

Formalization of Generic Authenticated Data Structures

Matthias Brun Dmitriy Traytel

June 17, 2024

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 $\lambda\bullet$ (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 $\lambda\bullet$ 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 $\lambda\bullet$	4
3	Semantics of $\lambda\bullet$	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	21
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	22
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	23

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]: $\{\} \#* t$
<proof>

lemma *supp_flip*[simp]:
fixes $a b :: _ :: at$
shows $supp (a \leftrightarrow b) = (if\ a = b\ then\ \{\} \ else\ \{atom\ a,\ atom\ b\})$
<proof>

lemma *Abs_lst_eq_flipI*:
fixes $a b :: _ :: at$ **and** $t :: _ :: fs$
assumes $atom\ b \# t$
shows $[[atom\ a]]lst.\ t = [[atom\ b]]lst.\ (a \leftrightarrow b) \cdot t$
<proof>

lemma *atom_not_fresh_eq*:
assumes $\neg atom\ a \# x$
shows $a = x$
<proof>

lemma *fresh_set_fresh_forall*:
shows $atom\ y \# xs = (\forall x \in set\ xs.\ atom\ y \# x)$
<proof>

lemma *finite_fresh_set_fresh_all*[simp]:
fixes $S :: (_ :: fs)\ set$
shows $finite\ S \implies atom\ a \# S \longleftrightarrow (\forall x \in S.\ atom\ a \# x)$
<proof>

lemma *case_option_eqvt*[eqvt]:
 $p \cdot case_option\ a\ b\ opt = case_option\ (p \cdot a)\ (p \cdot b)\ (p \cdot opt)$
<proof>

Nominal setup for finite maps.

abbreviation *fmap_update* $(_ '(_ \ \$\$:= _') [1000,0,0] 1000)$ **where** *fmap_update* $\Gamma\ x\ \tau \equiv fmupd\ x\ \tau\ \Gamma$

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  $(p \cdot \{\$\$ \}) = \{\$\$ \}$ 
  ⟨proof⟩

lemma fmap_update_eqvt[eqvt]:
  shows  $(p \cdot f(a \ \$\$ = b)) = (p \cdot f)((p \cdot a) \ \$\$ = (p \cdot b))$ 
  ⟨proof⟩

lemma fmap_apply_eqvt[eqvt]:
  shows  $(p \cdot (f \ \$\$ b)) = (p \cdot f) \ \$\$ (p \cdot b)$ 
  ⟨proof⟩

lemma fresh_fmempty[simp]:
  shows  $a \ \#\ \{\$\$ \}$ 
  ⟨proof⟩

lemma fresh_fmap_update:
  shows  $\llbracket a \ \#\ f; a \ \#\ x; a \ \#\ y \rrbracket \implies a \ \#\ f(x \ \$\$ = y)$ 
  ⟨proof⟩

lemma supp_fmempty[simp]:
  shows  $\text{supp } \{\$\$ \} = \{\}$ 
  ⟨proof⟩

lemma supp_fmap_update:
  shows  $\text{supp } (f(x \ \$\$ = y)) \subseteq \text{supp}(f, x, y)$ 
  ⟨proof⟩

instance fmap :: (fs, fs) fs
  ⟨proof⟩

lemma fresh_transfer[transfer_rule]:
   $((=) \implies \text{pcr\_fmap } (=) (=) \implies (=)) \text{ fresh fresh}$ 
  ⟨proof⟩

lemma fmmmap_eqvt[eqvt]:  $p \cdot (\text{fmmmap } f F) = \text{fmmmap } (p \cdot f) (p \cdot F)$ 
  ⟨proof⟩

lemma fmap_freshness_lemma:
  fixes  $h :: ('a::at, 'b::pt) \text{ fmap}$ 
  assumes  $a: \exists a. \text{atom } a \ \#\ (h, h \ \$\$ a)$ 
  shows  $\exists x. \forall a. \text{atom } a \ \#\ h \longrightarrow h \ \$\$ a = x$ 
  ⟨proof⟩

lemma fmap_freshness_lemma_unique:
  fixes  $h :: ('a::at, 'b::pt) \text{ fmap}$ 
  assumes  $\exists a. \text{atom } a \ \#\ (h, h \ \$\$ a)$ 
  shows  $\exists !x. \forall a. \text{atom } a \ \#\ h \longrightarrow h \ \$\$ a = x$ 
  ⟨proof⟩

