Formalization of Generic Authenticated Data Structures

Matthias Brun Dmitriy Traytel

December 14, 2021

Abstract

Authenticated data structures are a technique for outsourcing data storage and maintenance to an untrusted server. The server is required to produce an efficiently checkable and cryptographically secure proof that it carried out precisely the requested computation. Miller et al. [2] introduced λ• (pronounced *lambda auth*)—a functional programming language with a built-in primitive authentication construct, which supports a wide range of user-specified authenticated data structures while guaranteeing certain correctness and security properties for all well-typed programs. We formalize λ• 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

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*]: $\{\} \sharp* t$
⟨*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 $atom\ b \sharp t$
shows $[[atom\ a]]lst.\ t = [[atom\ b]]lst.\ (a \leftrightarrow b) \cdot t$
⟨*proof*⟩

lemma *atom_not_fresh_eq*:
assumes $\neg\ atom\ a \sharp x$
shows $a = x$
⟨*proof*⟩

lemma *fresh_set_fresh_forall*:
shows $atom\ y \sharp xs = (\forall x \in set\ xs.\ atom\ y \sharp x)$
⟨*proof*⟩

lemma *finite_fresh_set_fresh_all*[*simp*]:
fixes $S :: (\_ :: fs)\ set$
shows $finite\ S \Longrightarrow atom\ a \sharp S \longleftrightarrow (\forall x \in S.\ atom\ a \sharp x)$
⟨*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* (*_'(_ $$:= _') [1000,0,0] 1000) **where** *fmap_update* Γ *x* τ ≡ *fmupd x* τ Γ
notation *fmlookup* (**infixl** $$ *999*)
notation *fmempty* ({$$})

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

unbundle *fmap.lifting*

lift_definition
permute_fmap :: *perm* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap*
is
permute :: *perm* ⇒ ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*b*)
⟨*proof*⟩

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 $a \sharp \{\$\$\}$
⟨*proof*⟩

lemma *fresh_fmap_update*:
shows $[\![a \sharp f;\ a \sharp x;\ a \sharp y]\!] \Longrightarrow a \sharp f(x\ \$\$:=\ y)$
⟨*proof*⟩

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

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

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

lemma *fresh_transfer*[*transfer_rule*]:
$((=) ===> pcr\_fmap\ (=)\ (=) ===> (=))\ fresh\ fresh$
⟨*proof*⟩

lemma *fmmap_eqvt*[*eqvt*]: $p \cdot (fmmap\ f\ F) = fmmap\ (p \cdot f)\ (p \cdot F)$
⟨*proof*⟩

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

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

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

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 *fresh_fmap_fresh_Some*:
atom x ♯ F ⟹ x ≠ y ⟹ F $$ y = Some a ⟹ atom x ♯ a
including *fmap.lifting*
⟨*proof*⟩

lemma *fmdrop_eqvt*: *p ∙ fmdrop x F = fmdrop (p ∙ x) (p ∙ F)*
⟨*proof*⟩

lemma *fmfilter_eqvt*: *p ∙ fmfilter Q F = fmfilter (p ∙ Q) (p ∙ F)*
⟨*proof*⟩

lemma *fmdrop_eq_iff*:
fmdrop x B = fmdrop y B ⟷ x = y ∨ (x ∉ fmdom′ B ∧ y ∉ fmdom′ 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_fmdrop_in_fmdom*: *⟦ x ∈ fmdom′ B; y ♯ B; y ♯ x ⟧ ⟹ y ♯ fmdrop x B*
⟨*proof*⟩

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*⟩

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 |
  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 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* v_1; *value* v_2 ⟧ ⟹ *value* (*Pair* v_1 v_2) |
  value v ⟹ *value* (*Roll v*) |
  value (*Hash* _) |
  value v ⟹ *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 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_real* (*map_fixed undefined* [] *t*)

lemma *permute_map_list*: *p* ∙ *l* = *map* (λ*x*. *p* ∙ *x*) *l*
⟨*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_idle_closed*:
closed t ⟹ *map_fixed undefined* [] *t* = *t*
⟨*proof*⟩

lemma *map_fixed_inj_closed*:
closed t ⟹ *closed u* ⟹ *map_fixed undefined* [] *t* = *map_fixed undefined* [] *u* ⟹ *t* = *u*
⟨*proof*⟩

