x
  shows a = x
⟨proof⟩

lemma fresh set_fresh forall:
  shows atom y \star xs = (\forall x \in set xs. atom y \star x)
⟨proof⟩

lemma finite fresh set_fresh all[simp]:
  fixes S :: (\_: fs) set
  shows finite S \Rightarrow atom a \star S \Leftarrow (\forall x \in S. atom a \star x)
⟨proof⟩

lemma case_option_eqvt[eqvt]:
  p · case_option a b opt = case_option (p · a) (p · b) (p · opt)
⟨proof⟩

Nominal setup for finite maps.

abbreviation fmap update (_\$:= \_\$) [1000,0,0] 1000) where fmap update Γ x τ \equiv fmupd x τ Γ
notation fmlookup (infixl \$\$ 999)
notation fmempty (\$\$)

instantiation fmap :: (pt, pt) pt
begin
unbundle fmap.lifting

lift_definition
  permute_fmap :: perm \Rightarrow ('a, 'b) fmap \Rightarrow ('a, 'b) fmap
is
  permute :: perm \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)
⟨proof⟩
instance
⟨proof⟩
end

lemma fmempty_eqvt[eqvt]:
  shows \((p \cdot \{\$$\}) = \{\$$\)\n  ⟨proof⟩
lemma fmap_update_eqvt[eqvt]:
  shows \((p \cdot f(\text{a }\$$:= \text{b})) = (p \cdot f)((p \cdot \text{a}) \$$:= (p \cdot \text{b})\)
  ⟨proof⟩
lemma fmap_apply_eqvt[eqvt]:
  shows \((p \cdot f(\text{a }$$:= \text{b})) = \Gamma(p \cdot f)\)
  ⟨proof⟩
lemma fresh_fmempty[simp]:
  shows \(\text{a }\# \{\$$\}\)
  ⟨proof⟩
lemma fresh_fmap_update:
  shows \(\text{a }\# f; \text{a }\# x; \text{a }\# y\Rightarrow \text{a }\# f(\text{a }\$$:= \text{y})\)
  ⟨proof⟩
lemma supp_fmempty[simp]:
  shows \(\text{supp }\{\$$\} = \{\}\)
  ⟨proof⟩
lemma supp_fmap_update:
  shows \(\text{supp }\varphi(\text{f }\text{x }\$$:= \text{y}) \subseteq \text{supp}(\text{f }, \text{x }, \text{y})\)
  ⟨proof⟩
instance fmap :: (fs, fs) fs
  ⟨proof⟩
lemma fmap_reorder_neq_updates:
  assumes \(\text{a }\neq \text{b}\)
  shows \(\Gamma(\text{a }\$$:= \text{x}) (\text{b }\$$:= \text{y}) = \Gamma(\text{b }\$$:= \text{y}) (\text{a }\$$:= \text{x})\)
  ⟨proof⟩
lemma fmap_upd_upd[simp]: \(\Gamma(\text{x }\$$:= \text{y}) (\text{x }\$$:= \text{z}) = \Gamma(\text{x }\$$:= \text{z})\)
  ⟨proof⟩
lemma fresh_transfer[transfer_rule]:
  \((=) \Rightarrow \text{pcr_fmap} (\text{=}) \Rightarrow (\text{=}) \Rightarrow \text{fresh fresh}\)
  ⟨proof⟩
lemma fmap_fmapd: \(\text{fnmap }\text{f }\text{F}(\text{x }\$$:= \text{y}) = \text{fnmap }\text{f }\text{F})(\text{x }\$$:= \text{f }\text{y})\)
  ⟨proof⟩
lemma fmap_eqvt[eqvt]: \(p \cdot (\text{fnmap }\varphi) = \text{fnmap }\varphi\)
  ⟨proof⟩
lemma fmap_freshness_lemma:
  fixes \(\varphi :: (\text{a }\cdot \text{at}, \text{b }\cdot \text{pt}) \text{fmap}\)
  assumes \(\exists \text{a }\cdot \text{atom }\text{a }\# (\varphi, \text{h }\$$:= \text{a})\)
  shows \(\exists \text{x }\cdot \forall \text{a }\cdot \text{atom }\text{a }\# \text{h }\Rightarrow \text{h }\$$:= \text{a }\text{x}\)
lemma fmap_freshness_lemma_unique:
  fixes h :: ('a::at,'b::pt) fmap
  assumes ∃ a. atom a ‰ (h, h $$ a)
  shows ∃! x. ∀ a. atom a ‰ h −→ h $$ a = x
⟨proof⟩

lemma fmapdrop_fset_fmapd[simp]:
(fmapdrop_fset A f)(x $$:= y) = fmapdrop_fset (A ├ |{x}|) f(x $$:= y)
including fmap.lifting fset.lifting
⟨proof⟩

lemma fresh_fset_fminus:
  assumes atom x ‰ A
  shows A ├ |{x}| = A
⟨proof⟩

lemma fresh_fun_app:
  shows atom x ‰ F =⇒ x ≠ y =⇒ f y = Some a =⇒ atom x ‰ a
⟨proof⟩

lemma fmap_fmap_fresh_Some:
  atom x ‰ F =⇒ x ≠ y =⇒ F $$ y = Some a =⇒ atom x ‰ a
including fmap.lifting
⟨proof⟩

lemma fmapdrop_eqvt: p · fmapdrop x F = fmapdrop (p · x) (p · F)
⟨proof⟩

lemma fnfilter_eqvt: p · fnfilter Q F = fnfilter (p · Q) (p · F)
⟨proof⟩

lemma fmapdrop_fnupd: fmapdrop x F(y $$:= z) = (if x = y then fmapdrop x F else (fmapdrop x F)(y $$:= z))
⟨proof⟩

lemma fmapdrop_eq_iff:
  fmapdrop x B = fmapdrop y B =⇒ x = y ∨ (x ‰ fdom' B ∧ y ‰ fdom' B)
⟨proof⟩

lemma fmapdrop_idle: x ‰ fdom' B =⇒ fmapdrop x B = B
⟨proof⟩

lemma fmapdrop_fnupd_same: fmapdrop x B(x $$:= y) = fmapdrop x B
⟨proof⟩

lemma fresh_fun_upd:
  shows [a ‰ f; a ‰ x; a ‰ y] =⇒ a ‰ f(x := y)
⟨proof⟩

lemma supp_fun_upd:
  shows supp (f(x := y)) ⊆ supp(f, x, y)
⟨proof⟩

lemma map_drop_fun_upd: map_drop x F = F(x := None)
⟨proof⟩

lemma fresh_fmapdrop_in_fdom': [ x ∈ fdom' B; y ‰ B; y ‰ x ] =⇒ y ‰ fmapdrop x B

4
lemma fresh_fmdrop:
  assumes x ♯ B x ♯ y
  shows x ♯ fmdrop y B
⟨proof⟩

lemma fresh_fmdrop_fset:
  fixes x :: atom and A :: (:: at_base) fset
  assumes x ♯ A x ♯ B
  shows x ♯ fmdrop_fset A B
⟨proof⟩

lemma fmdrop_fset_fmdom[simp]: fmdrop_fset (fmdom A) A = {∎}
⟨proof⟩

2 Syntax of λ•

typedecl hash
instantiation hash :: pure
begin
definition permute_hash :: perm ⇒ hash ⇒ hash where
  permute_hash π h = h
instance ⟨proof⟩
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 | 
  Pry1 term | 
  Pry2 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 α::tvar τ::ty binds α in τ | 
  Alpha tvar |
AuthT ty

lemma no_tvars_in_term[simp]: atom (x :: tvar) ≠ (t :: term)
⟨proof⟩

lemma no_vars_in_ty[simp]: atom (x :: var) ≠ (τ :: ty)
⟨proof⟩

inductive value :: term ⇒ bool where
value Unit |
value (Var _ ) |
value (Lam _ ) |
value (Rec _ ) |
value v ➞ value (Inj1 v ) |
value v ➞ value (Inj2 v ) |
[ value v1; value v2 ] ➞ value (Pair v1 v2 ) |
value v ➞ value (Roll v ) |
value (Hash _ ) |
value (Hashed _ )

declare value.intros[simp]
declare value.intros[intro]

equivariance value

lemma value_inv[simp]:
¬ value (Let e1 x e2 )
¬ value (App v v )'
¬ value (Case v v1 v2 )
¬ value (Prj1 v )
¬ value (Prj2 v )
¬ value (Unroll v )
¬ value (Auth v )
¬ value (Unauth v )
⟨proof⟩

inductive_cases value_Inj1_inv[elim]: value (Inj1 v )
inductive_cases value_Inj2_inv[elim]: value (Inj2 v )
inductive_cases value_Pair_inv[elim]: value (Pair e1 e2 )
inductive_cases value_Roll_inv[elim]: value (Roll v )

abbreviation closed :: term ⇒ bool where
closed t ≡ (∀ x::var. atom x ≠ t )