lemma fmdrop_fset_fmupd[simp]:
   $(\text{fmdrop\_fset } A f)(x \ \$\$ = y) = \text{fmdrop\_fset } (A \ |- \ \{|x|}) f(x \ \$\$ = y)$ 

```

including *fmap.lifting fset.lifting*
<proof>

lemma *fresh_fset_fminus*:
assumes *atom x # A*
shows $A \setminus \{|x|\} = A$
<proof>

lemma *fresh_fun_app*:
shows $\text{atom } x \# F \implies x \neq y \implies F y = \text{Some } a \implies \text{atom } x \# a$
<proof>

lemma *fresh_fmap_fresh_Some*:
 $\text{atom } x \# F \implies x \neq y \implies F \$\$ y = \text{Some } a \implies \text{atom } x \# a$
including *fmap.lifting*
<proof>

lemma *fmdrop_eqvt*: $p \cdot \text{fmdrop } x F = \text{fmdrop } (p \cdot x) (p \cdot F)$
<proof>

lemma *fmfilter_eqvt*: $p \cdot \text{fmfilter } Q F = \text{fmfilter } (p \cdot Q) (p \cdot F)$
<proof>

lemma *fmdrop_eq_iff*:
 $\text{fmdrop } x B = \text{fmdrop } y B \iff x = y \vee (x \notin \text{fmdom}' B \wedge y \notin \text{fmdom}' B)$
<proof>

lemma *fresh_fun_upd*:
shows $\llbracket a \# f; a \# x; a \# y \rrbracket \implies a \# f(x := y)$
<proof>

lemma *supp_fun_upd*:
shows $\text{supp } (f(x := y)) \subseteq \text{supp}(f, x, y)$
<proof>

lemma *map_drop_fun_upd*: $\text{map_drop } x F = F(x := \text{None})$
<proof>

lemma *fresh_fmdrop_in_fndom*: $\llbracket x \in \text{fmdom}' B; y \# B; y \# x \rrbracket \implies y \# \text{fmdrop } x B$
<proof>

lemma *fresh_fmdrop*:
assumes $x \# B \ x \# y$
shows $x \# \text{fmdrop } y B$
<proof>

lemma *fresh_fmdrop_fset*:
fixes $x :: \text{atom}$ **and** $A :: (_ :: \text{at_base}) \text{ fset}$
assumes $x \# A \ x \# B$
shows $x \# \text{fmdrop_fset } A B$
<proof>

2 Syntax of λ

typeddecl *hash*
instantiation *hash* :: *pure*
begin

definition *permute_hash* :: *perm* \Rightarrow *hash* \Rightarrow *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* |
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*)
<proof>

lemma *no_vars_in_ty*[*simp*]: *atom* (*x* :: *var*) $\#$ (τ :: *ty*)
<proof>

inductive *value* :: *term* \Rightarrow *bool* **where**
value *Unit* |
value (*Var* $_$) |
value (*Lam* $_ _$) |
value (*Rec* $_ _$) |
value *v* \Longrightarrow *value* (*Inj1* *v*) |
value *v* \Longrightarrow *value* (*Inj2* *v*) |
 \llbracket *value* *v*₁; *value* *v*₂ \rrbracket \Longrightarrow *value* (*Pair* *v*₁ *v*₂) |
value *v* \Longrightarrow *value* (*Roll* *v*) |
value (*Hash* $_$) |
value *v* \Longrightarrow *value* (*Hashed* $_$ *v*)

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

equivariance *value*

lemma *value_inv[simp]*:

¬ *value* (*Let* e_1 x e_2)
¬ *value* (*App* v v')
¬ *value* (*Case* v v_1 v_2)
¬ *value* (*Prj1* v)
¬ *value* (*Prj2* v)
¬ *value* (*Unroll* v)
¬ *value* (*Auth* v)
¬ *value* (*Unauth* v)
{*proof*}

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* e_1 e_2)

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

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

abbreviation *closed* :: *term* ⇒ *bool* **where**

closed t ≡ (∀ x ::*var.* *atom* x # t)

3 Semantics of $\lambda\bullet$

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 $y \in set$ 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)))

$\langle \text{proof} \rangle$
nominal_termination (*eqvt*)
 $\langle \text{proof} \rangle$

definition *hash where*
 $\text{hash } t = \text{hash_real } (\text{map_fixed } \text{undefined } [] t)$

lemma *permute_map_list*: $p \cdot l = \text{map } (\lambda x. p \cdot x) l$
 $\langle \text{proof} \rangle$

lemma *map_fixed_eqvt*: $p \cdot l = l \implies \text{map_fixed } v l (p \cdot t) = \text{map_fixed } v l t$
 $\langle \text{proof} \rangle$

lemma *hash_eqvt[eqvt]*: $p \cdot \text{hash } t = \text{hash } (p \cdot t)$
 $\langle \text{proof} \rangle$

lemma *map_fixed_idle*: $\{x. \neg \text{atom } x \# t\} \subseteq \text{set } l \implies \text{map_fixed } v l t = t$
 $\langle \text{proof} \rangle$

lemma *map_fixed_idle_closed*:
 $\text{closed } t \implies \text{map_fixed } \text{undefined } [] t = t$
 $\langle \text{proof} \rangle$

lemma *map_fixed_inj_closed*:
 $\text{closed } t \implies \text{closed } u \implies \text{map_fixed } \text{undefined } [] t = \text{map_fixed } \text{undefined } [] u \implies t = u$
 $\langle \text{proof} \rangle$

lemma *hash_eq_hash_real_closed*:
assumes *closed t*
shows $\text{hash } t = \text{hash_real } t$
 $\langle \text{proof} \rangle$

3.2 Substitution

nominal_function *subst_term* :: $\text{term} \Rightarrow \text{term} \Rightarrow \text{var} \Rightarrow \text{term} \ (_ _ _ / _)$ [250, 200, 200] 250) **where**

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

$\langle \text{proof} \rangle$
nominal_termination (*eqvt*)
 $\langle \text{proof} \rangle$

type_synonym *tenv* = $(\text{var}, \text{term}) \text{ fmap}$

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* \Longrightarrow *psubst_term* (*Lam* *y* *t*) *f* = *Lam* *y* (*psubst_term* *t* *f*) |
atom *y* # *f* \Longrightarrow *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* \Longrightarrow *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* \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*) \Longrightarrow *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*] \longleftrightarrow (*x* = *x'* \vee *atom* *x* # *t*) \wedge (*atom* *x'* # *t* \vee *atom* *x* # *t'*)
<proof>

lemma *term_fresh_subst[simp]*: *atom* *x* # *t* \Longrightarrow *atom* *x* # *s* \Longrightarrow (*atom* (*x::var*)) # *t*[*s / y*]
<proof>

lemma *term_subst_idle[simp]*: *atom* *y* # *t* \Longrightarrow *t*[*s / y*] = *t*
<proof>

lemma *term_subst_subst*: *atom* *y1* \neq *atom* *y2* \Longrightarrow *atom* *y1* # *s2* \Longrightarrow *t*[*s1 / y1*][*s2 / y2*] = *t*[*s2 / y2*][*s1*[*s2 / y2*] / *y1*]
<proof>

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' \longleftrightarrow ((α = α' \vee *atom* α # τ) \wedge (*atom* α' # τ \vee *atom* α # τ'))
<proof>

lemma *type_fresh_subst*[simp]: $atom\ x \# t \implies atom\ x \# s \implies (atom\ (x::tvar)) \# subst_type\ t\ s\ y$
 ⟨proof⟩

lemma *type_subst_idle*[simp]: $atom\ y \# t \implies subst_type\ t\ s\ y = t$
 ⟨proof⟩

lemma *type_subst_subst*: $atom\ y1 \neq atom\ y2 \implies atom\ y1 \# s2 \implies$
 $subst_type\ (subst_type\ t\ s1\ y1)\ s2\ y2 = subst_type\ (subst_type\ t\ s2\ y2)\ (subst_type\ s1\ s2\ y2)\ y1$
 ⟨proof⟩

3.3 Weak Typing Judgement

type_synonym *tyenv* = (var, ty) fmap

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

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

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

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

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

<proof>

lemma *swap_permute_swap*: $\text{atom } x \# \pi \implies \text{atom } y \# \pi \implies (x \leftrightarrow y) \cdot \pi \cdot (x \leftrightarrow y) \cdot t = \pi \cdot t$

<proof>

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

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

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

<proof>

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

<proof>

lemma *jw_Roll_inv[elim]*:

assumes $\Gamma \vdash_W \text{Roll } e : \tau$

and $\text{atom } \alpha \# (\Gamma, \tau)$

obtains τ' **where** $\tau = \text{Mu } \alpha \ \tau' \ \Gamma \vdash_W \ e : \text{subst_type } \tau' \ (\text{Mu } \alpha \ \tau') \ \alpha$

<proof>

lemma *jw_Unroll_inv[elim]*:

assumes $\Gamma \vdash_W \text{Unroll } e : \tau$

and $\text{atom } \alpha \# (\Gamma, \tau)$

obtains τ' **where** $\tau = \text{subst_type } \tau' \ (\text{Mu } \alpha \ \tau') \ \alpha \ \Gamma \vdash_W \ e : \text{Mu } \alpha \ \tau'$

<proof>

Additional inversion rules based on type rather than term.

inductive_cases *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 $e \ x \ \text{where } v = \text{Lam } x \ e \ \vee \ v = \text{Rec } x \ e \ \text{atom } x \ \# (c::\text{term})$
 $\langle \text{proof} \rangle$

lemma *jw_Mu_inv[elim]*:

assumes $\{\$\$\} \vdash_W v : \text{Mu } \alpha \ \tau \ \text{value } v$

obtains $v' \ \text{where } v = \text{Roll } v'$
 $\langle \text{proof} \rangle$

3.4 Erasure of Authenticated Types

nominal_function *erase* :: $ty \Rightarrow ty$ **where**

$\text{erase } \text{One} = \text{One} \mid$
 $\text{erase } (\text{Fun } \tau_1 \tau_2) = \text{Fun } (\text{erase } \tau_1) (\text{erase } \tau_2) \mid$
 $\text{erase } (\text{Sum } \tau_1 \tau_2) = \text{Sum } (\text{erase } \tau_1) (\text{erase } \tau_2) \mid$
 $\text{erase } (\text{Prod } \tau_1 \tau_2) = \text{Prod } (\text{erase } \tau_1) (\text{erase } \tau_2) \mid$
 $\text{erase } (\text{Mu } \alpha \ \tau) = \text{Mu } \alpha \ (\text{erase } \tau) \mid$
 $\text{erase } (\text{Alpha } \alpha) = \text{Alpha } \alpha \mid$
 $\text{erase } (\text{AuthT } \tau) = \text{erase } \tau$
 $\langle \text{proof} \rangle$

nominal_termination (*eqvt*)

$\langle \text{proof} \rangle$

lemma *fresh_erase_fresh*:

assumes $\text{atom } x \ \# \ \tau$

shows $\text{atom } x \ \# \ \text{erase } \tau$
 $\langle \text{proof} \rangle$

lemma *fresh_fmmap_erase_fresh*:

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

shows $\text{atom } x \ \# \ \text{fmmap } \text{erase } \Gamma$
 $\langle \text{proof} \rangle$

lemma *erase_subst_type_shift[simp]*:

$\text{erase } (\text{subst_type } \tau \ \tau' \ \alpha) = \text{subst_type } (\text{erase } \tau) (\text{erase } \tau') \ \alpha$
 $\langle \text{proof} \rangle$

definition *erase_env* :: $tyenv \Rightarrow tyenv$ **where**

$\text{erase_env} = \text{fmmap } \text{erase}$

3.5 Strong Typing Judgement

inductive *judge* :: $tyenv \Rightarrow \text{term} \Rightarrow ty \Rightarrow \text{bool}$ ($_ \vdash _ : _ [150,0,150] 149$) **where**

j_Unit: $\Gamma \vdash \text{Unit} : \text{One} \mid$

j_Var: $\llbracket \Gamma \ \$\$ \ x = \text{Some } \tau \rrbracket$
 $\implies \Gamma \vdash \text{Var } x : \tau \mid$

j_Lam: $\llbracket \text{atom } x \ \# \ \Gamma; \Gamma(x \ \$\$:= \tau_1) \vdash e : \tau_2 \rrbracket$
 $\implies \Gamma \vdash \text{Lam } x \ e : \text{Fun } \tau_1 \tau_2 \mid$

j_App: $\llbracket \Gamma \vdash e : \text{Fun } \tau_1 \tau_2; \Gamma \vdash e' : \tau_1 \rrbracket$
 $\implies \Gamma \vdash \text{App } e \ e' : \tau_2 \mid$

j_Let: $\llbracket \text{atom } x \ \# \ (\Gamma, e_1); \Gamma \vdash e_1 : \tau_1; \Gamma(x \ \$\$:= \tau_1) \vdash e_2 : \tau_2 \rrbracket$
 $\implies \Gamma \vdash \text{Let } e_1 \ x \ e_2 : \tau_2 \mid$

j_Rec: $\llbracket \text{atom } x \ \# \ \Gamma; \text{atom } y \ \# \ (\Gamma, x); \Gamma(x \ \$\$:= \text{Fun } \tau_1 \tau_2) \vdash \text{Lam } y \ e' : \text{Fun } \tau_1 \tau_2 \rrbracket$
 $\implies \Gamma \vdash \text{Rec } x \ (\text{Lam } y \ e') : \text{Fun } \tau_1 \tau_2 \mid$

j_Inj1: $\llbracket \Gamma \vdash e : \tau_1 \rrbracket$

$\implies \Gamma \vdash \text{Inj1 } e : \text{Sum } \tau_1 \tau_2 \mid$