lemma *hash_eq_hash_real_closed*:
assumes *closed t*
shows *hash t* = *hash_real t*
⟨*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 x* = *y 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*[*t′* / *x*]) |
(*Inj2 t*)[*t′* / *x*] = *Inj2* (*t*[*t′* / *x*]) |
(*Pair t1 t2*)[*t′* / *x*] = *Pair* (*t1*[*t′* / *x*]) (*t2*[*t′* / *x*]) |
(*Roll t*)[*t′* / *x*] = *Roll* (*t*[*t′* / *x*]) |
atom y ♯ (*x*, *t′*) ⟹ (*Let t1 y t2*)[*t′* / *x*] = *Let* (*t1*[*t′* / *x*]) *y* (*t2*[*t′* / *x*]) |
(*App t1 t2*)[*t′* / *x*] = *App* (*t1*[*t′* / *x*]) (*t2*[*t′* / *x*]) |
(*Case t1 t2 t3*)[*t′* / *x*] = *Case* (*t1*[*t′* / *x*]) (*t2*[*t′* / *x*]) (*t3*[*t′* / *x*]) |
(*Prj1 t*)[*t′* / *x*] = *Prj1* (*t*[*t′* / *x*]) |
(*Prj2 t*)[*t′* / *x*] = *Prj2* (*t*[*t′* / *x*]) |
(*Unroll t*)[*t′* / *x*] = *Unroll* (*t*[*t′* / *x*]) |
(*Auth t*)[*t′* / *x*] = *Auth* (*t*[*t′* / *x*]) |
(*Unauth t*)[*t′* / *x*] = *Unauth* (*t*[*t′* / *x*]) |
(*Hash h*)[*t′* / *x*] = *Hash h* |
(*Hashed h t*)[*t′* / *x*] = *Hashed h* (*t*[*t′* / *x*])
⟨*proof*⟩
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*]
⟨*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′* ⟷ ((*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* (_ ⊢$_W$ _ : _ [*150*,*0*,*150*] *149*) **where**
jw_Unit: Γ ⊢$_W$ *Unit* : *One* |
jw_Var: ⟦ Γ $$ *x* = *Some* τ ⟧
⟹ Γ ⊢$_W$ *Var x* : τ |
jw_Lam: ⟦ *atom x* ♯ Γ; Γ(*x* $$:= τ$_1$) ⊢$_W$ *e* : τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Lam x e* : *Fun* τ$_1$ τ$_2$ |
jw_App: ⟦ Γ ⊢$_W$ *e* : *Fun* τ$_1$ τ$_2$; Γ ⊢$_W$ *e′* : τ$_1$ ⟧
⟹ Γ ⊢$_W$ *App e e′* : τ$_2$ |
jw_Let: ⟦ *atom x* ♯ (Γ, *e*$_1$); Γ ⊢$_W$ *e*$_1$: τ$_1$; Γ(*x* $$:= τ$_1$) ⊢$_W$ *e*$_2$: τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Let e*$_1$ *x e*$_2$: τ$_2$ |
jw_Rec: ⟦ *atom x* ♯ Γ; *atom y* ♯ (Γ,*x*); Γ(*x* $$:= *Fun* τ$_1$ τ$_2$) ⊢$_W$ *Lam y e* : *Fun* τ$_1$ τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Rec x* (*Lam y e*) : *Fun* τ$_1$ τ$_2$ |
jw_Inj1: ⟦ Γ ⊢$_W$ *e* : τ$_1$ ⟧
⟹ Γ ⊢$_W$ *Inj1 e* : *Sum* τ$_1$ τ$_2$ |
jw_Inj2: ⟦ Γ ⊢$_W$ *e* : τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Inj2 e* : *Sum* τ$_1$ τ$_2$ |
jw_Case: ⟦ Γ ⊢$_W$ *e* : *Sum* τ$_1$ τ$_2$; Γ ⊢$_W$ *e*$_1$: *Fun* τ$_1$ τ; Γ ⊢$_W$ *e*$_2$: *Fun* τ$_2$ τ ⟧
⟹ Γ ⊢$_W$ *Case e e*$_1$ *e*$_2$: τ |
jw_Pair: ⟦ Γ ⊢$_W$ *e*$_1$: τ$_1$; Γ ⊢$_W$ *e*$_2$: τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Pair e*$_1$ *e*$_2$: *Prod* τ$_1$ τ$_2$ |
jw_Prj1: ⟦ Γ ⊢$_W$ *e* : *Prod* τ$_1$ τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Prj1 e* : τ$_1$ |
jw_Prj2: ⟦ Γ ⊢$_W$ *e* : *Prod* τ$_1$ τ$_2$ ⟧
⟹ Γ ⊢$_W$ *Prj2 e* : τ$_2$ |
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*: α
| *jw_Unroll*: α

⟨*proof*⟩

Inversion rules for typing judgment.

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

lemma *jw_Lam_inv*[*elim*]:
assumes $\Gamma \vdash_W \mathit{Lam}\ x\ e : \tau$
and $\mathit{atom}\ x \sharp \Gamma$
obtains $\tau_1\ \tau_2$ **where** $\tau = \mathit{Fun}\ \tau_1\ \tau_2\ (\Gamma(x\ \$\$:= \tau_1)) \vdash_W e : \tau_2$
⟨*proof*⟩

lemma *swap_permute_swap*: $\mathit{atom}\ x \sharp \pi \Longrightarrow \mathit{atom}\ y \sharp \pi \Longrightarrow (x \leftrightarrow y) \cdot \pi \cdot (x \leftrightarrow y) \cdot t = \pi \cdot t$
⟨*proof*⟩

lemma *jw_Rec_inv*[*elim*]:
assumes $\Gamma \vdash_W \mathit{Rec}\ x\ t : \tau$
and $\mathit{atom}\ x \sharp \Gamma$
obtains $y\ e\ \tau_1\ \tau_2$ **where** $\mathit{atom}\ y \sharp (\Gamma,x)\ t = \mathit{Lam}\ y\ e\ \tau = \mathit{Fun}\ \tau_1\ \tau_2\ \Gamma(x\ \$\$:= \mathit{Fun}\ \tau_1\ \tau_2) \vdash_W \mathit{Lam}\ y\ e : \mathit{Fun}\ \tau_1\ \tau_2$
⟨*proof*⟩

inductive_cases *jw_Inj1_inv*[*elim*]: $\Gamma \vdash_W \mathit{Inj1}\ e : \tau$
inductive_cases *jw_Inj2_inv*[*elim*]: $\Gamma \vdash_W \mathit{Inj2}\ e : \tau$
inductive_cases *jw_Pair_inv*[*elim*]: $\Gamma \vdash_W \mathit{Pair}\ e_1\ e_2 : \tau$

lemma *jw_Let_inv*[*elim*]:
assumes $\Gamma \vdash_W \mathit{Let}\ e_1\ x\ e_2 : \tau_2$
and $\mathit{atom}\ x \sharp (e_1, \Gamma)$
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*]: $\Gamma \vdash_W \mathit{Prj1}\ e : \tau_1$
inductive_cases *jw_Prj2_inv*[*elim*]: $\Gamma \vdash_W \mathit{Prj2}\ e : \tau_2$
inductive_cases *jw_App_inv*[*elim*]: $\Gamma \vdash_W \mathit{App}\ e\ e' : \tau_2$
inductive_cases *jw_Case_inv*[*elim*]: $\Gamma \vdash_W \mathit{Case}\ e\ e_1\ e_2 : \tau$
inductive_cases *jw_Auth_inv*[*elim*]: $\Gamma \vdash_W \mathit{Auth}\ e : \tau$
inductive_cases *jw_Unauth_inv*[*elim*]: $\Gamma \vdash_W \mathit{Unauth}\ e : \tau$

lemma *subst_type_perm_eq*:
assumes $\mathit{atom}\ b \sharp t$
shows $\mathit{subst\_type}\ t\ (\mathit{Mu}\ a\ t)\ a = \mathit{subst\_type}\ ((a \leftrightarrow b) \cdot t)\ (\mathit{Mu}\ b\ ((a \leftrightarrow b) \cdot t))\ b$
⟨*proof*⟩

lemma *jw_Roll_inv*[*elim*]:
assumes $\Gamma \vdash_W \mathit{Roll}\ e : \tau$
and $\mathit{atom}\ \alpha \sharp (\Gamma, \tau)$
obtains τ' **where** $\tau = \mathit{Mu}\ \alpha\ \tau'\ \Gamma \vdash_W e : \mathit{subst\_type}\ \tau'\ (\mathit{Mu}\ \alpha\ \tau')\ \alpha$
⟨*proof*⟩

lemma *jw_Unroll_inv*[*elim*]:
assumes $\Gamma \vdash_W \mathit{Unroll}\ e : \tau$
and $\mathit{atom}\ \alpha \sharp (\Gamma, \tau)$
obtains τ' **where** $\tau = \mathit{subst\_type}\ \tau'\ (\mathit{Mu}\ \alpha\ \tau')\ \alpha\ \Gamma \vdash_W e : \mathit{Mu}\ \alpha\ \tau'$
⟨*proof*⟩