3 Semantics of λ•

Avoid clash with substitution notation.

no_notation inverse Divide (infixl ’ 70 )

Help automated provers with smallsteps.
declare One_nat_def[simp del]

3.1 Equivariant Hash Function

consts hash_real :: term ⇒ hash
nominal_function map_fixed :: var ⇒ var list ⇒ term ⇒ term where

map_fixed fp l Unit = Unit |
map_fixed fp l (Var y) = (if y ∈ set l then (Var y) else (Var fp)) |
atom y ≠ (fp, l) ⇒⇒ map_fixed fp l (Lam y t) = (Lam y ((map_fixed fp (y # l) t))) |
atom y ≠ (fp, l) ⇒⇒ 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) ⇒⇒ 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)))

{proof}

nominal_termination (eqvt)
{proof}

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

{proof}

lemma permute_map_list: p ⋅ l = map (λx. p ⋅ x) l
{proof}

{proof}

lemma map_fixed_eqvt: p ⋅ l = l ⇒⇒ map_fixed v l (p ⋅ t) = map_fixed v l t

{proof}

lemma hash_eqvt [eqvt]: p ⋅ hash t = hash (p ⋅ t)
{proof}

lemma map_fixed_idle: {x. ¬ atom x ≠ t} ⊆ set l ⇒⇒ map_fixed v l t = t
{proof}

lemma map_fixed_inj_closed:
closed t ⇒⇒ closed u ⇒⇒ map_fixed undefined [] t = map_fixed undefined [] u ⇒⇒ t = u
{proof}

3.2 Substitution

nominal_function subst_term :: term ⇒ term ⇒ var ⇒ term ([,.../,] [250, 200, 200] 250) where

Unit(t’ / x) = Unit |
(Var y)[t’ / x] = (if y = x then t’ else Var y) |
atom y ≠ (x, t’) ⇒⇒ (Lam y t)[t’ / x] = Lam y (t[t’ / x]) |
atom y ≠ (x, t’) ⇒⇒ (Rec y t)[t’ / x] = Rec y (t[t’ / x]) |
(Inj1 t)[t’ / x] = Inj1 ([t’ / x]) |
(Inj2 t)[t’ / x] = Inj2 ([t’ / x]) |
(Pair t1 t2)[t’ / x] = Pair (t1[t’ / x]) (t2[t’ / x]) |
(Roll t)[t’ / x] = Roll ([t’ / x]) |
atom y ≠ (x, t’) ⇒⇒ (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]) |
(Hashed h)[t' / x] = Hashed h |
(Hashed h t)[t' / x] = Hached h (t[t' / x])

nominal_termination (eqvt)
{proof}

type_synonym tenv = (var, term) fmap

nominal_function psubst_term :: term ⇒ tenv ⇒ term where
  psubst_term Unit f = Unit |
  psubst_term (Var y) f = (case f $$ y of Some t ⇒ t | None ⇒ Var y) |
  atom y $ f ⇒ psubst_term (Lam y t) f = Lam y (psubst_term t f) |
  atom y $ f ⇒ 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 ⇒ 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)
{proof}

nominal_termination (eqvt)
{proof}

nominal_function subst_type :: ty ⇒ ty ⇒ tvar ⇒ 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)
{proof}

nominal_termination (eqvt)
{proof}

lemma fresh_subst_term: atom x $ t[t' / x] ⇐⇒ (x = x' ∨ atom x $ t) ∧ (atom x' $ t ∨ atom x $ t')
{proof}

lemma term_fresh_subst[simp]: atom x $ t ⇒ atom x $ s ⇒ (atom (x::var)) $ t[s / y]
{proof}

lemma term_subst_idle[simp]: atom y $ t ⇒ t[s / y] = t
{proof}

lemma term_subst_subst: atom y1 ≠ atom y2 ⇒ atom y1 $ s2 ⇒ t[s1 / y1][s2 / y2] = t[s2 / y2][s1[s2 / y2] / y1]
lemma fresh_psubst:
fixes x :: var
assumes atom x # e atom x # vs
shows atom x # psubst_term e vs
(proof)

lemma fresh_subst_type:
atom α # subst_type τ τ' α' # ((α = α' ∨ atom α # τ) ∧ (atom α' # τ ∨ atom α # τ'))
(proof)

lemma type_fresh_subst[simp]: atom x # t => atom x # s => (atom (x::tvar)) # subst_type t s y
(proof)

lemma type_subst_idle[simp]: atom y # t => subst_type t s y = t
(proof)

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
(proof)

3.3 Weak Typing Judgement

type synonym tyenv = (var, ty) fmap

inductive judge_weak :: tyenv => term => ty => bool (.: ) where

jw_Unit: Γ # W Unit : One |
jw_Var: Γ # $x = Some τ |
⇒ Γ # W Var x : τ |
jw_Lam: atom x # Γ # Γ(x $#= τ1) # W e : τ2 |
⇒ Γ # W Lam x e : Fun τ1 τ2 |
jw_App: Γ # W e : Fun τ1 τ2; Γ # W e' : τ1 |
⇒ Γ # W App e e' : τ2 |
jw_Let: atom x # (Γ, e1); Γ # W e1 : τ1; Γ(x $#= τ1) # W e2 : τ2 |
⇒ Γ # W Let e1 e2 : τ2 |
jw_Rec: atom x # Γ; atom y # (Γ, x); Γ(x $#= Fun τ1 τ2) # W Lam y e : Fun τ1 τ2 |
⇒ Γ # W Rec x (Lam y e) : Fun τ1 τ2 |
jw_Inj1: Γ # W e : τ1 |
⇒ Γ # W Inj1 e : Sum τ1 τ2 |
jw_Inj2: Γ # W e : τ2 |
⇒ Γ # W Inj2 e : Sum τ1 τ2 |
jw_Case: Γ # W e : Sum τ1 τ2; Γ # W e1 : Fun τ1 τ2; Γ # W e2 : Fun τ2 τ2 |
⇒ Γ # W Case e e1 e2 : τ |
jw_Pair: Γ # W e1 : τ1; Γ # W e2 : τ2 |
⇒ Γ # W Pair e1 e2 : Prod τ1 τ2 |
jw_Pnj1: Γ # W e : Prod τ1 τ2 |
⇒ Γ # W Pnj1 e : τ1 |
jw_Pnj2: Γ # W e : Prod τ1 τ2 |
⇒ Γ # W Pnj2 e : τ2 |
jw_Roll: atom α # Γ; Γ # W e : subst_type τ (Mu α τ) α |
⇒ Γ # W Roll e : Mu α τ |
jw_Unroll: atom α # Γ; Γ # W e : Mu α τ |
⇒ Γ # W Unroll e : subst_type τ (Mu α τ) α |
jw_Auth: Γ # W e : τ |
⇒ Γ # W Auth e : τ |
jw_Unauth: Γ # W e : τ |
⇒ Γ # W Unauth e : τ
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: \( \alpha \)
  | jw_Unroll: \( \alpha \)
⟨proof⟩