```

j_Inj2:  [ Γ ⊢ e : τ2 ]
        ⇒ Γ ⊢ Inj2 e : Sum τ1 τ2 |
j_Case:  [ Γ ⊢ e : Sum τ1 τ2; Γ ⊢ e1 : Fun τ1 τ; Γ ⊢ e2 : Fun τ2 τ ]
        ⇒ Γ ⊢ Case e e1 e2 : τ |
j_Pair:  [ Γ ⊢ e1 : τ1; Γ ⊢ e2 : τ2 ]
        ⇒ Γ ⊢ Pair e1 e2 : Prod τ1 τ2 |
j_Prj1:  [ Γ ⊢ e : Prod τ1 τ2 ]
        ⇒ Γ ⊢ Prj1 e : τ1 |
j_Prj2:  [ Γ ⊢ e : Prod τ1 τ2 ]
        ⇒ Γ ⊢ Prj2 e : τ2 |
j_Roll:  [ atom α ‡ Γ; Γ ⊢ e : subst_type τ (Mu α τ) α ]
        ⇒ Γ ⊢ Roll e : Mu α τ |
j_Unroll: [ atom α ‡ Γ; Γ ⊢ e : Mu α τ ]
        ⇒ Γ ⊢ Unroll e : subst_type τ (Mu α τ) α |
j_Auth:  [ Γ ⊢ e : τ ]
        ⇒ Γ ⊢ Auth e : AuthT τ |
j_Unauth: [ Γ ⊢ e : AuthT τ ]
        ⇒ Γ ⊢ Unauth e : τ

```

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
⟨proof⟩

