POPLmark, locally nameless, in Coq

Xavier Leroy

October 10, 2005

A solution to part 1 of the POPLmark challenge using a locally nameless representation of types:

- bound variables are represented by de Bruijn indices, therefore alpha-equivalence is term equality;
- free variables are represented by names, therefore statements are close to those of the paper proof.

For background and motivations for this mixed representation, see Randy Pollack's talk "Reasoning About Languages with Binding",

`http://homepages.inf.ed.ac.uk/rap/export/bindingChallenge_slides.pdf`

Require Import *Arith*.
Require Import *ZArith*.
Require Import *List*.
Require Import *extralibrary*.

1 Names

We use names (also called atoms) to represent free variables in terms. Any infinite type with decidable equality will do. In preparation for the second part of the challenge, we attach a kind to every name: either type or term, and ensure that there are infinitely many names of each kind. Concretely, we represent names by pairs of a kind and an integer (type Z).

Inductive *name_kind* : *Set* :=
  | *TYPE*: *name_kind*
  | *TERM*: *name_kind*.

Definition *name* : *Set* := (*name_kind* × *Z*)%*type*.

Definition *kind* (*n*: *name*) : *name_kind* := *fst n*.

Equality between names is decidable.

Lemma *eq_name*: ∀ (*n1 n2*: *name*), {*n1* = *n2*} + {*n1* ≠ *n2*}.
Proof.

assert ($\forall$ *k1 k2*: *name_kind*, $\{k1 = k2\} + \{k1 \neq k2\}$).
decide equality.
generalize Z_eq_dec; *intro.*
decide equality.
Qed.

Moreover, we have the following obvious simplification rules on tests over name equality.

Lemma *eq_name_true*:
$\forall$ (*A*: *Set*) (*n*: *name*) (*a b*: *A*),
(*if eq_name n n then a else b*) = *a*.
Proof.
intros. case (*eq_name n n*); *congruence.*
Qed.

Lemma *eq_name_false*:
$\forall$ (*A*: *Set*) (*n m*: *name*) (*a b*: *A*),
$n \neq m \rightarrow$ (*if eq_name n m then a else b*) = *b*.
Proof.
intros. case (*eq_name n m*); *congruence.*
Qed.

The following lemma shows that there always exists a name of the given kind that is fresh w.r.t. the given list of names, that is, distinct from all the names in this list.

Lemma *fresh_name*:
$\forall$ (*k*: *name_kind*) (*l*: *list name*), $\exists$ *n*, $\neg$*In n l* $\wedge$ *kind n* = *k*.
Proof.
intros.
set (*ident*:= *fun* (*n*: *name*) $\Rightarrow$ *snd n*).
set (*maxid*:=
fold_right (*fun* (*n*:*name*) *x* $\Rightarrow$ *Zmax* (*ident n*) *x*) 0%*Z*).
assert ($\forall$ *x*, *In x l* $\rightarrow$ (*ident x* $\leq$ *maxid l*)%*Z*).
generalize l. induction l0; *simpl*; *intros.*
elim H.
elim H; *intros. subst x. apply Zmax1.*
apply Zle_trans with (*maxid l0*). *auto. apply Zmax2.*
$\exists$ (*k*, 1 + *maxid l*)%*Z*.
split. red; *intro. generalize* (*H _ H0*). *unfold ident*, *snd. omega.*
reflexivity.
Qed.

1.1 Swaps

As argued by Pitts and others, swaps (permutations of two names) are an interesting special case of renamings. We will use swaps later to prove that our definitions are insensitive to the choices of fresh identifiers, as suggested in Pollack's talk.

Definition $swap\ (u\ v\ x\colon\ name) : name :=$
  $if\ \ eq\_name\ x\ u\ then\ v\ else\ if\ \ eq\_name\ x\ v\ then\ u\ else\ x.$

The following lemmas are standard properties of swaps: self-inverse, injective, kind-preserving.

Lemma $swap\_left$: $\forall\ x\ y,\ swap\ x\ y\ x = y.$
Proof. $intros.\ unfold\ swap.\ apply\ eq\_name\_true.$ Qed.

Lemma $swap\_right$: $\forall\ x\ y,\ swap\ x\ y\ y = x.$
Proof.
  $intros.\ unfold\ swap.\ case\ (eq\_name\ y\ x);\ intro.\ auto.$
  $apply\ eq\_name\_true.$
Qed.

Lemma $swap\_other$: $\forall\ x\ y\ z,\ z \neq x \rightarrow z \neq y \rightarrow swap\ x\ y\ z = z.$
Proof. $intros.\ unfold\ swap.\ repeat\ rewrite\ eq\_name\_false;\ auto.$ Qed.

Lemma $swap\_inv$:
  $\forall\ u\ v\ x,\ swap\ u\ v\ (swap\ u\ v\ x) = x.$
Proof.
  $intros;\ unfold\ swap.$
  $case\ (eq\_name\ x\ u);\ intro.$
    $case\ (eq\_name\ v\ u);\ intro.\ congruence.\ rewrite\ eq\_name\_true.\ congruence.$
  $case\ (eq\_name\ x\ v);\ intro.$
    $rewrite\ eq\_name\_true.\ congruence.$
  $repeat\ rewrite\ eq\_name\_false;\ auto.$
Qed.