Additional inversion rules based on type rather than term.

inductive_cases *jw_Prod_inv*[*elim*]: $\{\$\$\} \vdash_W e : \mathit{Prod}\ \tau_1\ \tau_2$

inductive_cases *jw_Sum_inv*[*elim*]: $\{\$\$\} \vdash_W e : Sum\ \tau_1\ \tau_2$

lemma *jw_Fun_inv*[*elim*]:
assumes $\{\$\$\} \vdash_W v : Fun\ \tau_1\ \tau_2\ value\ v$
obtains $e\ x$ **where** $v = Lam\ x\ e \vee v = Rec\ x\ e\ atom\ x \sharp (c::term)$
⟨*proof*⟩

lemma *jw_Mu_inv*[*elim*]:
assumes $\{\$\$\} \vdash_W v : Mu\ \alpha\ \tau\ value\ v$
obtains v' **where** $v = Roll\ v'$
⟨*proof*⟩

3.4 Erasure of Authenticated Types

nominal_function *erase* :: *ty* ⇒ *ty* **where**
erase One = *One* |
erase (*Fun* $\tau_1\ \tau_2$) = *Fun* (*erase* τ_1) (*erase* τ_2) |
erase (*Sum* $\tau_1\ \tau_2$) = *Sum* (*erase* τ_1) (*erase* τ_2) |
erase (*Prod* $\tau_1\ \tau_2$) = *Prod* (*erase* τ_1) (*erase* τ_2) |
erase (*Mu* $\alpha\ \tau$) = *Mu* α (*erase* τ) |
erase (*Alpha* α) = *Alpha* α |
erase (*AuthT* τ) = *erase* τ
⟨*proof*⟩
nominal_termination (*eqvt*)
⟨*proof*⟩

lemma *fresh_erase_fresh*:
assumes *atom* $x \sharp \tau$
shows *atom* $x \sharp$ *erase* τ
⟨*proof*⟩

lemma *fresh_fmmap_erase_fresh*:
assumes *atom* $x \sharp \Gamma$
shows *atom* $x \sharp$ *fmmap erase* Γ
⟨*proof*⟩

lemma *erase_subst_type_shift*[*simp*]:
erase (*subst_type* $\tau\ \tau'\ \alpha$) = *subst_type* (*erase* τ) (*erase* τ') α
⟨*proof*⟩

definition *erase_env* :: *tyenv* ⇒ *tyenv* **where**
erase_env = *fmmap erase*

3.5 Strong Typing Judgement

inductive *judge* :: *tyenv* ⇒ *term* ⇒ *ty* ⇒ *bool* (_ ⊢ _ : _ [*150,0,150*] *149*) **where**
j_Unit: $\Gamma \vdash Unit : One$ |
j_Var: $[\![\Gamma\ \$\$\ x = Some\ \tau]\!]$
$\Longrightarrow \Gamma \vdash Var\ x : \tau$ |
j_Lam: $[\![atom\ x \sharp \Gamma; \Gamma(x\ \$\$:= \tau_1) \vdash e : \tau_2]\!]$
$\Longrightarrow \Gamma \vdash Lam\ x\ e : Fun\ \tau_1\ \tau_2$ |
j_App: $[\![\Gamma \vdash e : Fun\ \tau_1\ \tau_2; \Gamma \vdash e' : \tau_1]\!]$
$\Longrightarrow \Gamma \vdash App\ e\ e' : \tau_2$ |
j_Let: $[\![atom\ x \sharp (\Gamma, e_1); \Gamma \vdash e_1 : \tau_1; \Gamma(x\ \$\$:= \tau_1) \vdash e_2 : \tau_2]\!]$
$\Longrightarrow \Gamma \vdash Let\ e_1\ x\ e_2 : \tau_2$ |
j_Rec: $[\![atom\ x \sharp \Gamma; atom\ y \sharp (\Gamma,x); \Gamma(x\ \$\$:= Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e' : Fun\ \tau_1\ \tau_2]\!]$
$\Longrightarrow \Gamma \vdash Rec\ x\ (Lam\ y\ e') : Fun\ \tau_1\ \tau_2$ |
j_Inj1: $[\![\Gamma \vdash e : \tau_1]\!]$
$\Longrightarrow \Gamma \vdash Inj1\ e : Sum\ \tau_1\ \tau_2$ |

j_Inj2: $[\![\Gamma \vdash e : \tau_2]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Inj2}\ e : \mathit{Sum}\ \tau_1\ \tau_2$ |
j_Case: $[\![\Gamma \vdash e : \mathit{Sum}\ \tau_1\ \tau_2; \Gamma \vdash e_1 : \mathit{Fun}\ \tau_1\ \tau; \Gamma \vdash e_2 : \mathit{Fun}\ \tau_2\ \tau]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Case}\ e\ e_1\ e_2 : \tau$ |
j_Pair: $[\![\Gamma \vdash e_1 : \tau_1; \Gamma \vdash e_2 : \tau_2]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Pair}\ e_1\ e_2 : \mathit{Prod}\ \tau_1\ \tau_2$ |
j_Prj1: $[\![\Gamma \vdash e : \mathit{Prod}\ \tau_1\ \tau_2]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Prj1}\ e : \tau_1$ |
j_Prj2: $[\![\Gamma \vdash e : \mathit{Prod}\ \tau_1\ \tau_2]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Prj2}\ e : \tau_2$ |
j_Roll: $[\![\mathit{atom}\ \alpha \sharp \Gamma; \Gamma \vdash e : \mathit{subst\_type}\ \tau\ (\mathit{Mu}\ \alpha\ \tau)\ \alpha]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Roll}\ e : \mathit{Mu}\ \alpha\ \tau$ |
j_Unroll: $[\![\mathit{atom}\ \alpha \sharp \Gamma; \Gamma \vdash e : \mathit{Mu}\ \alpha\ \tau]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Unroll}\ e : \mathit{subst\_type}\ \tau\ (\mathit{Mu}\ \alpha\ \tau)\ \alpha$ |
j_Auth: $[\![\Gamma \vdash e : \tau]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Auth}\ e : \mathit{AuthT}\ \tau$ |
j_Unauth: $[\![\Gamma \vdash e : \mathit{AuthT}\ \tau]\!]$
$\Longrightarrow \Gamma \vdash \mathit{Unauth}\ e : \tau$

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 $\Gamma \vdash e : \tau$
shows $\mathit{erase\_env}\ \Gamma \vdash_W e : \mathit{erase}\ \tau$
⟨*proof*⟩