```

lemma *judge_imp_judge_weak*:

```

assumes Γ ⊢ e : τ
shows   erase_env Γ ⊢W e : erase τ
⟨proof⟩

```

3.6 Shallow Projection

nominal_function *shallow* :: *term* ⇒ *term* (*(|_)*) **where**

```

(|Unit|) = Unit |
(|Var v|) = Var v |
(|Lam x e|) = Lam x (|e|) |
(|Rec x e|) = Rec x (|e|) |
(|Inj1 e|) = Inj1 (|e|) |
(|Inj2 e|) = Inj2 (|e|) |
(|Pair e1 e2|) = Pair (|e1|) (|e2|) |
(|Roll e|) = Roll (|e|) |
(|Let e1 x e2|) = Let (|e1|) x (|e2|) |
(|App e1 e2|) = App (|e1|) (|e2|) |
(|Case e e1 e2|) = Case (|e|) (|e1|) (|e2|) |
(|Prj1 e|) = Prj1 (|e|) |
(|Prj2 e|) = Prj2 (|e|) |
(|Unroll e|) = Unroll (|e|) |
(|Auth e|) = Auth (|e|) |
(|Unauth e|) = Unauth (|e|) |
— No rule is defined for Hash, but: "[...] preserving that structure in every case but that of <h, v> [...]"
(|Hash h|) = Hash h |
(|Hashed h e|) = Hash h
⟨proof⟩

```


$s_Roll: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\quad \Rightarrow \ll \pi, Roll\ e \gg m \rightarrow \ll \pi', Roll\ e' \gg \mid$
 $s_UnrollRoll: \ll \text{value } v \gg$
 $\quad \Rightarrow \ll \pi, Unroll\ (Roll\ v) \gg _ \rightarrow \ll \pi, v \gg \mid$
— Mode-specific rules
 $s_Auth: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\quad \Rightarrow \ll \pi, Auth\ e \gg m \rightarrow \ll \pi', Auth\ e' \gg \mid$
 $s_Unauth: \quad \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$
 $\quad \Rightarrow \ll \pi, Unauth\ e \gg m \rightarrow \ll \pi', Unauth\ e' \gg \mid$
 $s_AuthI: \quad \ll \text{value } v \gg$
 $\quad \Rightarrow \ll \pi, Auth\ v \gg I \rightarrow \ll \pi, v \gg \mid$
 $s_UnauthI: \quad \ll \text{value } v \gg$
 $\quad \Rightarrow \ll \pi, Unauth\ v \gg I \rightarrow \ll \pi, v \gg \mid$
 $s_AuthP: \quad \ll \text{closed } (v); \text{value } v \gg$
 $\quad \Rightarrow \ll \pi, Auth\ v \gg P \rightarrow \ll \pi, Hashed\ (hash\ (v))\ v \gg \mid$
 $s_UnauthP: \quad \ll \text{value } v \gg$
 $\quad \Rightarrow \ll \pi, Unauth\ (Hashed\ h\ v) \gg P \rightarrow \ll \pi @ [(v)], v \gg \mid$
 $s_AuthV: \quad \ll \text{closed } v; \text{value } v \gg$
 $\quad \Rightarrow \ll \pi, Auth\ v \gg V \rightarrow \ll \pi, Hash\ (hash\ v) \gg \mid$
 $s_UnauthV: \quad \ll \text{closed } s_0; hash\ s_0 = h \gg$
 $\quad \Rightarrow \ll s_0 \# \pi, Unauth\ (Hash\ h) \gg V \rightarrow \ll \pi, s_0 \gg$

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

equivariance *smallstep*
nominal_inductive *smallstep*
avoids *s_AppLam*: *x*
 \mid *s_AppRec*: *x*
 \mid *s_Let1*: *x*
 \mid *s_Let2*: *x*
<proof>

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

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

equivariance *smallsteps*
nominal_inductive *smallsteps* *<proof>*

lemma *steps_1_step*[simp]: $\ll \pi, e \gg m \rightarrow 1 \ll \pi', e' \gg = \ll \pi, e \gg m \rightarrow \ll \pi', e' \gg$ (**is** ?L \leftrightarrow ?R)
<proof>

Inversion rules for smallstep(s) predicates.

lemma *value_no_step*[intro]:
assumes $\ll \pi_1, v \gg m \rightarrow \ll \pi_2, t \gg$ *value* *v*
shows *False*
<proof>

lemma *subst_term_perm*:
assumes *atom* $x' \# (x, e)$
shows $e[v / x] = ((x \leftrightarrow x') \cdot e)[v / x]$
<proof>

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 $\ll \pi, Let\ e_1\ x\ e_2 \gg m \rightarrow \ll \pi', e' \gg$
and $atom\ x \# (e_1, \pi)$
obtains e_1' **where** $(e' = e_2[e_1 / x] \wedge value\ e_1 \wedge \pi = \pi') \vee (\ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg \wedge e' = Let\ e_1'\ x\ e_2 \wedge \neg value\ e_1)$
<proof>

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

assumes $\ll \pi, Let\ e_1\ x\ e_2 \gg m \rightarrow \ll \pi', e' \gg$
and $atom\ x \# (e_1, \pi)$
and $e' = e_2[e_1 / x] \wedge value\ e_1 \wedge \pi = \pi' \implies Q$
and $\bigwedge e_1'. \ll \pi, e_1 \gg m \rightarrow \ll \pi', e_1' \gg \wedge e' = Let\ e_1'\ x\ e_2 \wedge \neg value\ e_1 \implies Q$
shows Q
<proof>