Lemma $swap\_inj$:
  $\forall\ u\ v\ x\ y,\ swap\ u\ v\ x = swap\ u\ v\ y \rightarrow x = y.$
Proof.
  $intros.\ rewrite \leftarrow (swap\_inv\ u\ v\ x).\ rewrite \leftarrow (swap\_inv\ u\ v\ y).$
  $congruence.$
Qed.

Lemma $swap\_kind$:
  $\forall\ u\ v\ x,\ kind\ u = kind\ v \rightarrow kind\ (swap\ u\ v\ x) = kind\ x.$
Proof.
  $intros.\ unfold\ swap.\ case\ (eq\_name\ x\ u);\ intro.$
  $congruence.\ case\ (eq\_name\ x\ v);\ intro.$
  $congruence.\ auto.$
Qed.

2 Types and typing environments

2.1 Type expressions

The syntax of types is standard, except that we have two representations for variables: *Tparam* represents free type variables, identified by a name, while *Tvar* represents bound type variables, identified by their de Bruijn indices. In a *Forall t1 t2* type, the variable *Tvar* 0 is bound by the *Forall* in type *t2*.

```
Inductive type: Set :=
  | Tparam: name → type
  | Tvar: nat → type
  | Top: type
  | Arrow: type → type → type
  | Forall: type → type → type.
```

The free names of a type are as follow. Notice the *Forall* case: *Forall* does not bind any name.

```
Fixpoint fv_type (t: type) : list name :=
  match t with
  | Tparam x ⇒ x :: nil
  | Tvar n ⇒ nil
  | Top ⇒ nil
  | Arrow t1 t2 ⇒ fv_type t1 ++ fv_type t2
  | Forall t1 t2 ⇒ fv_type t1 ++ fv_type t2
  end.
```

There are two substitution operations over types, written *vsubst* and *psubst* in Pollack's talk. *vsubst* substitutes a type for a bound variable (a de Bruijn index). *psubst* substitutes a type for a free variable (a name).

The crucial observation is that variable capture cannot occur during either substitution:

- Types never contain free de Bruijn indices, since these indices are used only for representing bound variables. Therefore, *vsubst* does not need to perform lifting of de Bruijn indices in the substituted type.
- Types never bind names, only de Bruijn indices. Therefore, *psubst* never needs to perform renaming of names in the substituted term when descending below a binder.

```
Fixpoint vsubst_type (a: type) (x: nat) (b: type) {struct a} : type :=
  match a with
  | Tparam n ⇒ Tparam n
  | Tvar n ⇒
      match compare_nat n x with
      | Nat_less _ ⇒ Tvar n
      | Nat_equal _ ⇒ b
      | Nat_greater _ ⇒ Tvar (pred n)
      end
  | Top ⇒ Top
```

```
  | Arrow a1 a2 ⇒ Arrow (vsubst_type a1 x b) (vsubst_type a2 x b)
  | Forall a1 a2 ⇒ Forall (vsubst_type a1 x b) (vsubst_type a2 (S x) b)
  end.
```

```
Fixpoint psubst_type (a: type) (x: name) (b: type) {struct a} : type :=
  match a with
  | Tparam n ⇒ if eq_name n x then b else Tparam n
  | Tvar n ⇒ Tvar n
  | Top ⇒ Top
  | Arrow a1 a2 ⇒ Arrow (psubst_type a1 x b) (psubst_type a2 x b)
  | Forall a1 a2 ⇒ Forall (psubst_type a1 x b) (psubst_type a2 x b)
  end.
```

In the remainder of the development, *vsubst* is only used to replace bound variable 0 by a fresh, free variable (a name) when taking apart a *Forall* type. This operation is similar to the "freshening" operation used in Fresh ML et al. Let's call it *freshen_type* for clarity.

```
Definition freshen_type (a: type) (x: name) : type :=
  vsubst_type a 0 (Tparam x).
```

Free variables and freshening play well together.

```
Lemma fv_type_vsubst_type:
  ∀ x a n b, In x (fv_type a) → In x (fv_type (vsubst_type a n b)).
Proof.
  induction a; simpl; intros.
  auto. contradiction. contradiction.
  elim (in_app_or _ _ _ H); auto.
  elim (in_app_or _ _ _ H); auto.
Qed.
```

```
Lemma fv_type_freshen_type:
  ∀ x a y, In x (fv_type a) → In x (fv_type (freshen_type a y)).
Proof.
  intros; unfold freshen_type; apply fv_type_vsubst_type; auto.
Qed.
```

We now define swaps (permutation of names) over types and show basic properties of swaps that will be useful later.

```
Fixpoint swap_type (u v: name) (t: type) {struct t} : type :=
  match t with
  | Tparam x ⇒ Tparam (swap u v x)
  | Tvar n ⇒ Tvar n
  | Top ⇒ Top
  | Arrow t1 t2 ⇒ Arrow (swap_type u v t1) (swap_type u v t2)
  | Forall t1 t2 ⇒ Forall (swap_type u v t1) (swap_type u v t2)
  end.
```

Swaps are involutions (self-inverse).

Lemma *swap_type_inv*:
  $\forall\ u\ v\ t,\ swap\_type\ u\ v\ (swap\_type\ u\ v\ t) = t.$
Proof.
  induction t; simpl; try congruence.
  rewrite swap_inv. auto.
Qed.