3.6 Shallow Projection

nominal_function *shallow* :: *term* ⇒ *term* ((⦇_⦈)) **where**
$(\!|\mathit{Unit}|\!) = \mathit{Unit}$ |
$(\!|\mathit{Var}\ v|\!) = \mathit{Var}\ v$ |
$(\!|\mathit{Lam}\ x\ e|\!) = \mathit{Lam}\ x\ (\!|e|\!)$ |
$(\!|\mathit{Rec}\ x\ e|\!) = \mathit{Rec}\ x\ (\!|e|\!)$ |
$(\!|\mathit{Inj1}\ e|\!) = \mathit{Inj1}\ (\!|e|\!)$ |
$(\!|\mathit{Inj2}\ e|\!) = \mathit{Inj2}\ (\!|e|\!)$ |
$(\!|\mathit{Pair}\ e_1\ e_2|\!) = \mathit{Pair}\ (\!|e_1|\!)\ (\!|e_2|\!)$ |
$(\!|\mathit{Roll}\ e|\!) = \mathit{Roll}\ (\!|e|\!)$ |
$(\!|\mathit{Let}\ e_1\ x\ e_2|\!) = \mathit{Let}\ (\!|e_1|\!)\ x\ (\!|e_2|\!)$ |
$(\!|\mathit{App}\ e_1\ e_2|\!) = \mathit{App}\ (\!|e_1|\!)\ (\!|e_2|\!)$ |
$(\!|\mathit{Case}\ e\ e_1\ e_2|\!) = \mathit{Case}\ (\!|e|\!)\ (\!|e_1|\!)\ (\!|e_2|\!)$ |
$(\!|\mathit{Prj1}\ e|\!) = \mathit{Prj1}\ (\!|e|\!)$ |
$(\!|\mathit{Prj2}\ e|\!) = \mathit{Prj2}\ (\!|e|\!)$ |
$(\!|\mathit{Unroll}\ e|\!) = \mathit{Unroll}\ (\!|e|\!)$ |
$(\!|\mathit{Auth}\ e|\!) = \mathit{Auth}\ (\!|e|\!)$ |
$(\!|\mathit{Unauth}\ e|\!) = \mathit{Unauth}\ (\!|e|\!)$ |
— No rule is defined for Hash, but: "[..] preserving that structure in every case but that of <h, v> [..]"
$(\!|\mathit{Hash}\ h|\!) = \mathit{Hash}\ h$ |
$(\!|\mathit{Hashed}\ h\ e|\!) = \mathit{Hash}\ h$
⟨*proof*⟩

nominal_termination (*eqvt*)
⟨*proof*⟩

lemma *fresh_shallow*: *atom x* ♯ *e* ⟹ *atom x* ♯ ⦇*e*⦈
⟨*proof*⟩

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, e_1 ≫ *m*→ ≪ π′, e_1′ ≫ ⟧
⟹ ≪ π, *App* e_1 e_2 ≫ *m*→ ≪ π′, *App* e_1′ e_2 ≫ |
s_App2: ⟦ *value* v_1; ≪ π, e_2 ≫ *m*→ ≪ π′, e_2′ ≫ ⟧
⟹ ≪ π, *App* v_1 e_2 ≫ *m*→ ≪ π′, *App* v_1 e_2′ ≫ |
s_AppLam: ⟦ *value v*; *atom x* ♯ (*v*,π) ⟧
⟹ ≪ π, *App* (*Lam x e*) *v* ≫ _→ ≪ π, *e*[*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* ♯ (e_1,e_1′,π,π′); ≪ π, e_1 ≫ *m*→ ≪ π′, e_1′ ≫ ⟧
⟹ ≪ π, *Let* e_1 *x* e_2 ≫ *m*→ ≪ π′, *Let* e_1′ *x* e_2 ≫ |
s_Let2: ⟦ *value v*; *atom x* ♯ (*v*,π) ⟧
⟹ ≪ π, *Let v x e* ≫ _→ ≪ π, *e*[*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 e* e_1 e_2 ≫ *m*→ ≪ π′, *Case e*′ e_1 e_2 ≫ |
— Case rules are different from paper to account for recursive functions.
s_CaseInj1: ⟦ *value v* ⟧
⟹ ≪ π, *Case* (*Inj1 v*) e_1 e_2 ≫ _→ ≪ π, *App* e_1 *v* ≫ |
s_CaseInj2: ⟦ *value v* ⟧
⟹ ≪ π, *Case* (*Inj2 v*) e_1 e_2 ≫ _→ ≪ π, *App* e_2 *v* ≫ |
s_Pair1: ⟦ ≪ π, e_1 ≫ *m*→ ≪ π′, e_1′ ≫ ⟧
⟹ ≪ π, *Pair* e_1 e_2 ≫ *m*→ ≪ π′, *Pair* e_1′ e_2 ≫ |
s_Pair2: ⟦ *value* v_1; ≪ π, e_2 ≫ *m*→ ≪ π′, e_2′ ≫ ⟧
⟹ ≪ π, *Pair* v_1 e_2 ≫ *m*→ ≪ π′, *Pair* v_1 e_2′ ≫ |
s_Prj1: ⟦ ≪ π, *e* ≫ *m*→ ≪ π′, *e*′ ≫ ⟧
⟹ ≪ π, *Prj1 e* ≫ *m*→ ≪ π′, *Prj1 e*′ ≫ |
s_Prj2: ⟦ ≪ π, *e* ≫ *m*→ ≪ π′, *e*′ ≫ ⟧
⟹ ≪ π, *Prj2 e* ≫ *m*→ ≪ π′, *Prj2 e*′ ≫ |
s_PrjPair1: ⟦ *value* v_1; *value* v_2 ⟧
⟹ ≪ π, *Prj1* (*Pair* v_1 v_2) ≫ _→ ≪ π, v_1 ≫ |
s_PrjPair2: ⟦ *value* v_1; *value* v_2 ⟧
⟹ ≪ π, *Prj2* (*Pair* v_1 v_2) ≫ _→ ≪ π, v_2 ≫ |
s_Unroll: ≪ π, *e* ≫ *m*→ ≪ π′, *e*′ ≫
⟹ ≪ π, *Unroll e* ≫ *m*→ ≪ π′, *Unroll e*′ ≫ |

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

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

equivariance *smallstep*
nominal_inductive *smallstep*
avoids *s_AppLam*: x
| *s_AppRec*: x
| *s_Let1*: x
| *s_Let2*: x
⟨*proof*⟩