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 $atom\ x \# e'$

<proof>

lemma *fresh_smallstep_P*:

fixes $x :: \text{var}$
assumes $\ll \pi, e \gg P \rightarrow \ll \pi', e' \gg \text{atom } x \# e$
shows $\text{atom } x \# e'$
<proof>

lemma *fresh_smallsteps_I*:

fixes $x :: \text{var}$
assumes $\ll \pi, e \gg I \rightarrow i \ll \pi', e' \gg \text{atom } x \# e$
shows $\text{atom } x \# e'$
<proof>

lemma *fresh_ps_smallstep_P*:

fixes $x :: \text{var}$
assumes $\ll \pi, e \gg P \rightarrow \ll \pi', e' \gg \text{atom } x \# e \text{ atom } x \# \pi$
shows $\text{atom } x \# \pi'$
<proof>

Proofstream lemmas.

lemma *smallstepI_ps_eq*:

assumes $\ll \pi, e \gg I \rightarrow \ll \pi', e' \gg$
shows $\pi = \pi'$
<proof>

lemma *smallstepI_ps_emptyD*:

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

lemma *smallstepsI_ps_eq*:

assumes $\ll \pi, e \gg I \rightarrow i \ll \pi', e' \gg$
shows $\pi = \pi'$
<proof>

lemma *smallstepsI_ps_emptyD*:

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

lemma *smallstepV_consumes_proofstream*:

assumes $\ll \pi_1, eV \gg V \rightarrow \ll \pi_2, eV' \gg$
obtains π **where** $\pi_1 = \pi @ \pi_2$
<proof>

lemma *smallstepsV_consumes_proofstream*:

assumes $\ll \pi_1, eV \gg V \rightarrow i \ll \pi_2, eV' \gg$
obtains π **where** $\pi_1 = \pi @ \pi_2$
<proof>

lemma *smallstepP_generates_proofstream*:

assumes $\ll \pi_1, eP \gg P \rightarrow \ll \pi_2, eP' \gg$
obtains π **where** $\pi_2 = \pi_1 @ \pi$
<proof>

lemma *smallstepsP_generates_proofstream*:

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

obtains π **where** $\pi_2 = \pi_1 @ \pi$
 ⟨proof⟩

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⟩

lemma *smallstepV_ps_to_suffix*:
assumes $\ll \pi, e \gg V \rightarrow \ll \pi' @ X, e' \gg$
obtains π'' **where** $\pi = \pi'' @ X$
 ⟨proof⟩

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⟩

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⟩

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⟩

3.8 Type Progress

lemma *type_progress*:
assumes $\{\$\$ \} \vdash_W e : \tau$
shows $\text{value } e \vee (\exists e'. \ll [], e \gg I \rightarrow \ll [], e' \gg)$
 ⟨proof⟩

3.9 Weak Type Preservation

lemma *fresh_tyenv_None*:
fixes $\Gamma :: \text{tyenv}$
shows $\text{atom } x \# \Gamma \longleftrightarrow \Gamma \ \$\$ x = \text{None}$ (**is** ?L \longleftrightarrow ?R)
 ⟨proof⟩

lemma *judge_weak_fresh_env_fresh_term[dest]*:
fixes $a :: \text{var}$
assumes $\Gamma \vdash_W e : \tau$ $\text{atom } a \# \Gamma$
shows $\text{atom } a \# e$
 ⟨proof⟩

lemma *judge_weak_weakening_1*:
assumes $\Gamma \vdash_W e : \tau$ $\text{atom } y \# e$
shows $\Gamma(y \ \$\$:= \tau') \vdash_W e : \tau$
 ⟨proof⟩

lemma *judge_weak_weakening_2*:
assumes $\Gamma \vdash_W e : \tau$ $\text{atom } y \# \Gamma$
shows $\Gamma(y \ \$\$:= \tau') \vdash_W e : \tau$
 ⟨proof⟩

lemma *judge_weak_weakening_env*:
assumes $\{\$\$ \} \vdash_W e : \tau$
shows $\Gamma \vdash_W e : \tau$
 ⟨proof⟩

lemma *value_subst_value*:

assumes *value e value e'*
shows *value (e[e' / x])*
<proof>

lemma *judge_weak_subst[intro]*:

assumes $\Gamma(a \text{ \textit{\$} \$} = \tau') \vdash_W e : \tau \text{ \textit{\$} \$} \vdash_W e' : \tau'$
shows $\Gamma \vdash_W e[e' / a] : \tau$
<proof>

lemma *type_preservation*:

assumes $\ll [], e \gg I \rightarrow \ll [], e' \gg \text{ \textit{\$} \$} \vdash_W e : \tau$
shows $\text{ \textit{\$} \$} \vdash_W e' : \tau$
<proof>

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

lemma *type_soundness*:

assumes $\text{ \textit{\$} \$} \vdash_W e : \tau$
shows *value e* $\vee (\exists e'. \ll [], e \gg I \rightarrow \ll [], e' \gg \wedge \text{ \textit{\$} \$} \vdash_W e' : \tau)$
<proof>

4 Agreement Relation

inductive *agree* :: *tyenv* \Rightarrow *term* \Rightarrow *term* \Rightarrow *term* \Rightarrow *ty* \Rightarrow *bool* ($_ \vdash _$, $_$, $_$: $_ [150,0,0,0,150]$ 149)
where