Swaps of variables that do not occur free in a type leave the type unchanged.

Lemma *swap_type_not_free*:
  $\forall\ u\ v\ t,$
  $\neg In\ u\ (fv\_type\ t) \rightarrow \neg In\ v\ (fv\_type\ t) \rightarrow swap\_type\ u\ v\ t = t.$
Proof.
  induction t; simpl; intros; try (auto; decEq; eauto).
  unfold swap. repeat rewrite eq_name_false. auto. tauto. tauto.
Qed.

Swaps commute with *vsubst* substitution and freshening.

Lemma *vsubst_type_swap*:
  $\forall\ u\ v\ a\ n\ b,$
  $swap\_type\ u\ v\ (vsubst\_type\ a\ n\ b) = vsubst\_type\ (swap\_type\ u\ v\ a)\ n\ (swap\_type\ u\ v\ b).$
Proof.
  induction a; simpl; intros; auto; try (decEq; auto).
  case (compare_nat n n0); auto.
Qed.

Lemma *freshen_type_swap*:
  $\forall\ u\ v\ a\ x,$
  $swap\_type\ u\ v\ (freshen\_type\ a\ x) = freshen\_type\ (swap\_type\ u\ v\ a)\ (swap\ u\ v\ x).$
Proof.
  intros; unfold freshen_type.
  change (Tparam (swap u v x)) with (swap_type u v (Tparam x)).
  apply vsubst_type_swap.
Qed.

Swaps and free variables.

Lemma *in_fv_type_swap*:
  $\forall\ u\ v\ x\ t,$
  $In\ x\ (fv\_type\ t) \leftrightarrow In\ (swap\ u\ v\ x)\ (fv\_type\ (swap\_type\ u\ v\ t)).$
Proof.
  induction t; simpl.
  intuition. subst n. tauto.
  left. eapply swap_inj; eauto.
  tauto. tauto.
  auto. auto.
Qed.

2.2 Typing environments

Typing environments are standard: lists of (name, type) pairs. Bindings are added to the left of the environment using the *cons* list operation. Thus, later bindings come first.

Definition *typenv* := *list* (*name* × *type*).

Definition *dom* (*e*: *typenv*) := *map* (@*fst name type*) *e*.

Looking up the type associated with a name in a typing environment.

```
Fixpoint lookup (x: name) (e: typenv) {struct e} : option type :=
  match e with
  | nil ⇒ None
  | (y, t) :: e' ⇒
      if eq_name x y then Some t else lookup x e'
  end.
```

```
Lemma lookup_inv:
  ∀ x t e, lookup x e = Some t → In x (dom e).
Proof.
  induction e; simpl. congruence.
  case a; intros x' t'. simpl.
  case (eq_name x x'); intro. subst x'; tauto.
  intros. right. auto.
Qed.
```

```
Lemma lookup_exists:
  ∀ x e, In x (dom e) → ∃ t, lookup x e = Some t.
Proof.
  induction e; simpl; intros.
  elim H.
  destruct a. simpl in H.
  case (eq_name x n); intro.
  ∃ t; auto.
  apply IHe. elim H; intro. congruence. auto.
Qed.
```

Swaps over environments.

```
Fixpoint swap_env (u v: name) (e: typenv) {struct e} : typenv :=
  match e with
  | nil ⇒ nil
  | (x, t) :: e' ⇒ (swap u v x, swap_type u v t) :: swap_env u v e'
  end.
```

Environment lookup commutes with swaps.

```
Lemma lookup_swap:
  ∀ u v x e t,
  lookup x e = Some t →
```

lookup (*swap* *u* *v* *x*) (*swap_env* *u* *v* *e*) = *Some* (*swap_type* *u* *v* *t*).
Proof.
induction *e*; *simpl.*
intros. *congruence.*
case *a*; *intros* *y* *t* *t'*. *simpl.*
case (*eq_name* *x* *y*); *intros.*
subst *y*. *rewrite* *eq_name_true.* *congruence.*
rewrite *eq_name_false.* *auto.*
generalize (*swap_inj* *u* *v* *x* *y*). *tauto.*
Qed.

The *dom* operation commutes with swaps.

Lemma *in_dom_swap*:
∀ *u* *v* *x* *e*,
In *x* (*dom* *e*) ↔ *In* (*swap* *u* *v* *x*) (*dom* (*swap_env* *u* *v* *e*)).
Proof.
induction *e*; *simpl.*
tauto.
case *a*; *intros* *y* *t*. *simpl.* *intuition.*
subst *x*. *intuition.*
left. *eapply* *swap_inj*; *eauto.*
Qed.

2.3 Well-formedness of types and environments

A type is well-formed in a typing environment if:

- all names free in the type are of kind TYPE
- all names free in the type are bound in the environment
- it does not contain free de Bruijn variables.

We capture these conditions by the following inference rules.