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

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* $\longleftrightarrow$ *?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* $x' \sharp (x,\ e)$
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 $atom\ x \sharp (e_1,\pi)$
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 $atom\ x \sharp (e_1,\pi)$
and $e' = e_2[e_1\ /\ x] \wedge value\ e_1 \wedge \pi = \pi' \Longrightarrow Q$
and $\bigwedge e_1'.\ \ll \pi,\ e_1 \gg\ m\rightarrow\ \ll \pi',\ e_1' \gg \wedge\ e' = Let\ e_1'\ x\ e_2 \wedge \neg\ value\ e_1 \Longrightarrow Q$
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 $\ll \pi,\ e \gg I\rightarrow \ll \pi',\ e' \gg atom\ x \sharp e$
shows $atom\ x \sharp e'$

⟨*proof*⟩

lemma *fresh_smallstep_P*:
fixes x :: *var*
assumes $\ll \pi,\ e \gg P{\rightarrow} \ll \pi',\ e' \gg$ *atom* $x \sharp e$
shows *atom* $x \sharp e'$
⟨*proof*⟩

lemma *fresh_smallsteps_I*:
fixes x :: *var*
assumes $\ll \pi,\ e \gg I{\rightarrow}i \ll \pi',\ e' \gg$ *atom* $x \sharp e$
shows *atom* $x \sharp e'$
⟨*proof*⟩

lemma *fresh_ps_smallstep_P*:
fixes x :: *var*
assumes $\ll \pi,\ e \gg P{\rightarrow} \ll \pi',\ e' \gg$ *atom* $x \sharp e$ *atom* $x \sharp \pi$
shows *atom* $x \sharp \pi'$
⟨*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*:
$\ll\pi,\ e\gg I{\rightarrow} \ll[],\ e'\gg \Longrightarrow \ll[],\ e\gg I{\rightarrow} \ll[],\ e'\gg$
$\ll[],\ e\gg I{\rightarrow} \ll\pi,\ e'\gg \Longrightarrow \ll[],\ e\gg I{\rightarrow} \ll[],\ e'\gg$
⟨*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*:
$\ll\pi,\ e\gg I{\rightarrow}i \ll[],\ e'\gg \Longrightarrow \ll[],\ e\gg I{\rightarrow}i \ll[],\ e'\gg$
$\ll[],\ e\gg I{\rightarrow}i \ll\pi,\ e'\gg \Longrightarrow \ll[],\ e\gg I{\rightarrow}i \ll[],\ e'\gg$
⟨*proof*⟩

lemma *smallstepV_consumes_proofstream*:
assumes $\ll \pi_1,\ eV \gg V{\rightarrow} \ll \pi_2,\ eV' \gg$
obtains π **where** $\pi_1 = \pi \mathbin{@} \pi_2$
⟨*proof*⟩

lemma *smallstepsV_consumes_proofstream*:
assumes $\ll \pi_1,\ eV \gg V{\rightarrow}i \ll \pi_2,\ eV' \gg$
obtains π **where** $\pi_1 = \pi \mathbin{@} \pi_2$
⟨*proof*⟩

lemma *smallstepP_generates_proofstream*:
assumes $\ll \pi_1,\ eP \gg P{\rightarrow} \ll \pi_2,\ eP' \gg$
obtains π **where** $\pi_2 = \pi_1 \mathbin{@} \pi$
⟨*proof*⟩

lemma *smallstepsP_generates_proofstream*:
assumes $\ll \pi_1,\ eP \gg P{\rightarrow}i \ll \pi_2,\ eP' \gg$

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

lemma *smallstepV_ps_append*:
$\ll \pi,\ eV \gg V\!\!\rightarrow \ll \pi',\ eV' \gg \longleftrightarrow \ll \pi \mathbin{@} X,\ eV \gg V\!\!\rightarrow \ll \pi' \mathbin{@} X,\ eV' \gg$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

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

lemma *smallstepsV_ps_append*:
$\ll \pi,\ eV \gg V\!\!\rightarrow\!i \ll \pi',\ eV' \gg \longleftrightarrow \ll \pi \mathbin{@} X,\ eV \gg V\!\!\rightarrow\!i \ll \pi' \mathbin{@} X,\ eV' \gg$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *smallstepP_ps_prepend*:
$\ll \pi,\ eP \gg P\!\!\rightarrow \ll \pi',\ eP' \gg \longleftrightarrow \ll X \mathbin{@} \pi,\ eP \gg P\!\!\rightarrow \ll X \mathbin{@} \pi',\ eP' \gg$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *smallstepsP_ps_prepend*:
$\ll \pi,\ eP \gg P\!\!\rightarrow\!i \ll \pi',\ eP' \gg \longleftrightarrow \ll X \mathbin{@} \pi,\ eP \gg P\!\!\rightarrow\!i \ll X \mathbin{@} \pi',\ eP' \gg$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

3.8 Type Progress

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

3.9 Weak Type Preservation

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

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

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

lemma *judge_weak_weakening_2*:
assumes $\Gamma \vdash_W e : \tau$ *atom* $y \sharp \Gamma$
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 Γ(*a* $$:= *τ′*) ⊢$_W$ *e* : *τ* {$$} ⊢$_W$ *e′* : *τ′*
shows Γ ⊢$_W$ *e*[*e′* / *a*] : *τ*
⟨*proof*⟩

lemma *type_preservation*:
assumes ≪ [], *e* ≫ *I*→ ≪ [], *e′* ≫ {$$} ⊢$_W$ *e* : *τ*
shows {$$} ⊢$_W$ *e′* : *τ*
⟨*proof*⟩

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

lemma *type_soundness*:
assumes {$$} ⊢$_W$ *e* : *τ*
shows *value e* ∨ (∃ *e′*. ≪ [], *e* ≫ *I*→ ≪ [], *e′* ≫ ∧ {$$} ⊢$_W$ *e′* : *τ*)
⟨*proof*⟩