a_Unit: $\Gamma \vdash \textit{Unit}, \textit{Unit}, \textit{Unit} : \textit{One} \mid$
a_Var: $\Gamma \text{ \textit{\$} \$} x = \textit{Some} \tau$
 $\Rightarrow \Gamma \vdash \textit{Var} x, \textit{Var} x, \textit{Var} x : \tau \mid$
a_Lam: $\ll \textit{atom} x \# \Gamma; \Gamma(x \text{ \textit{\$} \$} = \tau_1) \vdash e, eP, eV : \tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Lam} x e, \textit{Lam} x eP, \textit{Lam} x eV : \textit{Fun} \tau_1 \tau_2 \mid$
a_App: $\ll \Gamma \vdash e_1, eP_1, eV_1 : \textit{Fun} \tau_1 \tau_2; \Gamma \vdash e_2, eP_2, eV_2 : \tau_1 \gg$
 $\Rightarrow \Gamma \vdash \textit{App} e_1 e_2, \textit{App} eP_1 eP_2, \textit{App} eV_1 eV_2 : \tau_2 \mid$
a_Let: $\ll \textit{atom} x \# (\Gamma, e_1, eP_1, eV_1); \Gamma \vdash e_1, eP_1, eV_1 : \tau_1; \Gamma(x \text{ \textit{\$} \$} = \tau_1) \vdash e_2, eP_2, eV_2 : \tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Let} e_1 x e_2, \textit{Let} eP_1 x eP_2, \textit{Let} eV_1 x eV_2 : \tau_2 \mid$
a_Rec: $\ll \textit{atom} x \# \Gamma; \textit{atom} y \# (\Gamma, x); \Gamma(x \text{ \textit{\$} \$} = \textit{Fun} \tau_1 \tau_2) \vdash \textit{Lam} y e, \textit{Lam} y eP, \textit{Lam} y eV : \textit{Fun} \tau_1$
 $\tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Rec} x (\textit{Lam} y e), \textit{Rec} x (\textit{Lam} y eP), \textit{Rec} x (\textit{Lam} y eV) : \textit{Fun} \tau_1 \tau_2 \mid$
a_Inj1: $\ll \Gamma \vdash e, eP, eV : \tau_1 \gg$
 $\Rightarrow \Gamma \vdash \textit{Inj1} e, \textit{Inj1} eP, \textit{Inj1} eV : \textit{Sum} \tau_1 \tau_2 \mid$
a_Inj2: $\ll \Gamma \vdash e, eP, eV : \tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Inj2} e, \textit{Inj2} eP, \textit{Inj2} eV : \textit{Sum} \tau_1 \tau_2 \mid$
a_Case: $\ll \Gamma \vdash e, eP, eV : \textit{Sum} \tau_1 \tau_2; \Gamma \vdash e_1, eP_1, eV_1 : \textit{Fun} \tau_1 \tau; \Gamma \vdash e_2, eP_2, eV_2 : \textit{Fun} \tau_2 \tau \gg$
 $\Rightarrow \Gamma \vdash \textit{Case} e e_1 e_2, \textit{Case} eP eP_1 eP_2, \textit{Case} eV eV_1 eV_2 : \tau \mid$
a_Pair: $\ll \Gamma \vdash e_1, eP_1, eV_1 : \tau_1; \Gamma \vdash e_2, eP_2, eV_2 : \tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Pair} e_1 e_2, \textit{Pair} eP_1 eP_2, \textit{Pair} eV_1 eV_2 : \textit{Prod} \tau_1 \tau_2 \mid$
a_Prj1: $\ll \Gamma \vdash e, eP, eV : \textit{Prod} \tau_1 \tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Prj1} e, \textit{Prj1} eP, \textit{Prj1} eV : \tau_1 \mid$
a_Prj2: $\ll \Gamma \vdash e, eP, eV : \textit{Prod} \tau_1 \tau_2 \gg$
 $\Rightarrow \Gamma \vdash \textit{Prj2} e, \textit{Prj2} eP, \textit{Prj2} eV : \tau_2 \mid$
a_Roll: $\ll \textit{atom} \alpha \# \Gamma; \Gamma \vdash e, eP, eV : \textit{subst_type} \tau (\textit{Mu} \alpha \tau) \alpha \gg$
 $\Rightarrow \Gamma \vdash \textit{Roll} e, \textit{Roll} eP, \textit{Roll} eV : \textit{Mu} \alpha \tau \mid$
a_Unroll: $\ll \textit{atom} \alpha \# \Gamma; \Gamma \vdash e, eP, eV : \textit{Mu} \alpha \tau \gg$
 $\Rightarrow \Gamma \vdash \textit{Unroll} e, \textit{Unroll} eP, \textit{Unroll} eV : \textit{subst_type} \tau (\textit{Mu} \alpha \tau) \alpha \mid$
a_Auth: $\ll \Gamma \vdash e, eP, eV : \tau \gg$
 $\Rightarrow \Gamma \vdash \textit{Auth} e, \textit{Auth} eP, \textit{Auth} eV : \textit{AuthT} \tau \mid$
a_Unauth: $\ll \Gamma \vdash e, eP, eV : \textit{AuthT} \tau \gg$

$\implies \Gamma \vdash \text{Unauth } e, \text{Unauth } eP, \text{Unauth } eV : \tau \mid$
 $a_HashI: \llbracket \{\$\$ \} \vdash v, vP, (vP) : \tau; \text{hash } (vP) = h; \text{value } v; \text{value } vP \rrbracket$
 $\implies \Gamma \vdash v, \text{Hashed } h \ vP, \text{Hash } h : \text{AuthT } \tau$

declare *agree.intros*[*intro*]

equivariance *agree*

nominal_inductive *agree*

avoids *a_Lam*: *x*
 \mid *a_Rec*: *x* **and** *y*
 \mid *a_Let*: *x*
 \mid *a_Roll*: α
 \mid *a_Unroll*: α
<proof>

lemma *Abs_lst_eq_3tuple*:

fixes *x x' :: var*
fixes *e eP eV e' eP' eV' :: term*
assumes $\llbracket \text{atom } x \rrbracket \text{lst. } e = \llbracket \text{atom } x' \rrbracket \text{lst. } e'$
and $\llbracket \text{atom } x \rrbracket \text{lst. } eP = \llbracket \text{atom } x' \rrbracket \text{lst. } eP'$
and $\llbracket \text{atom } x \rrbracket \text{lst. } eV = \llbracket \text{atom } x' \rrbracket \text{lst. } eV'$
shows $\llbracket \text{atom } x \rrbracket \text{lst. } (e, eP, eV) = \llbracket \text{atom } x' \rrbracket \text{lst. } (e', eP', eV')$
<proof>

lemma *agree_fresh_env_fresh_term*:

fixes *a :: var*
assumes $\Gamma \vdash e, eP, eV : \tau$ *atom* *a* $\# \Gamma$
shows *atom* *a* $\# (e, eP, eV)$
<proof>

lemma *agree_empty_fresh*[*dest*]:

fixes *a :: var*
assumes $\{\$\$ \} \vdash e, eP, eV : \tau$
shows $\{\text{atom } a\} \#* \{e, eP, eV\}$
<proof>

Inversion rules for agreement.