Inductive *wf_type*: *typenv* → *type* → *Prop* :=
| *wf_type_param*: ∀ *x* *e* *t*,
kind *x* = *TYPE* →
lookup *x* *e* = *Some* *t* →
wf_type *e* (*Tparam* *x*)
| *wf_type_top*: ∀ *e*,
wf_type *e* *Top*
| *wf_type_arrow*: ∀ *e* *t1* *t2*,
wf_type *e* *t1* → *wf_type* *e* *t2* → *wf_type* *e* (*Arrow* *t1* *t2*)
| *wf_type_forall*: ∀ *e* *t1* *t2*,
wf_type *e* *t1* →

```
    (∀ x,
        kind x = TYPE →
        ¬In x (fv_type t2) → ¬In x (dom e) →
        wf_type ((x, t1) :: e) (freshen_type t2 x)) →
    wf_type e (Forall t1 t2).
```

The rules are straightforward, except perhaps the *wf_type_forall* rule. It follows a general pattern for operating over sub-terms of a binder, such as *t2* in *Forall t1 t2*. The de Bruijn variable *Tvar* 0 is potentially free in *t2*. To recover a well-formed term, without free de Bruijn variables, we substitute *Tvar* 0 with a fresh name *x*. Therefore, the premise for *t2* applies to *freshen_type t2 x*.

How should *x* be chosen? As in the name-based specification, *x* must not be in the domain of *e*, otherwise the extended environment (*x*, *t1*) :: *e* would be ill-formed. In addition, the name *x* must not be free in *t2*, otherwise the freshening *freshen_type t2 x* would incorrectly identify the bound, universally-quantified variable of the *Forall* types with an existing, free type variable.

How should *x* be quantified? That is, should the second premise *wf_type* ((*x*, *t1*) :: *e*) (*freshen_type t2 x*)) hold for one particular name *x* not in *dom*(*e*), or for all names *x* not in *dom*(*e*)? The "for all" alternative obviously leads to a stronger induction principle: proofs that proceed by inversion or induction over an hypothesis *wf_type e* (*Forall t1 t2*) can then choose any convenient *x* fresh for *e* to exploit the second premise, rather than having to cope with a fixed, earlier choice of *x*. Symmetrically, the "for one" alternative is more convenient for proofs that must conclude *wf_type e* (*Forall t1 t2*): it suffices to exhibit one suitable *x* fresh in *e* that satisfies the second premise, rather than having to establish the second premise for all such *x*.

The crucial observation is that those two alternative are equivalent: the same subtyping judgements can be derived with the "for all" rule and the "for one" rule. (See Pollack's talk for more explanations.) Therefore, in the definition of the *wf_type* predicate above, we chose the "for all" rule, so as to get the strongest induction principle. And we will show shortly that the "for one" rule is admissible and can be used in proofs that conclude *wf_type e* (*Forall t1 t2*).

An environment is well-formed if every type it contains is well-formed in the part of the environment that occurs to its right, i.e. the environment at the time this type was introduced. This ensures in particular that all the variables in this type are bound earlier (i.e. to the right) in the environment. Moreover, we impose that no name is bound twice in an environment.

```
Inductive wf_env: typenv → Prop :=
  | wf_env_nil:
        wf_env nil
  | wf_env_cons:
        ∀ x t e,
        wf_env e → ¬In x (dom e) → wf_type e t →
        wf_env ((x, t) :: e).

Lemma wf_type_env_incr:
  ∀ e t, wf_type e t →
  ∀ e', incl (dom e) (dom e') → wf_type e' t.
Proof.
  induction 1; simpl; intros.
```

assert (∃ *t'*, *lookup x e'* = *Some t'*).
apply lookup_exists. apply H1. eapply lookup_inv; *eauto.*
elim H2; *intros. econstructor*; *eauto.*
constructor.
constructor; *auto.*
constructor; *auto. intros. apply H1*; *auto.*
red; *simpl*; *intros. generalize* (*H2 a*). *tauto.*
Qed.

A type well formed in e has all its free names in the domain of e.

Lemma *fv_wf_type*:
∀ *x e t*,
wf_type e t → *In x* (*fv_type t*) → *In x* (*dom e*).
Proof.
induction 1; *simpl*; *intros.*
replace x with x0. eapply lookup_inv; *eauto. tauto.*
elim H.
elim (*in_app_or* _ _ _ *H1*); *auto.*
elim (*in_app_or* _ _ _ *H2*); *auto.*
intro. elim (*fresh_name TYPE* (*x* :: *fv_type t2* ++ *dom e*)); *intros y* [FRESH KIND].
assert (*In x* (*dom* ((*y*, *t1*) :: *e*))).
apply H1; *eauto. apply fv_type_freshen_type. auto.*
elim H4; *auto.*
simpl; *intro*; *subst y. simpl in FRESH. tauto.*
Qed.

Looking up the type of a name in a well-formed environment returns a well-formed type.

Lemma *wf_type_lookup*:
∀ *e*, *wf_env e* →
∀ *x t*, *lookup x e* = *Some t* → *wf_type e t*.
Proof.
induction 1; *intros.*
discriminate.
apply wf_type_env_incr with e.
simpl in H2. destruct (*eq_name x0 x*). *replace t0 with t. auto. congruence.*
eauto.
simpl. red; *intros*; *apply in_cons*; *auto.*
Qed.

Type well-formedness is stable by swapping.