4 Agreement Relation

inductive *agree* :: *tyenv* ⇒ *term* ⇒ *term* ⇒ *term* ⇒ *ty* ⇒ *bool* (_ ⊢ _, _, _ : _ [*150*,*0*,*0*,*0*,*150*] *149*)
where
a_Unit: Γ ⊢ *Unit*, *Unit*, *Unit* : *One* |
a_Var: Γ $$ *x* = *Some τ*
⟹ Γ ⊢ *Var x*, *Var x*, *Var x* : *τ* |
a_Lam: ⟦ *atom x* ♯ Γ; Γ(*x* $$:= *τ*$_1$) ⊢ *e*, *eP*, *eV* : *τ*$_2$ ⟧
⟹ Γ ⊢ *Lam x e*, *Lam x eP*, *Lam x eV* : *Fun τ*$_1$ *τ*$_2$ |
a_App: ⟦ Γ ⊢ *e*$_1$, *eP*$_1$, *eV*$_1$: *Fun τ*$_1$ *τ*$_2$; Γ ⊢ *e*$_2$, *eP*$_2$, *eV*$_2$: *τ*$_1$ ⟧
⟹ Γ ⊢ *App e*$_1$ *e*$_2$, *App eP*$_1$ *eP*$_2$, *App eV*$_1$ *eV*$_2$: *τ*$_2$ |
a_Let: ⟦ *atom x* ♯ (Γ, *e*$_1$, *eP*$_1$, *eV*$_1$); Γ ⊢ *e*$_1$, *eP*$_1$, *eV*$_1$: *τ*$_1$; Γ(*x* $$:= *τ*$_1$) ⊢ *e*$_2$, *eP*$_2$, *eV*$_2$: *τ*$_2$ ⟧
⟹ Γ ⊢ *Let e*$_1$ *x e*$_2$, *Let eP*$_1$ *x eP*$_2$, *Let eV*$_1$ *x eV*$_2$: *τ*$_2$ |
a_Rec: ⟦ *atom x* ♯ Γ; *atom y* ♯ (Γ,*x*); Γ(*x* $$:= *Fun τ*$_1$ *τ*$_2$) ⊢ *Lam y e*, *Lam y eP*, *Lam y eV* : *Fun τ*$_1$ *τ*$_2$ ⟧
⟹ Γ ⊢ *Rec x* (*Lam y e*), *Rec x* (*Lam y eP*), *Rec x* (*Lam y eV*) : *Fun τ*$_1$ *τ*$_2$ |
a_Inj1: ⟦ Γ ⊢ *e*, *eP*, *eV* : *τ*$_1$ ⟧
⟹ Γ ⊢ *Inj1 e*, *Inj1 eP*, *Inj1 eV* : *Sum τ*$_1$ *τ*$_2$ |
a_Inj2: ⟦ Γ ⊢ *e*, *eP*, *eV* : *τ*$_2$ ⟧
⟹ Γ ⊢ *Inj2 e*, *Inj2 eP*, *Inj2 eV* : *Sum τ*$_1$ *τ*$_2$ |
a_Case: ⟦ Γ ⊢ *e*, *eP*, *eV* : *Sum τ*$_1$ *τ*$_2$; Γ ⊢ *e*$_1$, *eP*$_1$, *eV*$_1$: *Fun τ*$_1$ *τ*; Γ ⊢ *e*$_2$, *eP*$_2$, *eV*$_2$: *Fun τ*$_2$ *τ* ⟧
⟹ Γ ⊢ *Case e e*$_1$ *e*$_2$, *Case eP eP*$_1$ *eP*$_2$, *Case eV eV*$_1$ *eV*$_2$: *τ* |
a_Pair: ⟦ Γ ⊢ *e*$_1$, *eP*$_1$, *eV*$_1$: *τ*$_1$; Γ ⊢ *e*$_2$, *eP*$_2$, *eV*$_2$: *τ*$_2$ ⟧
⟹ Γ ⊢ *Pair e*$_1$ *e*$_2$, *Pair eP*$_1$ *eP*$_2$, *Pair eV*$_1$ *eV*$_2$: *Prod τ*$_1$ *τ*$_2$ |
a_Prj1: ⟦ Γ ⊢ *e*, *eP*, *eV* : *Prod τ*$_1$ *τ*$_2$ ⟧
⟹ Γ ⊢ *Prj1 e*, *Prj1 eP*, *Prj1 eV* : *τ*$_1$ |
a_Prj2: ⟦ Γ ⊢ *e*, *eP*, *eV* : *Prod τ*$_1$ *τ*$_2$ ⟧
⟹ Γ ⊢ *Prj2 e*, *Prj2 eP*, *Prj2 eV* : *τ*$_2$ |
a_Roll: ⟦ *atom α* ♯ Γ; Γ ⊢ *e*, *eP*, *eV* : *subst_type τ* (*Mu α τ*) *α* ⟧
⟹ Γ ⊢ *Roll e*, *Roll eP*, *Roll eV* : *Mu α τ* |
a_Unroll: ⟦ *atom α* ♯ Γ; Γ ⊢ *e*, *eP*, *eV* : *Mu α τ* ⟧
⟹ Γ ⊢ *Unroll e*, *Unroll eP*, *Unroll eV* : *subst_type τ* (*Mu α τ*) *α* |
a_Auth: ⟦ Γ ⊢ *e*, *eP*, *eV* : *τ* ⟧
⟹ Γ ⊢ *Auth e*, *Auth eP*, *Auth eV* : *AuthT τ* |
a_Unauth: ⟦ Γ ⊢ *e*, *eP*, *eV* : *AuthT τ* ⟧

$\Longrightarrow \Gamma \vdash \mathit{Unauth}\ e,\ \mathit{Unauth}\ eP,\ \mathit{Unauth}\ eV : \tau$ |
a_HashI: $[\![\ \{\$\$\} \vdash v,\ vP,\ (\!|vP|\!) : \tau;\ \mathit{hash}\ (\!|vP|\!) = h;\ \mathit{value}\ v;\ \mathit{value}\ vP\ ]\!]$
$\Longrightarrow \Gamma \vdash v,\ \mathit{Hashed}\ h\ vP,\ \mathit{Hash}\ h : \mathit{AuthT}\ \tau$

declare *agree.intros*[*intro*]

equivariance *agree*
nominal_inductive *agree*
avoids *a_Lam*: x
| *a_Rec*: x **and** y
| *a_Let*: x
| *a_Roll*: α
| *a_Unroll*: α
⟨*proof*⟩

lemma *Abs_lst_eq_3tuple*:
fixes $x\ x'$ :: *var*
fixes $e\ eP\ eV\ e'\ eP'\ eV'$ :: *term*
assumes $[[\mathit{atom}\ x]]\mathit{lst}.\ e = [[\mathit{atom}\ x']]\mathit{lst}.\ e'$
and $[[\mathit{atom}\ x]]\mathit{lst}.\ eP = [[\mathit{atom}\ x']]\mathit{lst}.\ eP'$
and $[[\mathit{atom}\ x]]\mathit{lst}.\ eV = [[\mathit{atom}\ x']]\mathit{lst}.\ eV'$
shows $[[\mathit{atom}\ x]]\mathit{lst}.\ (e,\ eP,\ eV) = [[\mathit{atom}\ x']]\mathit{lst}.\ (e',\ eP',\ eV')$
⟨*proof*⟩

lemma *agree_fresh_env_fresh_term*:
fixes a :: *var*
assumes $\Gamma \vdash e,\ eP,\ eV : \tau\ \mathit{atom}\ a \sharp \Gamma$
shows $\mathit{atom}\ a \sharp (e,\ eP,\ eV)$
⟨*proof*⟩

lemma *agree_empty_fresh*[*dest*]:
fixes a :: *var*
assumes $\{\$\$\} \vdash e,\ eP,\ eV : \tau$
shows $\{\mathit{atom}\ a\} \sharp* \{e,\ eP,\ eV\}$
⟨*proof*⟩