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 atom \( x \notin \Gamma \)
  obtains \( \tau_1 \tau_2 \) where \( \tau = \text{Fun} \tau_1 \tau_2 (\Gamma(x \ll s:= \tau_1)) \vdash_{W} e : \tau_2 \)
⟨proof⟩

lemma swap_permute_swap: atom \( x \not\in \pi \implies \) atom \( y \not\in \pi \implies (x \leftrightarrow y) \cdot \pi \cdot (x \leftrightarrow y) \cdot t = \pi \cdot t \)
⟨proof⟩

lemma jw_Rec_inv[elim]:
  assumes \( \Gamma \vdash_{W} \text{Rec} x t : \tau \)
  and atom \( \alpha \notin (\Gamma, \tau) \)
  obtains \( \alpha e \tau_1 \tau_2 \) where atom \( y \not\in \tau_1 \tau_2 \) (\( \Gamma(x \ll s:= \tau_1) \)) \vdash_{W} \text{Lam} y e : \tau_1 \tau_2 \)
⟨proof⟩

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 atom \( x \notin (e_1, \Gamma) \)
  obtains \( \tau_1 \) where \( \Gamma \vdash_{W} e_1 : \tau_1 \) (\( \Gamma(x \ll s:= \tau_1) \)) \vdash_{W} e_2 : \tau_2 \)
⟨proof⟩

inductive_cases jw_Pj1_inv[elim]: \( \Gamma \vdash_{W} \text{Pj1} e : \tau_1 \)
inductive_cases jw_Pj2_inv[elim]: \( \Gamma \vdash_{W} \text{Pj2} e : \tau_2 \)
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 atom \( b \not\in t \)
  shows subst_type \( t (\text{Mu} a t) a = \text{subst_type} ((a \leftrightarrow b) \cdot t) (\text{Mu} b ((a \leftrightarrow b) \cdot t)) \) \( b \)
⟨proof⟩

lemma jw_Roll_inv[elim]:
  assumes \( \Gamma \vdash_{W} \text{Roll} e : \tau \)
  and atom \( \alpha \not\in (\Gamma, \tau) \)

⟨proof⟩
obtains $\tau'$ where $\tau = \text{Mu }\alpha \tau' \Gamma \vdash W e : \text{subst}_{\text{type}} \tau' (\text{Mu }\alpha \tau') \alpha$

<proof>

lemma \texttt{jw.Unroll.inv[elim]}:
  assumes $\Gamma \vdash W \text{Unroll } e : \tau$
  and $\text{atom }\alpha \not\in (\Gamma, \tau)$
  obtains $\tau'$ where $\tau = \text{subst}_{\text{type}} \tau' (\text{Mu }\alpha \tau') \alpha \Gamma \vdash W e : \text{Mu }\alpha \tau'$
  <proof>

Additional inversion rules based on type rather than term.

inductive_cases \texttt{jw.Prod.inv[elim]}: \{$$\}$ \vdash W e : Prod $\tau_1 \tau_2$

inductive_cases \texttt{jw.Sum.inv[elim]}: \{$$\}$ \vdash W e : Sum $\tau_1 \tau_2$

lemma \texttt{jw.Fun.inv[elim]}:
  assumes \{$$\}$ \vdash W $v : \text{Fun }\tau_1 \tau_2$ \text{ value } v
  obtains $e x$ where $v = \text{Lam }x e \lor v = \text{Rec }x e \text{ atom } x \not\in (c :: \text{term})$
  <proof>

lemma \texttt{jw.Mu.inv[elim]}:
  assumes \{$$\}$ \vdash W $v : \text{Mu }\alpha \tau$
  shows $v'$ where $v = \text{Roll } v'$
  <proof>

3.4 Erasure of Authenticated Types

nominal_function erase :: ty \Rightarrow ty where
  erase \text{One} = \text{One} |
  erase (\text{Fun }\tau_1 \tau_2) = \text{Fun } (\text{erase } \tau_1) (\text{erase } \tau_2) |
  erase (\text{Sum }\tau_1 \tau_2) = \text{Sum } (\text{erase } \tau_1) (\text{erase } \tau_2) |
  erase (\text{Prod }\tau_1 \tau_2) = \text{Prod } (\text{erase } \tau_1) (\text{erase } \tau_2) |
  erase (\text{Mu }\alpha \tau) = \text{Mu }\alpha (\text{erase } \tau) |
  erase (\text{Alpha }\alpha) = \text{Alpha }\alpha |
  erase (\text{AuthT }\tau) = \text{erase } \tau

<proof>

nominal_termination (eqvt)
  <proof>

lemma \texttt{fresh erase fresh}:
  assumes $\text{atom }x \not\in \tau$
  shows $\text{atom }x \not\in \text{erase } \tau$
  <proof>

lemma \texttt{fresh fmmap erase fresh}:
  assumes $\text{atom }x \not\in \Gamma$
  shows $\text{atom }x \not\in \text{fmmap erase } \Gamma$
  <proof>

lemma \texttt{erase subst_type shift[simp]}:
  erase ($\text{subst}_{\text{type}} \tau \tau' \alpha$) = subst$_{\text{type}}$ (erase $\tau$) (erase $\tau'$) $\alpha$
  <proof>

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 (\vdash - : [-150,0,150]) where
  jUnit: $\Gamma \vdash \text{Unit} : \text{One} |
j. Var:  \[ \Gamma \triangleright x = \text{Some } \tau \]  \[ \Rightarrow \Gamma \vdash \text{Var } x : \tau \]

j. Lam:  \[ \Gamma \triangleright \text{atom } x \not\in \Gamma; \Gamma(x \triangleright \tau_1) \vdash e : \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Lam } x e : \text{Fun } \tau_1 \tau_2 \]

j. App:  \[ \Gamma \vdash e : \text{Fun } \tau_1 \tau_2; \Gamma \vdash e' : \tau_1 \]  \[ \Rightarrow \Gamma \vdash \text{App } e e' : \tau_2 \]

j. Let:  \[ \Gamma \triangleright \text{atom } x \not\in \Gamma); \Gamma \vdash e_1 : \tau_1; \Gamma(x \triangleright \tau_1) \vdash e_2 : \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Let } e_1 x e_2 : \tau_2 \]

j. Rec:  \[ \Gamma \ vdash e : \text{atom } y \not\in \Gamma, x; \Gamma(x \triangleright \tau_1 \tau_2) \vdash \text{Lam } y e' : \text{Fun } \tau_1 \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Rec } x (\text{Lam } y e') : \text{Fun } \tau_1 \tau_2 \]

j. Inj1:  \[ \Gamma \vdash e : \tau_1 \]  \[ \Rightarrow \Gamma \vdash \text{Inj1 } e : \text{Sum } \tau_1 \tau_2 \]

j. Inj2:  \[ \Gamma \vdash e : \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Inj2 } e : \text{Sum } \tau_1 \tau_2 \]

j. Case:  \[ \Gamma \vdash e : \text{Sum } \tau_1 \tau_2; \Gamma \vdash e_1 : \text{Fun } \tau_1 \tau_1; \Gamma \vdash e_2 : \text{Fun } \tau_2 \tau \]  \[ \Rightarrow \Gamma \vdash \text{Case } e e_1 e_2 : \tau \]

j. Pair:  \[ \Gamma \vdash e_1 : \tau_1 1; \Gamma \vdash e_2 : \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Pair } e_1 e_2 : \text{Prod } \tau_1 \tau_2 \]

j. Prj1:  \[ \Gamma \vdash e : \text{Prod } \tau_1 \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Prj1 } e : \tau_1 \]

j. Prj2:  \[ \Gamma \vdash e : \text{Prod } \tau_1 \tau_2 \]  \[ \Rightarrow \Gamma \vdash \text{Prj2 } e : \tau_2 \]

j. Roll:  \[ \Gamma \vdash e : \text{subst_type } \tau (\text{Mu } \alpha \tau) \alpha \]  \[ \Rightarrow \Gamma \vdash \text{Roll } e : \text{Mu } \alpha \tau \]

j. Unroll:  \[ \Gamma \vdash e : \text{Mu } \alpha \tau \]  \[ \Rightarrow \Gamma \vdash \text{Unroll } e : \text{subst_type } \tau (\text{Mu } \alpha \tau) \alpha \]

j. Auth:  \[ \Gamma \vdash e : \tau \]  \[ \Rightarrow \Gamma \vdash \text{Auth } e : \text{Auth } \tau \]

j. Unauth:  \[ \Gamma \vdash e : \text{Auth } \tau \]  \[ \Rightarrow \Gamma \vdash \text{Unauth } e : \tau \]