Lemma *wf_type_swap*:
∀ *u v e t*,
kind u = *kind v* →
wf_type e t → *wf_type* (*swap_env u v e*) (*swap_type u v t*).
Proof.
intros until t. intro KIND.

induction 1; *simpl*; *intros*.
apply wf_type_param with (*swap_type u v t*).
rewrite swap_kind; *auto*.
apply lookup_swap; *auto*.
apply wf_type_top.
apply wf_type_arrow; *auto*.
apply wf_type_forall; *auto*.
intros.
pose (*x':= swap u v x*).
assert (*x = swap u v x'*). *unfold x'*; *symmetry*; *apply swap_inv*.
assert (~*In x'* (*fv_type t2*)). *rewrite H5 in H3*.
generalize (*in_fv_type_swap u v x' t2*). *tauto*.
assert (~*In x'* (*dom e*)). *rewrite H5 in H4*.
generalize (*in_dom_swap u v x' e*). *tauto*.
assert (*kind x' = TYPE*).
unfold x'. *rewrite swap_kind*; *auto*.
generalize (*H1 x' H8 H6 H7*). *simpl*.
rewrite freshen_type_swap. *simpl*. *rewrite* ← *H5*.
auto.

Qed.

Environment well-formedness is stable by swapping.

Lemma *wf_env_swap*:
∀ *u v e*,
kind u = kind v → *wf_env e* → *wf_env* (*swap_env u v e*).

Proof.
induction 2; *simpl*.
constructor.
constructor. *auto*.
generalize (*in_dom_swap u v x e*). *tauto*.
apply wf_type_swap; *auto*.

Qed.

Well-formed environments are invariant by swaps of names that are not in the domains of the environments.

Lemma *swap_env_not_free*:
∀ *u v e*,
wf_env e → ¬*In u* (*dom e*) → ¬*In v* (*dom e*) → *swap_env u v e = e*.

Proof.
induction 1; *simpl*; *intros*.
auto.
decEq. *decEq*. *apply swap_other*; *tauto*.
apply swap_type_not_free.
generalize (*fv_wf_type u e t H1*). *tauto*.
generalize (*fv_wf_type v e t H1*). *tauto*.
apply IHwf_env. *tauto*. *tauto*.

Qed.

We now show that the alternate formulation of rule *wf_type_forall* (the one with "for one fresh name x" instead of "for all fresh names x" in the second premise) is admissible.

```
Lemma wf_type_forall':
  ∀ e x t t1 t2,
  wf_env e →
  wf_type e t1 →
  kind x = TYPE →
  ¬In x (fv_type t2) →
  ¬In x (dom e) →
  wf_type ((x, t) :: e) (freshen_type t2 x) →
  wf_type e (Forall t1 t2).
Proof.
  intros. constructor. auto. intros.
  assert (kind x = kind x0). congruence.
  generalize (wf_type_swap x x0 _ _ H8 H4). simpl.
  rewrite swap_left. rewrite swap_env_not_free; auto.
  rewrite freshen_type_swap. simpl. rewrite swap_left.
  rewrite (swap_type_not_free x x0 t2); auto.
  intro. eapply wf_type_env_incr; eauto. simpl. apply incl_refl.
Qed.
```

3 Algorithmic subtyping

We now define the subtyping judgement as an inductive predicate. Each constructor of the predicate corresponds to an inference rule in the original definition of subtyping.

```
Inductive is_subtype: typenv → type → type → Prop :=
  | sa_top: ∀ e s,
      wf_env e →
      wf_type e s →
      is_subtype e s Top
  | sa_refl_tvar: ∀ e x u,
      wf_env e →
      kind x = TYPE →
      lookup x e = Some u →
      is_subtype e (Tparam x) (Tparam x)
  | sa_trans_tvar: ∀ e x u t,
      kind x = TYPE →
      lookup x e = Some u →
      is_subtype e u t →
      is_subtype e (Tparam x) t
  | sa_arrow: ∀ e s1 s2 t1 t2,
      is_subtype e t1 s1 →
```

```
    is_subtype e s2 t2 →
    is_subtype e (Arrow s1 s2) (Arrow t1 t2)
| sa_all: ∀ e s1 s2 t1 t2,
    is_subtype e t1 s1 →
    (∀ x,
      kind x = TYPE →
      ¬In x (dom e) →
      is_subtype ((x, t1) :: e) (freshen_type s2 x) (freshen_type t2 x)) →
    is_subtype e (Forall s1 s2) (Forall t1 t2).
```

The *sa_all* rule for *Forall* types follows the patter that we already introduced for *wf_type*, rule *wf_type_forall*. In the original, name-based specification, we say that $E \vdash (\forall x <: \sigma_1.\ \sigma_2) <: (\forall x <: \tau_1.\ \tau_2)$ if $E \vdash \tau_1 <: \sigma_1$ and $E,\ x : \tau_1 \vdash \sigma_2 <: \tau_2$. The type variable x, being α-convertible in the conclusion, is (implicitly or explicitly) chosen so that $E,\ x : \tau_1$ is well-formed in the second premise, that is, x is chosen not free in E.

In our mixed, name / de Bruijn representation, the type variables bound by *Forall* in the conclusion do not have names. We must therefore invent a suitable name x and substitute it for the bound variable *TVar* 0 in the types *s2* and *t2*. Therefore, the second premise puts *freshen_type s2 x* and *freshen_type t2 x* in subtype relation.