declare $\llbracket \text{simproc del: } \alpha _ \text{lst} \rrbracket$

lemma *a_Lam_inv_I*[*elim*]:

assumes $\Gamma \vdash (\text{Lam } x \ e'), eP, eV : (\text{Fun } \tau_1 \ \tau_2)$
and *atom* *x* $\# \Gamma$
obtains *eP' eV'* **where** $eP = \text{Lam } x \ eP' \ eV = \text{Lam } x \ eV' \ \Gamma(x \ \$\$:= \tau_1) \vdash e', eP', eV' : \tau_2$
<proof>

lemma *a_Lam_inv_P*[*elim*]:

assumes $\{\$\$ \} \vdash v, (\text{Lam } x \ vP'), vV : (\text{Fun } \tau_1 \ \tau_2)$
obtains *v' vV'* **where** $v = \text{Lam } x \ v' \ vV = \text{Lam } x \ vV' \ \{\$\$ \}(x \ \$\$:= \tau_1) \vdash v', vP', vV' : \tau_2$
<proof>

lemma *a_Lam_inv_V*[*elim*]:

assumes $\{\$\$ \} \vdash v, vP, (\text{Lam } x \ vV') : (\text{Fun } \tau_1 \ \tau_2)$
obtains *v' vP'* **where** $v = \text{Lam } x \ v' \ vP = \text{Lam } x \ vP' \ \{\$\$ \}(x \ \$\$:= \tau_1) \vdash v', vP', vV' : \tau_2$
<proof>

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 $e = Lam\ y\ e' eP = Rec\ x\ (Lam\ y\ eP')\ eV = Rec\ x\ (Lam\ y\ eV')\ atom\ y\ \# (\Gamma, x)$
 $\Gamma(x\ \$\$:= Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e', Lam\ y\ eP', Lam\ y\ eV' : Fun\ \tau_1\ \tau_2$
 $\langle proof \rangle$

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 $e = Rec\ x\ (Lam\ y\ e')\ eP = Lam\ y\ eP'\ eV = Rec\ x\ (Lam\ y\ eV')\ atom\ y\ \# (\Gamma, x)$
 $\Gamma(x\ \$\$:= Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e', Lam\ y\ eP', Lam\ y\ eV' : Fun\ \tau_1\ \tau_2$
 $\langle proof \rangle$

lemma $a_Rec_inv_V[elim]$:
assumes $\Gamma \vdash e, eP, Rec\ x\ eV : Fun\ \tau_1\ \tau_2$
and $atom\ x\ \# \Gamma$
obtains $y e' eP' eV'$
where $e = Rec\ x\ (Lam\ y\ e')\ eP = Rec\ x\ (Lam\ y\ eP')\ eV = Lam\ y\ eV'\ atom\ y\ \# (\Gamma, x)$
 $\Gamma(x\ \$\$:= Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e', Lam\ y\ eP', Lam\ y\ eV' : Fun\ \tau_1\ \tau_2$
 $\langle proof \rangle$

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

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

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

lemma $a_Roll_inv_I[elim]$:
assumes $\Gamma \vdash Roll\ e', eP, eV : Mu\ \alpha\ \tau$
obtains $eP' eV'$
where $eP = Roll\ eP'\ eV = Roll\ eV'\ \Gamma \vdash e', eP', eV' : subst_type\ \tau\ (Mu\ \alpha\ \tau)\ \alpha$
 $\langle proof \rangle$

lemma $a_Roll_inv_P[elim]$:
assumes $\Gamma \vdash e, Roll\ eP', eV : Mu\ \alpha\ \tau$
obtains $e' eV'$
where $e = Roll\ e'\ eV = Roll\ eV'\ \Gamma \vdash e', eP', eV' : subst_type\ \tau\ (Mu\ \alpha\ \tau)\ \alpha$
 $\langle proof \rangle$

lemma $a_Roll_inv_V[elim]$:
assumes $\Gamma \vdash e, eP, Roll\ eV' : Mu\ \alpha\ \tau$
obtains $e' eP'$
where $e = Roll\ e'\ eP = Roll\ eP'\ \Gamma \vdash e', eP', eV' : subst_type\ \tau\ (Mu\ \alpha\ \tau)\ \alpha$
 $\langle proof \rangle$

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

Inversion on types for agreement.

lemma $a_AuthT_value_inv$:
assumes $\{\$\$ \} \vdash v, vP, vV : AuthT\ \tau$
and $value\ v\ value\ vP\ value\ vV$
obtains vP' **where** $vP = Hashed\ (hash\ (vP'))\ vP'\ vV = Hash\ (hash\ (vP'))\ value\ vP'$
 $\langle proof \rangle$

inductive_cases *a_Mu_inv*[*elim*]: $\Gamma \vdash e, eP, eV : Mu \ \alpha \ \tau$
inductive_cases *a_Sum_inv*[*elim*]: $\Gamma \vdash e, eP, eV : Sum \ \tau_1 \ \tau_2$
inductive_cases *a_Prod_inv*[*elim*]: $\Gamma \vdash e, eP, eV : Prod \ \tau_1 \ \tau_2$
inductive_cases *a_Fun_inv*[*elim*]: $\Gamma \vdash e, eP, eV : Fun \ \tau_1 \ \tau_2$

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 $\Gamma(y \ \$\$:= \tau') \vdash e, eP, eV : \tau$

<proof>

lemma *agree_weakening_2*:

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

<proof>

lemma *agree_weakening_env*:

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

<proof>

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

lemma *judge_imp_agree*:

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

<proof>

5.1 Lemma 2.1

lemma *lemma2_1*:

assumes $\Gamma \vdash e, eP, eV : \tau$
shows $(eP) = eV$

<proof>

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>

lemma *smallstep_ideal_deterministic*:

$\llbracket [], t \rrbracket I \rightarrow \llbracket [], u \rrbracket \implies \llbracket [], t \rrbracket I \rightarrow \llbracket [], u' \rrbracket \implies u = u'$

<proof>

lemma *smallsteps_ideal_deterministic*:

$\llbracket [], t \rrbracket I \rightarrow i \llbracket [], u \rrbracket \implies \llbracket [], t \rrbracket I \rightarrow i \llbracket [], u' \rrbracket \implies u = u'$

<proof>

5.3 Lemma 2.3

lemma *lemma2_3*:

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

shows $erase_env \Gamma \vdash_W e : erase \tau$
 $\langle proof \rangle$

5.4 Lemma 2.4

lemma $lemma2_4[dest]$:
assumes $\Gamma \vdash e, eP, eV : \tau$
shows $value e \wedge value eP \wedge value eV \vee \neg value e \wedge \neg value eP \wedge \neg value eV$
 $\langle proof \rangle$