\textbf{declare} \ \textit{judge.intros[intro]}

\textbf{equivariance judge}

\textbf{nominal_inductive judge}

\textbf{avoids} j.Lam: x  
|  j.Rec: x and y  
|  j.Let: x  
|  j.Roll: \alpha  
|  j.Unroll: \alpha

\textit{proof}

\textbf{lemma} \ \textit{judge_imp_judge_weak:}

\textbf{assumes} \ \Gamma \vdash e : \tau

\textbf{shows} \ \textit{erase_env} \ \Gamma \vdash_W e : \textit{erase } \tau

\textit{proof}

\textbf{3.6 Shallow Projection}

\textbf{nominal_function} \ \textit{shallow} :: \textit{term} \Rightarrow \textit{term} (\langle \_ \rangle) \ \textit{where}

\begin{align*}
(\langle \text{Unit} \rangle) &= \text{Unit} \ |
(\langle \text{Var } e \rangle) &= \text{Var } v \ |
(\langle \text{Lam } x e \rangle) &= \text{Lam } x (\langle e \rangle) \ |
(\langle \text{Rec } x e \rangle) &= \text{Rec } x (\langle e \rangle) \ |
(\langle \text{Inj1 } e \rangle) &= \text{Inj1 } (\langle e \rangle) \ |
(\langle \text{Inj2 } e \rangle) &= \text{Inj2 } (\langle e \rangle) \ |
(\langle \text{Pair } e_1 e_2 \rangle) &= \text{Pair } (\langle e_1 \rangle) (\langle e_2 \rangle) \ |
(\langle \text{Roll } e \rangle) &= \text{Roll } (\langle e \rangle) \ |
\end{align*}

12
3.7 Small-step Semantics

datatype mode = I | P | V — Ideal, Prover and Verifier modes

instantiation mode :: pure
begin
definition permute_mode :: perm ⇒ mode ⇒ mode where
  permute_mode π h = h
instance {proof}
end

type_synonym proofstream = term list

inductive smallstep :: proofstream ⇒ term ⇒ mode ⇒ proofstream ⇒ term ⇒ bool (≪... ≫... → ≫...)
where
  s_App1: [≪π, e1≫ m→ ≪π', e1'≫]
    ⇒ ≪π, App e1 e2≫ m→ ≪π', App e1' e2≫
  s_App2: [value v1; ≪π, e2≫ m→ ≪π', e2'≫]
    ⇒ ≪π, App e1 v1≫ m→ ≪π', App e1' v1≫
  s_AppLam: [value v; atom x ≡ (v,π)]
    ⇒ ≪π, App (Lam x e) v≫ → ≪π, [v / x]≫
  s_AppRec: [value v; atom x ≡ (v,π)]
    ⇒ ≪π, App (Rec x e) v≫ → ≪π, App (e(Rec x e / x)) v≫
  s_Let1: [atom x ≡ (e1,e1',π,π'); ≪π, e1≫ m→ ≪π', e1'≫]
    ⇒ ≪π, Let e1 x e2≫ m→ ≪π', Let e1' x e2≫
  s_Let2: [value v; atom x ≡ (v,π)]
    ⇒ ≪π, Let v x e≫ → ≪π, [v / x]≫
  s_Inj1: [≪π, e≫ m→ ≪π', e'≫]
    ⇒ ≪π, Inj1 e≫ m→ ≪π', Inj1 e'≫
  s_Inj2: [≪π, e≫ m→ ≪π', e'≫]
    ⇒ ≪π, Inj2 e≫ m→ ≪π', Inj2 e'≫
  s_Case: [≪π, e≫ m→ ≪π', e'≫]
    ⇒ ≪π, Case e1 e2≫ m→ ≪π', Case e'1 e2≫
— Case rules are different from paper to account for recursive functions.
  s_CaseInj1: [value v]
    ⇒ ≪π, Case (Inj1 v) e1 e2≫ → ≪π, App e1 v≫
  s_CaseInj2: [value v]
    ⇒ ≪π, Case (Inj2 v) e1 e2≫ → ≪π, App e2 v≫
  s_Pair1: [≪π, e1≫ m→ ≪π', e1'≫]
    ⇒ ≪π, Pair e1 e2≫ m→ ≪π', Pair e1' e2≫
\[\begin{align*}
&\text{s_Pair2:} & \text{value } v_1; \langle \pi, e_2 \rangle & \rightarrow \langle \pi', e_2' \rangle \\
& & \rightarrow \langle \pi, \text{Pair } v_1 e_2 \rangle & \rightarrow \langle \pi', \text{Pair } v_1 e_2' \rangle \\
&\text{s_Pij1:} & \langle \langle \pi, e \rangle & \rightarrow \langle \pi', e' \rangle \rangle \\
& & \rightarrow \langle \pi, \text{Prij1 } e \rangle & \rightarrow \langle \pi', \text{Prij1 } e' \rangle \\
&\text{s_Prij2:} & \langle \langle \pi, e \rangle & \rightarrow \langle \pi', e' \rangle \rangle \\
& & \rightarrow \langle \pi, \text{Prij2 } e \rangle & \rightarrow \langle \pi', \text{Prij2 } e' \rangle \\
&\text{s_PrijPair1:} & \text{value } v_1; \text{value } v_2 \\
& & \rightarrow \langle \langle \pi, \text{Prij1 } (\text{Pair } v_1 v_2) \rangle \rangle & \rightarrow \langle \pi, v_1 \rangle \\
&\text{s_PrijPair2:} & \text{value } v_1; \text{value } v_2 \\
& & \rightarrow \langle \langle \pi, \text{Prij2 } (\text{Pair } v_1 v_2) \rangle \rangle & \rightarrow \langle \pi, v_2 \rangle \\
&\text{s_Unroll:} & \langle \langle \pi, e \rangle & \rightarrow \langle \pi', e' \rangle \rangle \\
& & \rightarrow \langle \pi, \Unroll e \rangle & \rightarrow \langle \pi', \Unroll e' \rangle \\
&\text{s_Roll:} & \langle \langle \pi, e \rangle & \rightarrow \langle \pi', e' \rangle \rangle \\
& & \rightarrow \langle \pi, \Roll e \rangle & \rightarrow \langle \pi', \Roll e' \rangle \\
&\text{s_UnrollRoll1:} & \text{value } v \\
& & \rightarrow \langle \langle \pi, \Unroll \ (\Roll v) \rangle \rangle & \rightarrow \langle \pi, v \rangle \\
\end{align*}\]

---

**Mode-specific rules**

\[\begin{align*}
&\text{s_Auth:} & \langle \langle \pi, e \rangle & \rightarrow \langle \pi', e' \rangle \rangle \\
& & \rightarrow \langle \pi, \text{Auth } e \rangle & \rightarrow \langle \pi', \text{Auth } e' \rangle \\
&\text{s_Unauth:} & \langle \langle \pi, e \rangle & \rightarrow \langle \pi', e' \rangle \rangle \\
& & \rightarrow \langle \pi, \text{Unauth } e \rangle & \rightarrow \langle \pi', \text{Unauth } e' \rangle \\
&\text{s_AuthI:} & \text{value } v \\
& & \rightarrow \langle \langle \pi, \text{Auth } v \rangle \rangle & \rightarrow \langle \pi, v \rangle \\
&\text{s_UnauthI:} & \text{value } v \\
& & \rightarrow \langle \langle \pi, \text{Unauth } v \rangle \rangle & \rightarrow \langle \pi, v \rangle \\
&\text{s_AuthP:} & \text{closed } \langle \{v\}; \text{value } v \rangle \\
& & \rightarrow \langle \langle \pi, \text{Auth } v \rangle \rangle & \rightarrow \langle \pi, \text{Hashed } (\text{Hash } \{v\}) v \rangle \\
&\text{s_UnauthP:} & \text{value } v \\
& & \rightarrow \langle \langle \pi, \text{Unauth } (\text{Hash } v) \rangle \rangle & \rightarrow \langle \pi, v \rangle \\
&\text{s_AuthV:} & \text{closed } \{v\}; \text{Hash } s_0 \rightarrow \text{Hash } v \\
& & \rightarrow \langle \langle \pi, \text{Auth } v \rangle \rangle & \rightarrow \langle \pi, \text{Hash } v \rangle \\
&\text{s_UnauthV:} & \text{closed } s_0; \text{Hash } s_0 \rightarrow \text{Hash } v \\
& & \rightarrow \langle \langle \pi, \text{Unauth } (\text{Hash } h) \rangle \rangle & \rightarrow \langle \pi, s_0 \rangle \\
\end{align*}\]