As mentioned already, x should be chosen not in the domain of e (otherwise the extended environment $(x,\ t1) :: e$ would be ill-formed) and not free in *s2* and *t2*, otherwise the freshenings *freshen_type s2 x* and *freshen_type t2 x* would incorrectly identify the bound, universally-quantified variable of the *Forall* types with an existing, free type variable. However, as we will prove below, the rules for *is_subtype* satisfy a well-formedness condition: if *is_subtype e u1 u2*, then *u1* and *u2* are well-formed in e, implying that a name not in the domain of e cannot be free in *u1* or *u2*. Therefore, the condition "x not in the domain of e" suffices to ensure that x is not free in *s2* and *t2*, and therefore that the freshenings *freshen_type s2 x* and *freshen_type t2 x* make sense.

As mentioned already as well, we have a choice between quantifying over all suitable x or over one suitable x in the second premise. Again, we go with the "for all" alternative in order to obtain the strongest induction principle, and we will show later that the "for one" alternative is derivable.

For the time being, we start with simple well-formedness properties of the types and environments involved in a *is_subtype* relation.

```
Lemma is_subtype_wf_env:
  ∀ e s t, is_subtype e s t → wf_env e.
Proof.
  induction 1; intros; eauto.
Qed.

Lemma is_subtype_wf_type:
  ∀ e s t, is_subtype e s t → wf_type e s ∧ wf_type e t.
Proof.
  induction 1; intros.
  split. auto. apply wf_type_top.
  split; apply wf_type_param with u; auto.
```

split. apply wf_type_param with u; auto. tauto.
split; apply wf_type_arrow; tauto.
elim IHis_subtype; intros.
elim (fresh_name TYPE (dom e ++ fv_type s2 ++ fv_type t2)).
intros x [FRESH KIND].
assert (~In x (dom e)). eauto.
elim (H1 x KIND H4); intros.
split; eapply wf_type_forall'; eauto.
eapply is_subtype_wf_env; eauto.
eapply is_subtype_wf_env; eauto.
Qed.

Lemma *is_subtype_wf_type_l*:
$\forall$ *e s t, is_subtype e s t* $\rightarrow$ *wf_type e s.*
Proof.
intros. elim (is_subtype_wf_type _ _ _ H); auto.
Qed.

Lemma *is_subtype_wf_type_r*:
$\forall$ *e s t, is_subtype e s t* $\rightarrow$ *wf_type e t.*
Proof.
intros. elim (is_subtype_wf_type _ _ _ H); auto.
Qed.

Hint *Resolve is_subtype_wf_env is_subtype_wf_type_l is_subtype_wf_type_r.*

We now show that the *is_subtype* predicate is stable by swapping. This property is crucial to show the equivalence of the "for all" and "for one" interpretations of rule *sa_all*.

Lemma *is_subtype_swap*:
$\forall$ *u v, kind u = kind v* $\rightarrow$
$\forall$ *e s t,*
is_subtype e s t $\rightarrow$
is_subtype (swap_env u v e) (swap_type u v s) (swap_type u v t).
Proof.
intros u v KINDuv. induction 1; simpl.
apply sa_top. apply wf_env_swap; auto. apply wf_type_swap; auto.
eapply sa_refl_tvar.
apply wf_env_swap; auto.
rewrite swap_kind; auto.
apply lookup_swap; eauto.
eapply sa_trans_tvar.
rewrite swap_kind; auto.
apply lookup_swap; eauto. auto.
apply sa_arrow; auto.
apply sa_all. auto.
intros.
pose (x':= swap u v x).

assert (*kind x'* = *TYPE*).
  unfold x'. rewrite swap_kind; auto.
assert (~*In x'* (*dom e*)).
  generalize (*in_dom_swap u v x' e*). *unfold x'. rewrite swap_inv. tauto.*
generalize (*H1 x' H4 H5*). *repeat rewrite freshen_type_swap. simpl.*
replace (*swap u v x'*) *with x. auto. unfold x'. rewrite swap_inv. auto.*

Qed.

Two silly lemmas about freshness of names in types.

Lemma *fresh_wf_type*:
  $\forall$ *x e t*,
  wf_type e t $\rightarrow \neg$*In x* (*dom e*) $\rightarrow \neg$*In x* (*fv_type t*).
Proof.
  intros. generalize (*fv_wf_type x _ _ H*). *tauto.*
Qed.

Lemma *fresh_freshen_type*:
  $\forall$ *x t1 e t y*,
  wf_type ((*x*, *t1*) :: *e*) (*freshen_type t x*) $\rightarrow$
  $\neg$*In y* (*dom e*) $\rightarrow x \neq y \rightarrow$
  $\neg$*In y* (*fv_type t*).
Proof.
  intros.
  red; *intro.*
  assert (*In y* (*dom* ((*x*, *t1*) :: *e*))).
    eapply fv_wf_type. eauto. apply fv_type_freshen_type. auto.
  elim H3; *simpl*; *intros. contradiction. contradiction.*
Qed.

We now show that the alternate presentation of rule *sa_all* (the one with “for one name” in the second premise instead of “for all names”) is admissible.