5.5 Lemma 3

lemma $lemma3_general$:
fixes $\Gamma :: tyenv$ **and** $vs vPs vVs :: tenu$
assumes $\Gamma \vdash e : \tau$ $A \mid \subseteq \mid fmdom \Gamma$
and $fmdom vs = A$ $fmdom vPs = A$ $fmdom vVs = A$
and $\forall x. x \mid \in \mid A \longrightarrow (\exists \tau' v vP h.$
 $\Gamma \ \$\$ x = Some (AuthT \tau') \wedge$
 $vs \ \$\$ x = Some v \wedge$
 $vPs \ \$\$ x = Some (Hashed h vP) \wedge$
 $vVs \ \$\$ x = Some (Hash h) \wedge$
 $\{\$\$\} \vdash v, Hashed h vP, Hash h : (AuthT \tau'))$
shows $fmdrop_fset A \Gamma \vdash psubst_term e vs, psubst_term e vPs, psubst_term e vVs : \tau$
 $\langle proof \rangle$

lemmas $lemma3 = lemma3_general[\text{where } A = fmdom \Gamma \text{ and } \Gamma = \Gamma, \text{simplified}] \text{ for } \Gamma$

5.6 Lemma 4

lemma $lemma4$:
assumes $\Gamma(x \ \$\$:= \tau') \vdash e, eP, eV : \tau$
and $\{\$\$\} \vdash v, vP, vV : \tau'$
and $value v \ value vP \ value vV$
shows $\Gamma \vdash e[v / x], eP[vP / x], eV[vV / x] : \tau$
 $\langle proof \rangle$

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 $\{\$\$\} \vdash e', eP', eV' : \tau \ \forall \pi_P. \ll \pi_P, eP \gg P \rightarrow \ll \pi_P @ \pi, eP' \gg \forall \pi'. \ll \pi @ \pi', eV \gg V \rightarrow$
 $\ll \pi', eV' \gg$
 $\langle proof \rangle$

5.8 Lemma 6: Single-Step Security

lemma $lemma6$:
assumes $\{\$\$\} \vdash e, eP, eV : \tau$
and $\ll \pi_A, eV \gg V \rightarrow \ll \pi', eV' \gg$
obtains $e' eP' \pi$
where $\ll [], e \gg I \rightarrow \ll [], e' \gg \forall \pi_P. \ll \pi_P, eP \gg P \rightarrow \ll \pi_P @ \pi, eP' \gg$
and $\{\$\$\} \vdash e', eP', eV' : \tau \wedge \pi_A = \pi @ \pi' \vee$
 $(\exists s s'. closed s \wedge closed s' \wedge \pi = [s] \wedge \pi_A = [s'] @ \pi' \wedge s \neq s' \wedge hash s = hash s')$
 $\langle proof \rangle$

5.9 Theorem 1: Correctness

lemma *theorem1_correctness*:
assumes $\{\$\$ \} \vdash e, eP, eV : \tau$
and $\ll \square, e \gg I \rightarrow i \ll \square, e' \gg$
obtains $eP' eV' \pi$
where $\ll \square, eP \gg P \rightarrow i \ll \pi, eP' \gg$
 $\ll \pi, eV \gg V \rightarrow i \ll \square, eV' \gg$
 $\{\$\$ \} \vdash e', eP', eV' : \tau$
<proof>

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
 $\exists e' eP' \pi j \pi_0 s s'. (\ll \square, e \gg I \rightarrow i \ll \square, e' \gg \wedge \ll \square, eP \gg P \rightarrow i \ll \pi, eP' \gg \wedge (\pi A = \pi @ \pi'))$
 $\wedge \{\$\$ \} \vdash e', eP', eV' : \tau) \vee$
 $(j \leq i \wedge \ll \square, eP \gg P \rightarrow j \ll \pi_0 @ [s], eP' \gg \wedge (\pi A = \pi_0 @ [s'] @ \pi') \wedge s \neq s' \wedge \text{hash } s = \text{hash } s')$
and *collision*: $\text{hash } (\text{Inj1 } \text{Unit}) = \text{hash } (\text{Inj2 } \text{Unit})$
and *no_collision_with_Unit*: $\bigwedge t. \text{hash } \text{Unit} = \text{hash } t \implies t = \text{Unit}$
shows *False*
<proof>

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
 $\exists e' eP' \pi j \pi_0 s s'. (\ll \square, e \gg I \rightarrow i \ll \square, e' \gg \wedge \ll \square, eP \gg P \rightarrow i \ll \pi, eP' \gg \wedge (\pi A = \pi @ \pi'))$
 $\wedge \{\$\$ \} \vdash e', eP', eV' : \tau) \vee$
 $(j \leq i \wedge \ll \square, eP \gg P \rightarrow j \ll \pi_0 @ [s], eP' \gg \wedge (\pi A = \pi_0 @ [s'] @ \pi') \wedge s \neq s' \wedge \text{hash } s = \text{hash } s')$
and *collision*: $\text{hash } (\text{Inj1 } \text{Unit}) = \text{hash } (\text{Inj2 } \text{Unit})$
and *no_collision_with_Unit*: $\bigwedge t. \text{hash } \text{Unit} = \text{hash } t \implies t = \text{Unit}$
shows *False*
<proof>

5.11 Corrected Theorem 1: Security

lemma *theorem1_security*:
assumes $\{\$\$ \} \vdash e, eP, eV : \tau$
and $\ll \pi A, eV \gg V \rightarrow i \ll \pi', eV' \gg$
shows $(\exists e' eP' \pi. \ll \square, e \gg I \rightarrow i \ll \square, e' \gg \wedge \ll \square, eP \gg P \rightarrow i \ll \pi, eP' \gg \wedge \pi A = \pi @ \pi' \wedge \{\$\$ \} \vdash e', eP', eV' : \tau) \vee$
 $(\exists eP' j \pi_0 \pi_0' s s'. j \leq i \wedge \ll \square, eP \gg P \rightarrow j \ll \pi_0 @ [s], eP' \gg \wedge \pi A = \pi_0 @ [s'] @ \pi_0' @ \pi' \wedge s \neq s' \wedge \text{hash } s = \text{hash } s' \wedge \text{closed } s \wedge \text{closed } s')$
<proof>

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** $\{\$\$ \} \vdash e', eP', eV' : \tau \wedge \ll \square, e \gg I \rightarrow \ll \square, e' \gg \wedge \ll \pi, eV \gg V \rightarrow \ll \square, eV' \gg$
<proof>

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 $\{\$\$ \} \vdash e', eP', eV' : \tau \ll [], e \gg I \rightarrow i \ll [], e' \gg \ll \pi, eV \gg V \rightarrow i \ll [], eV' \gg$

<proof>

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.