**Declare** \(\text{smallstep\_intros}[\text{simp}]\)

**Declare** \(\text{smallstep\_intros}[\text{intro}]\)

**Equiavariance** \(\text{smallstep}\)

**Nominal inductive** \(\text{smallstep}\)

**Avoids** \(\text{s_Applam}: x\)

\[\begin{align*}
&| \text{s_AppRec}: x \\
&| \text{s_Let1}: x \\
&| \text{s_Let2}: x \\
\end{align*}\]

**Proof**

**Inductive** \(\text{smallsteps} : \text{proofstream} \Rightarrow \text{term} \Rightarrow \text{mode} \Rightarrow \text{nat} \Rightarrow \text{proofstream} \Rightarrow \text{term} \Rightarrow \text{bool} (\langle \pi, \langle \_ \_ \rangle \rightarrow \langle \_ \_ \rangle, \langle \_ \_ \rangle) \text{ where}\)

\[\begin{align*}
&\text{s_Id:} & \langle \langle \pi, e \rangle \rangle & \rightarrow 0 & \rightarrow 0 & \langle \pi, e \rangle \\
&\text{s_Tr:} & \langle \langle \pi_1, e_1 \rangle \rangle & \rightarrow \langle \pi_2, e_2 \rangle \rightarrow m & \rightarrow \langle \pi_3, e_3 \rangle \\
& & \rightarrow \langle \pi_1, e_1 \rangle & \rightarrow m \rightarrow (i+1) & \rightarrow \langle \pi_3, e_3 \rangle \\
\end{align*}\]

**Declare** \(\text{smallsteps\_intros}[\text{simp}]\)

**Declare** \(\text{smallsteps\_intros}[\text{intro}]\)

**Equiavariance** \(\text{smallsteps}\)

**Nominal inductive** \(\text{smallsteps} \ (\text{proof})\)

**Lemma** \(\text{steps\_1\_step}[\text{simp}]: \langle \pi, e \rangle \rightarrow m \rightarrow 1 & \rightarrow \langle \pi', e' \rangle \rightarrow m \rightarrow \langle \pi', e' \rangle \rightarrow m \rightarrow (\text{is } \text{L} \Rightarrow \text{R})\)
\langle \text{proof} \rangle

Inversion rules for smallstep(s) predicates.

**lemma value_no_step|intro:**

assumes \( \ll \pi_1, v \gg m \rightarrow \ll \pi_2, t \gg v 
\)

shows \( \text{False} \)

\langle \text{proof} \rangle

**lemma subst_term_perm:**

assumes \( \text{atom } x \Downarrow (x, e) \)

shows \( e[v / x] = ((x \leftrightarrow x') \cdot e)[v / x'] \)

\langle \text{proof} \rangle

**inductive_cases s_Unit_inv|elim:** \( \ll \pi_1, \ Unit \gg m \rightarrow \ll \pi_2, v \gg \)

**inductive_cases s_App_inv|consumes 1, case_names App1 App2 AppLam AppRec, elim:** \( \ll \pi, \ App v_1 v_2 \gg m \rightarrow \ll \pi', e \gg \)

**lemma s_Let_inv':**

assumes \( \ll \pi, \ Let_1 e_1 x e_2 \gg m \rightarrow \ll \pi', e' \gg \)

and \( \text{atom } x \Downarrow (e_1, \pi) \)

obtains \( e_1' \where (e' = e_2[e_1 / x] \land \text{value } e_1 \land \pi = \pi') \lor (\ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg \land e' = \text{Let } e_1' x e_2 \land \neg \text{value } e_1) \)

\langle \text{proof} \rangle

**lemma s_Let_inv[consumes 2, case_names Let1 Let2, elim]:**

assumes \( \ll \pi, \ Let_1 e_1 x e_2 \gg m \rightarrow \ll \pi', e' \gg \)

and \( \text{atom } x \Downarrow (e_1, \pi) \)

and \( e' = e_2[e_1 / x] \land \text{value } e_1 \land \pi = \pi' \Rightarrow Q \)

and \( \land e_1'. \ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg \land e' = \text{Let } e_1' x e_2 \land \neg \text{value } e_1 \Rightarrow Q \)

shows \( Q \)

\langle \text{proof} \rangle

**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_Pry1_inv|consumes 1, case_names Prj1 PrjPair1, elim:**

\( \ll \pi, \ Prj1 e \gg m \rightarrow \ll \pi', v \gg \)

**inductive_cases s_Pry2_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:**

\( \ll \pi, \ Inj_1 e \gg m \rightarrow \ll \pi', e' \gg \)

**inductive_cases s_Inj2_inv|consumes 1, case_names Inj2, elim:**

\( \ll \pi, \ Inj_2 e \gg m \rightarrow \ll \pi', e' \gg \)

**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_AuthL_inv|consumes 1, case_names Auth AuthI, elim:**

\( \ll \pi, \ Auth e \gg I \rightarrow \ll \pi', e' \gg \)

**inductive_cases s_AuthL_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_AuthP_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:**
\(\langle \pi, \text{Auth} \rangle \rightarrow V \rightarrow \langle \pi', e' \rangle\)

**inductive cases** \(s_{\text{UnauthV\_inv}}[\text{consumes}\ 1,\ \text{case}\_\text{names}\ \text{Unauth}\ \text{UnauthV},\ \text{elim}]\):
- \(\langle \pi, \text{Unauth} \rangle \rightarrow e \rightarrow \langle \pi', e' \rangle\)

**inductive cases** \(s_{\text{Id\_inv}}[\text{elim}]\):
- \(\langle \pi_1, e_1 \rangle \rightarrow m \rightarrow \theta \rightarrow \langle \pi_2, e_2 \rangle\)

**inductive cases** \(s_{\text{Tr\_inv}}[\text{elim}]\):
- \(\langle \pi_1, e_1 \rangle \rightarrow m \rightarrow i \rightarrow \langle \pi_3, e_3 \rangle\)

Freshness with smallstep.

**lemma** \(\text{fresh\_smallstep}\_I\):
- \(\text{fixes } x :: \text{var}\)
- \(\text{assumes } \langle \pi, e \rangle \rightarrow I \rightarrow \langle \pi', e' \rangle \rightarrow \text{atom } x \not\in e\)
- \(\text{shows } \text{atom } x \not\in e'\)

**proof**

**lemma** \(\text{fresh\_smallstep}\_P\):
- \(\text{fixes } x :: \text{var}\)
- \(\text{assumes } \langle \pi, e \rangle \rightarrow P \rightarrow \langle \pi', e' \rangle \rightarrow \text{atom } x \not\in e\)
- \(\text{shows } \text{atom } x \not\in e'\)

**proof**

**lemma** \(\text{fresh\_smallsteps}\_I\):
- \(\text{fixes } x :: \text{var}\)
- \(\text{assumes } \langle \pi, e \rangle \rightarrow I \rightarrow i \rightarrow \langle \pi', e' \rangle \rightarrow \text{atom } x \not\in e\)
- \(\text{shows } \text{atom } x \not\in e'\)

**proof**

**lemma** \(\text{fresh\_ps\_smallstep}\_P\):
- \(\text{fixes } x :: \text{var}\)
- \(\text{assumes } \langle \pi, e \rangle \rightarrow P \rightarrow \langle \pi', e' \rangle \rightarrow \text{atom } x \not\in e \text{ atom } x \not\in \pi\)
- \(\text{shows } \text{atom } x \not\in \pi'\)

**proof**

Proofstream lemmas.