Lemma *sa_all'*:
  $\forall$ *e s1 s2 t1 t2 x*,
  is_subtype e t1 s1 $\rightarrow$
  kind x = *TYPE* $\rightarrow$
  $\neg$*In x* (*dom e*) $\rightarrow \neg$*In x* (*fv_type s2*) $\rightarrow \neg$*In x* (*fv_type t2*) $\rightarrow$
  is_subtype ((*x*, *t1*) :: *e*) (*freshen_type s2 x*) (*freshen_type t2 x*) $\rightarrow$
  is_subtype e (*Forall s1 s2*) (*Forall t1 t2*).
Proof.
  intros. apply sa_all. auto.
  intros y DOM.
  case (*eq_name x y*); *intro. subst y. auto.*
  elim (*is_subtype_wf_type _ _ _ H*). *intros.*
  elim (*is_subtype_wf_type _ _ _ H4*). *intros.*
  assert (~*In y* (*fv_type s2*)).
    apply fresh_freshen_type with x t1 e; *auto.*

```
  assert (~In y (fv_type t2)).
    apply fresh_freshen_type with x t1 e; auto.
  replace ((y, t1) :: e) with (swap_env x y ((x, t1) :: e)).
  replace (freshen_type s2 y)
     with (swap_type x y (freshen_type s2 x)).
  replace (freshen_type t2 y)
     with (swap_type x y (freshen_type t2 x)).
  apply is_subtype_swap; auto. congruence.
  rewrite freshen_type_swap. rewrite swap_left.
  rewrite swap_type_not_free; auto.
  rewrite freshen_type_swap. simpl. rewrite swap_left.
  rewrite swap_type_not_free; auto.
  simpl. rewrite swap_left. rewrite swap_env_not_free; auto.
  rewrite swap_type_not_free. auto.
  apply fresh_wf_type with e; auto.
  apply fresh_wf_type with e; auto.
  eapply is_subtype_wf_env; eauto.
Qed.
```

4 The challenge, part 1

We now turn (at last!) to proving the two theorems of part 1 of the POPLmark challenge: reflexivity and transitivity of subtyping.

Transitivity of subtyping is shown by straightforward induction on the derivation of well-formedness of the type. As noted by Pollack in his talk, such inductions conveniently replace inductions on the structure of types.

```
Theorem is_subtype_refl:
  ∀ t e, wf_type e t → wf_env e → is_subtype e t t.
Proof.
  induction 1; intros.
  apply sa_refl_tvar with t; auto.
  apply sa_top. auto. constructor.
  apply sa_arrow. auto. auto.
  elim (fresh_name TYPE (fv_type t2 ++ dom e));
  intros x [FRESH KIND].
  apply sa_all' with x; eauto.
  apply H1; eauto.
  constructor; eauto.
Qed.
```

We now do some scaffolding work for the proof of transitivity. First, we will need to perform inductions over the size of types. We cannot just do inductions over the structure of types, as the original paper proof did, because in the case of *Forall t1 t2*, we will need to recurse not on *t2* but on *freshen_type t2 x* for some x, which is not a sub-term of *t2*. However, the size of *freshen_type t2 x* is the same as the size of *t2*, so induction over sizes will work.

```
Fixpoint size_type (t: type): nat :=
  match t with
  | Tparam _ ⇒ 1
  | Tvar _ ⇒ 1
  | Top ⇒ 1
  | Arrow t1 t2 ⇒ size_type t1 + size_type t2
  | Forall t1 t2 ⇒ size_type t1 + size_type t2
  end.

Lemma size_type_pos:
  ∀ t, size_type t > 0.
Proof.
  induction t; simpl; omega.
Qed.

Lemma vsubst_type_size:
  ∀ x a n, size_type (vsubst_type a n (Tparam x)) = size_type a.
Proof.
  induction a; simpl; intros; auto.
  case (compare_nat n n0); reflexivity.
Qed.

Lemma freshen_type_size:
  ∀ x a, size_type (freshen_type a x) = size_type a.
Proof.
  unfold freshen_type; intros. apply vsubst_type_size.
Qed.
```

We now define a notion of inclusion between environments that we call "weakening".

```
Definition env_weaken (e1 e2: typenv) : Prop :=
  ∀ x t, lookup x e1 = Some t → lookup x e2 = Some t.

Lemma env_weaken_incl_dom:
  ∀ e1 e2, env_weaken e1 e2 → incl (dom e1) (dom e2).
Proof.
  unfold incl; intros.
  elim (lookup_exists _ _ H0). intros t LOOKUP.
  apply lookup_inv with t. apply H. auto.
Qed.

Lemma sub_weaken:
  ∀ e s t, is_subtype e s t →
  ∀ e', wf_env e' → env_weaken e e' → is_subtype e' s t.
Proof.
  induction 1; intros.
  apply sa_top. auto. apply wf_type_env_incr with e. auto.
```

apply env_weaken_incl_dom; auto.
apply sa_refl_tvar with u; auto.
apply sa_trans_tvar with u; auto.
apply sa_arrow; auto.
apply sa_all; auto. intros. apply H1; auto.
generalize (env_weaken_incl_dom _ _ H3 x). tauto.
constructor; auto. apply wf_type_env_incr with e; eauto.
apply env_weaken_incl_dom; auto.
red; intro y. simpl. case (eq_name y x); auto.
Qed.