Inversion rules for agreement.

declare [[*simproc del*: *alpha_lst*]]

lemma *a_Lam_inv_I*[*elim*]:
assumes $\Gamma \vdash (\mathit{Lam}\ x\ e'),\ eP,\ eV : (\mathit{Fun}\ \tau_1\ \tau_2)$
and $\mathit{atom}\ x \sharp \Gamma$
obtains $eP'\ eV'$ **where** $eP = \mathit{Lam}\ x\ eP'\ eV = \mathit{Lam}\ x\ eV'\ \Gamma(x\ \$\$:= \tau_1) \vdash e',\ eP',\ eV' : \tau_2$
⟨*proof*⟩

lemma *a_Lam_inv_P*[*elim*]:
assumes $\{\$\$\} \vdash v,\ (\mathit{Lam}\ x\ vP'),\ vV : (\mathit{Fun}\ \tau_1\ \tau_2)$
obtains $v'\ vV'$ **where** $v = \mathit{Lam}\ x\ v'\ vV = \mathit{Lam}\ x\ vV'\ \{\$\$\}(x\ \$\$:= \tau_1) \vdash v',\ vP',\ vV' : \tau_2$
⟨*proof*⟩

lemma *a_Lam_inv_V*[*elim*]:
assumes $\{\$\$\} \vdash v,\ vP,\ (\mathit{Lam}\ x\ vV') : (\mathit{Fun}\ \tau_1\ \tau_2)$
obtains $v'\ vP'$ **where** $v = \mathit{Lam}\ x\ v'\ vP = \mathit{Lam}\ x\ vP'\ \{\$\$\}(x\ \$\$:= \tau_1) \vdash v',\ vP',\ vV' : \tau_2$
⟨*proof*⟩

lemma *a_Rec_inv_I*[*elim*]:
assumes $\Gamma \vdash \mathit{Rec}\ x\ e,\ eP,\ eV : \mathit{Fun}\ \tau_1\ \tau_2$
and $\mathit{atom}\ x \sharp \Gamma$

obtains $y\ e'\ eP'\ eV'$
where $e = Lam\ y\ e'\ eP = Rec\ x\ (Lam\ y\ eP')\ eV = Rec\ x\ (Lam\ y\ eV')\ atom\ y \sharp (\Gamma,x)$
$\Gamma(x\ \$\$:= Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e',\ Lam\ y\ eP',\ Lam\ y\ eV' : Fun\ \tau_1\ \tau_2$
⟨*proof*⟩

lemma *a_Rec_inv_P*[*elim*]:
assumes $\Gamma \vdash e,\ Rec\ x\ eP,\ eV : Fun\ \tau_1\ \tau_2$
and $atom\ x \sharp \Gamma$
obtains $y\ e'\ eP'\ eV'$
where $e = Rec\ x\ (Lam\ y\ e')\ eP = Lam\ y\ eP'\ eV = Rec\ x\ (Lam\ y\ eV')\ atom\ y \sharp (\Gamma,x)$
$\Gamma(x\ \$\$:= Fun\ \tau_1\ \tau_2) \vdash Lam\ y\ e',\ Lam\ y\ eP',\ Lam\ y\ eV' : Fun\ \tau_1\ \tau_2$
⟨*proof*⟩

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

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 $eP = Roll\ eP'\ eV = Roll\ eV'\ \Gamma \vdash e',\ eP',\ eV' : subst\_type\ \tau\ (Mu\ \alpha\ \tau)\ \alpha$
⟨*proof*⟩

lemma *a_Roll_inv_P*[*elim*]:
assumes $\Gamma \vdash e,\ Roll\ eP',\ eV : Mu\ \alpha\ \tau$
obtains $e'\ eV'$
where $e = Roll\ e'\ eV = Roll\ eV'\ \Gamma \vdash e',\ eP',\ eV' : subst\_type\ \tau\ (Mu\ \alpha\ \tau)\ \alpha$
⟨*proof*⟩

lemma *a_Roll_inv_V*[*elim*]:
assumes $\Gamma \vdash e,\ eP,\ Roll\ eV' : Mu\ \alpha\ \tau$
obtains $e'\ eP'$
where $e = Roll\ e'\ eP = Roll\ eP'\ \Gamma \vdash e',\ eP',\ eV' : subst\_type\ \tau\ (Mu\ \alpha\ \tau)\ \alpha$
⟨*proof*⟩

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** $vP = Hashed\ (hash\ (|vP'|))\ vP'\ vV = Hash\ (hash\ (|vP'|))\ value\ vP'$
⟨*proof*⟩

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 $\Gamma \vdash e, eP, eV : \tau$ *atom* $y \sharp e$ *atom* $y \sharp eP$ *atom* $y \sharp eV$
shows $\Gamma(y\ \$\$:= \tau') \vdash e, eP, eV : \tau$
⟨*proof*⟩

lemma *agree_weakening_2*:
assumes $\Gamma \vdash e, eP, eV : \tau$ *atom* $y \sharp \Gamma$
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 $\bigwedge \Gamma\ e\ eP\ eV\ \tau\ eP'\ eV'.\ [\![\ \Gamma \vdash e, eP, eV : \tau; \Gamma \vdash e, eP', eV' : \tau\ ]\!] \Longrightarrow eP = eP' \wedge eV = eV'$
shows *False*
⟨*proof*⟩

lemma *smallstep_ideal_deterministic*:
$\ll[], t\gg I{\rightarrow} \ll[], u\gg \Longrightarrow \ll[], t\gg I{\rightarrow} \ll[], u'\gg \Longrightarrow u = u'$
⟨*proof*⟩

lemma *smallsteps_ideal_deterministic*:
$\ll[], t\gg I{\rightarrow}i \ll[], u\gg \Longrightarrow \ll[], t\gg I{\rightarrow}i \ll[], u'\gg \Longrightarrow u = u'$
⟨*proof*⟩

5.3 Lemma 2.3

lemma *lemma2_3*:
assumes $\Gamma \vdash e, eP, eV : \tau$

shows *erase_env* $\Gamma \vdash_W e : \mathit{erase}\ \tau$
⟨*proof*⟩

5.4 Lemma 2.4

lemma *lemma2_4*[*dest*]:
assumes $\Gamma \vdash e, eP, eV : \tau$
shows $\mathit{value}\ e \wedge \mathit{value}\ eP \wedge \mathit{value}\ eV \vee \neg\ \mathit{value}\ e \wedge \neg\ \mathit{value}\ eP \wedge \neg\ \mathit{value}\ eV$
⟨*proof*⟩