**lemma** \(\text{smallstep}\_I\_ps\_eq\):
- \(\text{assumes } \langle \pi, e \rangle \rightarrow I \rightarrow \langle \pi', e' \rangle \rightarrow \pi = \pi'\)
- \(\text{proof}\)

**lemma** \(\text{smallstep}\_I\_ps\_emptyD\):
- \(\langle [] \rangle \rightarrow \langle [], e' \rangle \Rightarrow \langle [], e \rangle \rightarrow \langle [], e' \rangle\)
- \(\langle [], e \rangle \rightarrow \langle \pi, e' \rangle \Rightarrow \langle [], e \rangle \rightarrow \langle [], e' \rangle\)
- \(\text{proof}\)

**lemma** \(\text{smallsteps}\_I\_ps\_eq\):
- \(\text{assumes } \langle \pi, e \rangle \rightarrow I \rightarrow i \rightarrow \langle \pi', e' \rangle \rightarrow \pi = \pi'\)
- \(\text{proof}\)

**lemma** \(\text{smallsteps}\_I\_ps\_emptyD\):
- \(\langle [], e \rangle \rightarrow i \rightarrow \langle [], e' \rangle \Rightarrow \langle [], e \rangle \rightarrow i \rightarrow \langle [], e' \rangle\)
- \(\text{proof}\)

**lemma** \(\text{smallstep}\_V\_consumes\_proofstream\):
- \(\text{assumes } \langle \pi_1, eV \rangle \rightarrow V \rightarrow \langle \pi_2, eV' \rangle\)
- \(\text{obtains } \pi \text{ where } \pi_1 \oplus \pi_2\)
- \(\text{proof}\)
lemma smallstepsV\_consumes\_proofstream:
assumes $\langle \pi_1, eV \rangle \xrightarrow{i} \langle \pi_2, eV' \rangle$
obtains $\pi$ where $\pi_1 = \pi \odot \pi_2$
(proof)

lemma smallstepP\_generates\_proofstream:
assumes $\langle \pi_1, eP \rangle \xrightarrow{P} \langle \pi_2, eP' \rangle$
obtains $\pi$ where $\pi_2 = \pi_1 \odot \pi$
(proof)

lemma smallstepsP\_generates\_proofstream:
assumes $\langle \pi_1, eP \rangle \xrightarrow{P} \langle \pi_2, eP' \rangle$
obtains $\pi$ where $\pi_2 = \pi_1 \odot \pi$
(proof)

lemma smallstepV\_ps\_append:
$\langle \pi, eV \rangle \xrightarrow{\pi_1 \times \pi_2 \odot \pi} \langle \pi', eV' \rangle$ $\iff \langle \pi \odot X, eV \rangle \xrightarrow{i} \langle \pi' \odot X, eV' \rangle$ (is ?L $\leftrightarrow$ ?R)
(proof)

lemma smallstepV\_ps\_to\_suffix:
assumes $\langle \pi, e \rangle \xrightarrow{\pi_1 \times \pi_2 \odot \pi} \langle \pi', e' \rangle$
obtains $\pi''$ where $\pi = \pi'' \odot X$
(proof)

lemma smallstepsV\_ps\_append:
$\langle \pi, eV \rangle \xrightarrow{\pi_1 \times \pi_2 \odot \pi} \langle \pi', eV' \rangle$ $\iff \langle \pi \odot X, eV \rangle \xrightarrow{i} \langle \pi' \odot X, eV' \rangle$ (is ?L $\leftrightarrow$ ?R)
(proof)

lemma smallstepP\_ps\_prepend:
$\langle \pi, eP \rangle \xrightarrow{P} \langle \pi_1 \times \pi_2 \odot \pi} \langle \pi', eP' \rangle$ $\iff \langle X \odot \pi, eP \rangle \xrightarrow{P} \langle X \odot \pi' \odot \pi' \odot \pi \odot \pi, eP' \rangle$ (is ?L $\leftrightarrow$ ?R)
(proof)

lemma smallstepsP\_ps\_prepend:
$\langle \pi, eP \rangle \xrightarrow{P} \langle \pi_1 \times \pi_2 \odot \pi} \langle \pi', eP' \rangle$ $\iff \langle X \odot \pi, eP \rangle \xrightarrow{P} \langle X \odot \pi' \odot \pi' \odot \pi \odot \pi, eP' \rangle$ (is ?L $\leftrightarrow$ ?R)
(proof)

3.8 Type Progress

lemma type\_progress:
assumes $\{$$\} \vdashW e : \tau$
shows value $e \lor (\exists e'. \langle [], e \rightarrow I \rangle \iff \langle [], e' \rangle)$
(proof)

3.9 Weak Type Preservation

lemma fresh\_tyenv\_None:
fixes $\Gamma :: tyenv$
shows atom $x \notin \Gamma \iff \Gamma \# x = \None$ (is ?L $\leftrightarrow$ ?R)
(proof)

lemma judge\_weak\_fresh\_env\_fresh\_term[dest]:
fixes $a :: \var$
assumes $\Gamma \vdashW e : \tau$ atom $a \notin \Gamma$
shows atom $a \notin e$
(proof)

lemma judge\_weak\_weakening\_1:
assumes $\Gamma \vdashW e : \tau$ atom $y \notin e$
shows $\Gamma(y \# : \tau') \vdashW e : \tau$

\section*{4 Agreement Relation}

\textbf{inductive agree :: tyenv ⇒ term ⇒ term ⇒ term ⇒ ty ⇒ bool} (.⇒_.⇒_.⇒_.⇒_.⇒_.⇒.⇒.⇒.[150,0,0,0,150]149)

\textbf{where}

\begin{itemize}
  \item \textbf{a.Unit}: \(\Gamma \vdash \text{Unit}, \text{Unit}, \text{Unit} : \text{One} \mid \)
  \item \textbf{a.Var}: \(\Gamma \vdash \text{Var} \; x, \text{Var} \; x, \text{Var} \; x : \tau \mid \)
  \item \textbf{a.Lam}: \([\text{atom} \; x \; \sharp \; \Gamma; \Gamma(x \; \$$\Rightarrow$$ \; \tau) \vdash e, eP, eV : \tau_2] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{Lam} \; x \; e, \text{Lam} \; x \; eP, \text{Lam} \; x \; eV : \text{Fun} \; \tau_1 \; \tau_2 \mid \)
  \item \textbf{a.App}: \([\Gamma \vdash e_1, eP_1, eV_1 : \text{Fun} \; \tau_1 \; \tau_2; \Gamma \vdash e_2, eP_2, eV_2 : \tau_1] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{App} \; e_1 \; e_2, \text{App} \; eP_1 \; eP_2, \text{App} \; eV_1 \; eV_2 : \tau_2 \mid \)
  \item \textbf{a.Let}: \([\text{atom} \; x \; \sharp \; (\Gamma, e_1, eP_1, eV_1); \Gamma \vdash e_1, eP_1, eV_1 : \tau_1; \Gamma(x \; \$$\Rightarrow$$ \; \tau) \vdash e_2, eP_2, eV_2 : \tau_2] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{Let} \; e_1 \; x \; e_2, \text{Let} \; eP_1 \; x \; eP_2, \text{Let} \; eV_1 \; x \; eV_2 : \tau_2 \mid \)
  \item \textbf{a.Rec}: \([\text{atom} \; x \; \sharp \; \Gamma; \text{atom} \; y \; \sharp \; (\Gamma, x); \Gamma(x \; \$$\Rightarrow$$ \; \tau_1 \; \tau_2) \vdash \text{Lam} \; y \; e, \text{Lam} \; y \; eP, \text{Lam} \; y \; eV : \text{Fun} \; \tau_1 \; \tau_2] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{Rec} \; x \; (\text{Lam} \; y \; e), \text{Rec} \; x \; (\text{Lam} \; y \; eP), \text{Rec} \; x \; (\text{Lam} \; y \; eV) : \text{Fun} \; \tau_1 \; \tau_2 \mid \)
  \item \textbf{a.Inj1}: \([\Gamma \vdash e, eP, eV : \tau_1] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{Inj1} \; e, \text{Inj1} \; eP, \text{Inj1} \; eV : \text{Sum} \; \tau_1 \; \tau_2 \mid \)
  \item \textbf{a.Inj2}: \([\Gamma \vdash e, eP, eV : \tau_2] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{Inj2} \; e, \text{Inj2} \; eP, \text{Inj2} \; eV : \text{Sum} \; \tau_1 \; \tau_2 \mid \)
  \item \textbf{a.Case}: \([\Gamma \vdash e, eP, eV : \text{Sum} \; \tau_1 \; \tau_2; \Gamma \vdash e_1, eP_1, eV_1 : \text{Fun} \; \tau_1 \; \tau; \Gamma \vdash e_2, eP_2, eV_2 : \text{Fun} \; \tau_2 \; \tau] \)
  \quad \Rightarrow \quad \Gamma \vdash \text{Case} \; e \; e_1 \; e_2, \text{Case} \; eP \; eP_1 \; eP_2, \text{Case} \; eV \; eV_1 \; eV_2 : \tau_2 \mid \)
  \item \textbf{a.Pair}: \([\Gamma \vdash e_1, eP_1, eV_1 : \tau_1; \Gamma \vdash e_2, eP_2, eV_2 : \tau_2] \)
\end{itemize}
\[
\text{lemma } \text{Abs_Lst_eq_3tuple:}
\text{fixes } x \ x' :: \text{var}
\text{fixes } e \ eP \ eV \ e' \ eP' \ eV' :: \text{term}
\text{assumes } \Gamma \vdash e, eP, eV : \tau \text{ atom } a :: \emptyset \Gamma
\text{shows } \text{atom } a :: (e, eP, eV)
\langle\text{proof}\rangle
\]

\[
\text{lemma } \text{agree_fresh_env_fresh_term:}
\text{fixes } a :: \text{var}
\text{assumes } \Gamma \vdash e, eP, eV : \tau \text{ atom } a :: \emptyset \Gamma
\text{shows } \text{atom } a :: (e, eP, eV)
\langle\text{proof}\rangle
\]

\[
\text{lemma } \text{agree_empty_fresh[dest]:}
\text{fixes } a :: \text{var}
\text{assumes } 
\{\text{empty}\} \vdash e, eP, eV : \tau
\text{shows } \{\text{atom } a :: \emptyset\} \subseteq \{e, eP, eV\}
\langle\text{proof}\rangle
\]

Inversion rules for agreement.

\[
\text{declare } \text{[[simpdel: alpha_lst]]}
\]

\[
\text{lemma } \text{a_Lam_inv_I[elim]:}
\text{assumes } \Gamma \vdash (\text{Lam } x e'), eP, eV : (\text{Fun } \tau_1 \tau_2)
\text{and } \text{atom } x :: \emptyset \Gamma
\text{obtains } eP' \ eV' \text{ where } eP = \text{Lam } x eP' eV = \text{Lam } x eV' \Gamma(x :: \emptyset \vdash e', eP', eV' : \tau_2)
\langle\text{proof}\rangle
\]

\[
\text{lemma } \text{a_Lam_inv_P[elim]:}
\]
assumes \( \{ \alpha \} \vdash v \), \( (\text{Lam } vP'), vV : (\text{Fun } \tau_1 \tau_2) \)
obtains \( v' \) \( vV' \) where \( v = \text{Lam } x v' vV = \text{Lam } x vV' \) \( \{ \alpha \} \vdash v', vP', vV' : \tau_2 \)

\( \langle \text{proof} \rangle \)

**Lemma a_Lam_inv_V[elim]**:
assumes \( \{ \alpha \} \vdash v, vP, (\text{Lam } x vV') : (\text{Fun } \tau_1 \tau_2) \)
obtains \( v' \) \( vP' \) where \( v = \text{Lam } x v' vP = \text{Lam } x vP' \) \( \{ \alpha \} \vdash v', vP', vV' : \tau_2 \)

\( \langle \text{proof} \rangle \)

**Lemma a_Rec_inv_I[elim]**:
assumes \( \Gamma \vdash e, eP, eV : \text{Fun } \tau_1 \tau_2 \)
and \( \text{atom } x \notin \Gamma \)
obtains \( y' \) \( eV' \) where \( e = \text{Rec } x (\text{Lam } y eP') \) \( eV = \text{Rec } x (\text{Lam } y eV') \) \( \text{atom } y \notin (\Gamma, x) \)
\( \Gamma(x \{ \alpha \} \vdash \text{Lam } y e', \text{Lam } y eP', \text{Lam } y eV' : \text{Fun } \tau_1 \tau_2) \)

\( \langle \text{proof} \rangle \)

**Lemma a_Rec_inv_P[elim]**:
assumes \( \Gamma \vdash e, eP, eV : \text{Fun } \tau_1 \tau_2 \)
and \( \text{atom } x \notin \Gamma \)
obtains \( y' \) \( eV' \) where \( e = \text{Rec } x (\text{Lam } y e') \) \( eP = \text{Rec } x (\text{Lam } y eP') \) \( eV = \text{Rec } x (\text{Lam } y eV') \) \( \text{atom } y \notin (\Gamma, x) \)
\( \Gamma(x \{ \alpha \} \vdash \text{Lam } y e', \text{Lam } y eP', \text{Lam } y eV' : \text{Fun } \tau_1 \tau_2) \)

\( \langle \text{proof} \rangle \)

**Lemma a_Rec_inv_V[elim]**:
assumes \( \Gamma \vdash e, eP, eV : \text{Fun } \tau_1 \tau_2 \)
and \( \text{atom } x \notin \Gamma \)
obtains \( y' \) \( eV' \) where \( e = \text{Rec } x (\text{Lam } y e') \) \( eP = \text{Rec } x (\text{Lam } y eP') \) \( eV = \text{Rec } x (\text{Lam } y eV') \) \( \text{atom } y \notin (\Gamma, x) \)
\( \Gamma(x \{ \alpha \} \vdash \text{Lam } y e', \text{Lam } y eP', \text{Lam } y eV' : \text{Fun } \tau_1 \tau_2) \)

\( \langle \text{proof} \rangle \)

**Inductive cases**

**Lemma a_Inj1_inv_I[elim]**:
\( \Gamma \vdash \text{Inj1 } e, eP, eV : \text{Sum } \tau_1 \tau_2 \)

**Lemma a_Inj1_inv_P[elim]**:
\( \Gamma \vdash e, eP, eV : \text{Sum } \tau_1 \tau_2 \)

**Lemma a_Inj1_inv_V[elim]**:
\( \Gamma \vdash e, eP, \text{Inj1 } eV : \text{Sum } \tau_1 \tau_2 \)

**Lemma a_Inj2_inv_I[elim]**:
\( \Gamma \vdash \text{Inj2 } e, eP, eV : \text{Sum } \tau_1 \tau_2 \)

**Lemma a_Inj2_inv_P[elim]**:
\( \Gamma \vdash e, eP, \text{Inj2 } eV : \text{Sum } \tau_1 \tau_2 \)

**Lemma a_Inj2_inv_V[elim]**:
\( \Gamma \vdash e, eP, \text{Inj2 } eV : \text{Sum } \tau_1 \tau_2 \)

**Lemma a_Pair_inv_I[elim]**:
\( \Gamma \vdash \text{Pair } e_1 e_2, eP, eV : \text{Prod } \tau_1 \tau_2 \)

**Lemma a_Pair_inv_P[elim]**:
\( \Gamma \vdash e, eP, eP_1 eP_2, eV : \text{Prod } \tau_1 \tau_2 \)

**Lemma a_Roll_inv_I[elim]**:
assumes \( \Gamma \vdash \text{Roll } e', eP, eV : \text{Mu } \alpha \tau \)
obtains \( eP' \) \( eV' \) where \( eP = \text{Roll } eP' eV = \text{Roll } eV' \) \( \Gamma \vdash e', eP', eV' : \text{subst_type } \tau (\text{Mu } \alpha \tau) \)

\( \langle \text{proof} \rangle \)

**Lemma a_Roll_inv_P[elim]**:
assumes \( \Gamma \vdash e, eP', eV : \text{Mu } \alpha \tau \)
obtains \( e' \) \( eV' \) where \( e = \text{Roll } e' eV = \text{Roll } eV' \) \( \Gamma \vdash e', eP', eV' : \text{subst_type } \tau (\text{Mu } \alpha \tau) \)

\( \langle \text{proof} \rangle \)

**Lemma a_Roll_inv_V[elim]**:
assumes \( \Gamma \vdash e, eP, \text{Roll } eV' : \text{Mu } \alpha \tau \)

20
obtains $e' \ eP'$
where $e = \text{Roll} \ e' \ eP = \text{Roll} \ eP' \ \Gamma \vdash e, eP', eV' : \text{subst} \_\text{type} \ \tau$ (Mu $\alpha \ \tau$) $\alpha$

\begin{proof}

\textbf{inductive cases a} \_\text{HashI} \_\text{inv} \_\text{elim}: \Gamma \vdash v, Hashed (hash (\|vP\|)) vP, Hash (hash (\|vP\|)) : AuthT $\tau$

Inversion on types for agreement.