5.5 Lemma 3

lemma *lemma3_general*:
fixes Γ :: *tyenv* **and** *vs vPs vVs* :: *tenv*
assumes $\Gamma \vdash e : \tau\ A\ |\subseteq|\ \mathit{fmdom}\ \Gamma$
and $\mathit{fmdom}\ vs = A\ \mathit{fmdom}\ vPs = A\ \mathit{fmdom}\ vVs = A$
and $\forall x.\ x\ |\in|\ A \longrightarrow (\exists \tau'\ v\ vP\ h.$
$\Gamma\ \$\$\ x = \mathit{Some}\ (\mathit{AuthT}\ \tau') \wedge$
$vs\ \$\$\ x = \mathit{Some}\ v \wedge$
$vPs\ \$\$\ x = \mathit{Some}\ (\mathit{Hashed}\ h\ vP) \wedge$
$vVs\ \$\$\ x = \mathit{Some}\ (\mathit{Hash}\ h) \wedge$
$\{\$\$\} \vdash v, \mathit{Hashed}\ h\ vP, \mathit{Hash}\ h : (\mathit{AuthT}\ \tau'))$
shows $\mathit{fmdrop\_fset}\ A\ \Gamma \vdash \mathit{psubst\_term}\ e\ vs, \mathit{psubst\_term}\ e\ vPs, \mathit{psubst\_term}\ e\ vVs : \tau$
⟨*proof*⟩

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

5.6 Lemma 4

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

5.7 Lemma 5: Single-Step Correctness

lemma *lemma5*:
assumes $\{\$\$\} \vdash e, eP, eV : \tau$
and $\ll [], e \gg I\!\!\rightarrow \ll [], e' \gg$
obtains $eP'\ eV'\ \pi$
where $\{\$\$\} \vdash e', eP', eV' : \tau\ \forall \pi_P.\ \ll \pi_P, eP \gg P\!\!\rightarrow \ll \pi_P\ @\ \pi, eP' \gg \forall \pi'.\ \ll \pi\ @\ \pi', eV \gg V\!\!\rightarrow \ll \pi', eV' \gg$
⟨*proof*⟩

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 $\{\$\$\} \vdash e', eP', eV' : \tau \wedge \pi_A = \pi\ @\ \pi' \vee$
$(\exists s\ s'.\ \mathit{closed}\ s \wedge \mathit{closed}\ s' \wedge \pi = [s] \wedge \pi_A = [s']\ @\ \pi' \wedge s \neq s' \wedge \mathit{hash}\ s = \mathit{hash}\ s')$
⟨*proof*⟩

5.9 Theorem 1: Correctness

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

5.10 Counterexamples to Theorem 1: Security

Counterexample using administrative normal form.

lemma *security_false*:
assumes *agree*: $\bigwedge e\ eP\ eV\ \tau\ \pi A\ i\ \pi'\ eV'.\ [\![\ \{\$\$\} \vdash e, eP, eV : \tau; \ll \pi A, eV \gg V{\rightarrow}i \ll \pi', eV' \gg\ ]\!] \Longrightarrow$
$\exists e'\ eP'\ \pi\ j\ \pi 0\ s\ s'.\ (\ll [], e \gg I{\rightarrow}i \ll [], e' \gg \wedge \ll [], eP \gg P{\rightarrow}i \ll \pi, eP' \gg \wedge (\pi A = \pi\ @\ \pi') \wedge \{\$\$\} \vdash e', eP', eV' : \tau) \vee$
$(j \leq i \wedge \ll [], eP \gg P{\rightarrow}j \ll \pi 0\ @\ [s], eP' \gg \wedge (\pi A = \pi 0\ @\ [s']\ @\ \pi') \wedge s \neq s' \wedge hash\ s = hash\ s')$
and *collision*: $hash\ (Inj1\ Unit) = hash\ (Inj2\ Unit)$
and *no_collision_with_Unit*: $\bigwedge t.\ hash\ Unit = hash\ t \Longrightarrow t = Unit$
shows *False*
⟨*proof*⟩

Alternative, shorter counterexample not in administrative normal form.

lemma *security_false_alt*:
assumes *agree*: $\bigwedge e\ eP\ eV\ \tau\ \pi A\ i\ \pi'\ eV'.\ [\![\ \{\$\$\} \vdash e, eP, eV : \tau; \ll \pi A, eV \gg V{\rightarrow}i \ll \pi', eV' \gg\ ]\!] \Longrightarrow$
$\exists e'\ eP'\ \pi\ j\ \pi 0\ s\ s'.\ (\ll [], e \gg I{\rightarrow}i \ll [], e' \gg \wedge \ll [], eP \gg P{\rightarrow}i \ll \pi, eP' \gg \wedge (\pi A = \pi\ @\ \pi') \wedge \{\$\$\} \vdash e', eP', eV' : \tau) \vee$
$(j \leq i \wedge \ll [], eP \gg P{\rightarrow}j \ll \pi 0\ @\ [s], eP' \gg \wedge (\pi A = \pi 0\ @\ [s']\ @\ \pi') \wedge s \neq s' \wedge hash\ s = hash\ s')$
and *collision*: $hash\ (Inj1\ Unit) = hash\ (Inj2\ Unit)$
and *no_collision_with_Unit*: $\bigwedge t.\ hash\ Unit = hash\ t \Longrightarrow t = Unit$
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 $(\exists e'\ eP'\ \pi.\ \ll [], e \gg I{\rightarrow}i \ll [], e' \gg \wedge \ll [], eP \gg P{\rightarrow}i \ll \pi, eP' \gg \wedge\ \pi_A = \pi\ @\ \pi' \wedge \{\$\$\} \vdash e', eP', eV' : \tau) \vee$
$(\exists eP'\ j\ \pi_0\ \pi_0'\ s\ s'.\ j \leq i \wedge \ll [], eP \gg P{\rightarrow}j \ll \pi_0\ @\ [s], eP' \gg \wedge\ \pi_A = \pi_0\ @\ [s']\ @\ \pi_0'\ @\ \pi' \wedge s \neq s' \wedge hash\ s = hash\ s' \wedge closed\ s \wedge closed\ s')$
⟨*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** $\{\$\$\} \vdash e', eP', eV' : \tau \wedge \ll [], e \gg I{\rightarrow} \ll [], e' \gg \wedge \ll \pi, eV \gg V{\rightarrow} \ll [], eV' \gg$
⟨*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.