\textbf{lemma a} \_\text{AuthT} \_\text{value} \_\text{inv}:
assumes\{\$$\} \vdash v, vP, vV : AuthT \tau

\begin{proof}

\textbf{declare} [|[simproc add: alpha_lst]|]

\textbf{lemma agree} \_\text{weakening} \_1:
assumes $\Gamma \vdash e, eP, eV : \tau$

\begin{proof}

\textbf{lemma agree} \_\text{weakening} \_2:
assumes $\Gamma \vdash e, eP, eV : \tau$

\begin{proof}

\textbf{lemma agree} \_\text{weakening} \_\text{env}:
assumes \{\$$\} \vdash e, eP, eV : \tau

\begin{proof}

\section{Formalization of Miller et al.'s [2] Main Results}

\textbf{lemma judge} \_\text{imp} \_\text{agree}:
assumes $\Gamma \vdash e : \tau$

\begin{proof}

\textbf{5.1 Lemma 2.1}

\textbf{lemma lemma2} \_1:
assumes $\Gamma \vdash e, eP, eV : \tau$

\begin{proof}

\textbf{5.2 Counterexample to Lemma 2.2}

\textbf{lemma lemma2} \_false:
fixes $x :: \text{var}$
assumes $\land \ \Gamma \vdash e \ eP \ eV \ \tau \ eP' \ eV'. [ \Gamma \vdash e, eP, eV : \tau; \Gamma \vdash e, eP', eV' : \tau ] \implies eP = eP' \land eV = eV'$

\begin{proof}
5.3 Lemma 2.3

lemma lemma2_3:
  assumes Γ ⊢ e, eP, eV : τ
  shows erase_env Γ ⊢̲ e : erase τ
  ⟨proof⟩

5.4 Lemma 2.4

lemma lemma2_4[dest]:
  assumes Γ ⊢ e, eP, eV : τ
  shows value e ∧ value eP ∧ value eV ∨ value e ∧ value eP ∧ ¬ value eV
  ⟨proof⟩

5.5 Lemma 3

lemma lemma3_general:
  fixes Γ :: tyenv and vs vPs vVs :: tenv
  assumes Γ ⊢ e, A ≤ fmdom Γ
  and fmdom vs = A fmdom vPs = A fmdom vVs = A
  and ∀ x. x ∈| A =⇒ (∃ τ' v vP h.
    Γ ?? x = Some (AuthT τ') ∧ vs ?? x = Some v ∧
    vPs ?? x = Some (Hashed h vP) ∧
    vVs ?? x = Some (Hashed h) ∧
    {??} ⊢ v. Hashed h vP. Hash h : (AuthT τ'))
  shows fmdrop_fset A Γ ⊢̲ psubst_term e vs, psubst_term e vPs, psubst_term e vVs : τ
  ⟨proof⟩

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, eP', eV' : τ'
  and value v value eP value eV
  shows Γ ⊢ e[v / x], eP'[vP / x], eV'[vV / x] : τ
  ⟨proof⟩

5.7 Lemma 5: Single-Step Correctness

lemma lemma5:
  assumes {??} ⊢ e, eP, eV : τ
  and ⟨[], e ⟩ I → ⟨[], e′ ⟩
  obtains eP'[eV' : π]
  V → ⟨π', eV' ⟩
5.8 Lemma 6: Single-Step Security

**Lemma lemma6:**
- **Assumes** \( \{ \$\} \vdash e, eP, eV : \tau \)
- **And** \( \ll \pi, \pi', eV \gg \mapsto V \ll \pi', eV' \gg \)
- **Obtains** \( e' eP' \pi \)
- **Where** \( \ll [], e \gg I \mapsto \ll [], e' \gg \equiv \pi P \ll \pi, eP \gg P \mapsto \ll \pi P \ll \pi, eP' \gg \)
- \( \{ \$\} \vdash e', eP', eV' : \tau \land \pi A \equiv \pi' \land \exists \ll s s', closed s \land closed s' \land \pi = [s] \land \pi A = [s'] \land s \neq s' \land \text{hash } s = \text{hash } s' \)

**Proof**

5.9 Theorem 1: Correctness

**Lemma theorem1_correctness:**
- **Assumes** \( \{ \$\} \vdash e, eP, eV : \tau \)
- **And** \( \ll [], e \gg I \mapsto \ll [], e' \gg \)
- **Obtains** \( eP' \ll eV' \pi \)
- **Where** \( \ll [], eP \gg P \mapsto i \ll \pi, eP' \gg \)
- \( \ll \pi, eV \gg V \mapsto i \ll [], eV' \gg \)
- \( \{ \$\} \vdash e', eP', eV' : \tau \)

**Proof**

5.10 Counterexamples to Theorem 1: Security

Counterexample using administrative normal form.

**Lemma security_false:**
- **Assumes** \( \{ \$\} \vdash e, eP, eV : \tau \;
- \( \{ \$\} \vdash e, eP, eV : \tau \ll \pi A, eV \gg \mapsto i \ll \pi', eV' \gg \)

**Proof**

Alternative, shorter counterexample not in administrative normal form.

**Lemma security_false_alt:**
- **Assumes** \( \{ \$\} \vdash e, eP, eV : \tau \ll \pi A, eV \gg \mapsto i \ll \pi', eV' \gg \)

**Proof**

5.11 Corrected Theorem 1: Security

**Lemma theorem1_security:**
- **Assumes** \( \{ \$\} \vdash e, eP, eV : \tau \)
- \( \ll \pi A, eV \gg \mapsto i \ll \pi', eV' \gg \)
- \( \{ \$\} \vdash e', eP', eV' : \tau \)

**Proof**

23
\((\exists e P \ j \ j_0 \ s \ s' \ j \leq i \land \ll \emptyset, eP \gg P \to j \ll \pi_0 \@ [s], eP' \gg \land \pi_A = \pi_0 \@ [s] \@ \pi_0' \@ \pi' \land \ s \neq s' \land \text{hash} \ s = \text{hash} \ s' \land \text{closed} \ s \land \text{closed} \ s')\)

\section{Remark 1}

\begin{lemma} \text{remark1_single}:\end{lemma}

\begin{assumes} \{\$\}\vdash e, eP, eV : \tau \and \ll \pi P, eP \gg P \to \ll \pi P \@ \pi, eP' \gg \obtains e' eV' where \{\$\}\vdash e', eP', eV' : \tau \land \ll \emptyset, e \gg I \to \ll \emptyset, e' \gg \land \ll \pi, eV \gg V \to \ll \emptyset, eV' \gg \end{assumes}

\begin{proof}\end{proof}

\begin{lemma} \text{remark1}:\end{lemma}

\begin{assumes} \{\$\}\vdash e, eP, eV : \tau \and \ll \pi P, eP \gg P \to i \ll \pi P \@ \pi, eP' \gg \obtains e' eV' where \{\$\}\vdash e', eP', eV' : \tau \ll \emptyset, e \gg I \to i \ll \emptyset, e' \gg \ll \pi, eV \gg V \to i \ll \emptyset, eV' \gg \end{assumes}

\begin{proof}\end{proof}

\section*{References}