Lemma *env_concat_weaken*:
∀ *delta gamma*,
wf_env (delta ++ gamma) → env_weaken gamma (delta ++ gamma).
Proof.
induction delta; simpl; intros.
red; auto.
inversion H. red; intros; simpl.
rewrite eq_name_false. apply IHdelta. auto. auto.
red; intro. subst x0. elim H3.
unfold dom. rewrite map_append. apply in_or_app. right.
apply lookup_inv with t0. auto.
Qed.

The following lemmas prove useful properties of environments of the form *e1 ++ (x, p) :: e2*, that is, all bindings of *e2*, followed by a binding of *p* to *x*, followed by all bindings of *e1*.

Lemma *dom_env_extends*:
∀ *e2 x p q e1*,
dom (e1 ++ (x, p) :: e2) = dom (e1 ++ (x, q) :: e2).
Proof.
induction e1; simpl. auto.
rewrite IHe1; auto.
Qed.

Lemma *wf_env_extends*:
∀ *e2 x p q e1*,
wf_env (e1 ++ (x, p) :: e2) → wf_type e2 q →
wf_env (e1 ++ (x, q) :: e2).
Proof.
induction e1; simpl; intros.
inversion H. constructor; auto.
inversion H. constructor; auto.
rewrite (dom_env_extends e2 x q p e1). auto.
apply wf_type_env_incr with (e1 ++ (x, p) :: e2); auto.
rewrite (dom_env_extends e2 x q p e1). apply incl_refl.
Qed.

Lemma *lookup_env_extends*:
 $\forall$ *e2 x p q y e1*,
 wf_env (*e1* ++ (*x*, *q*) :: *e2*) $\rightarrow$
 lookup y (*e1* ++ (*x*, *p*) :: *e2*) =
 if eq_name y x then Some p else lookup y (*e1* ++ (*x*, *q*) :: *e2*).
Proof.
 induction e1; *simpl*; *intros*.
 case (*eq_name y x*); *auto*.
 destruct a. inversion H. subst x0; *subst t0*; *subst e*.
 case (*eq_name y n*); *intro. subst n*.
 rewrite eq_name_false. auto. red; *intros*; *subst x*.
 elim H4. unfold dom. rewrite map_append.
 apply in_or_app. right. simpl. tauto.
 apply IHe1. auto.
Qed.

Now comes the major result: transitivity and the narrowing property of subtyping, proved simultaneously. The proof follows the structure of the paper proof, with the structural induction on q being replaced by a Peano induction on the size of q.

Lemma *sub_trans_narrow*:
 $\forall$ *n*,
 ($\forall$ *e s q t*,
  size_type q $\leq$ *n* $\rightarrow$
  is_subtype e s q $\rightarrow$ *is_subtype e q t* $\rightarrow$
  is_subtype e s t)
 $\wedge$
 ($\forall$ *x e1 e2 p q r s*,
  size_type q $\leq$ *n* $\rightarrow$
  is_subtype (*e1* ++ (*x*, *q*) :: *e2*) *r s* $\rightarrow$ *is_subtype e2 p q* $\rightarrow$
  is_subtype (*e1* ++ (*x*, *p*) :: *e2*) *r s*).
Proof.
 intro n0. pattern n0. apply Peano_induction.
 intros size HRsize.
Part 1: transitivity
 assert ($\forall$ *e s q*, *is_subtype e s q* $\rightarrow$
   $\forall$ *t*, *size_type q* $\leq$ *size* $\rightarrow$ *is_subtype e q t* $\rightarrow$ *is_subtype e s t*).
Sub-induction on the derivation of *is_subtype e s q*
 induction 1; *intros*.
Case *sa_top*
 inversion H2. apply sa_top. auto. eauto.
Case *sa_refl_tvar*
 auto.
Case *sa_trans_tvar*
 apply sa_trans_tvar with u; *auto*.
Case *sa_arrow*

inversion H2.

apply sa_top. auto. inversion H4. constructor; eauto.

subst e0; subst s0; subst s3.

assert (SZpos: pred size < size).

generalize (size_type_pos (Arrow t1 t2)). omega.

elim (HRsize (pred size) SZpos); intros HR1 HR2.

apply sa_arrow.

Application of the outer induction hypothesis to *t1*

apply HR1 with t1; auto.

simpl in H1. generalize (size_type_pos t2). omega.

Application of the outer induction hypothesis to *t2*

apply HR1 with t2; auto.

simpl in H1. generalize (size_type_pos t1). omega.

Case *sa_forall*

inversion H3.

apply sa_top. auto.

apply is_subtype_wf_type_l with (Forall t1 t2). constructor; assumption.

subst e0; subst s0; subst s3.

assert (SZpos: pred size < size).

generalize (size_type_pos (Forall t1 t2)). omega.

elim (HRsize (pred size) SZpos); intros HR1 HR2.

Choice of an appropriately fresh name x

elim (fresh_name TYPE (dom e ++ fv_type t3 ++ fv_type s2)).

intros x [FRESH KIND].

apply sa_all' with x; eauto.

Application of the outer induction hypothesis to *t1*

apply HR1 with t1; auto.

simpl in H2. generalize (size_type_pos t2). omega.

Application of the outer induction hypothesis to *freshen t2 x*

apply HR1 with (freshen_type t2 x).

rewrite freshen_type_size.

simpl in H2. generalize (size_type_pos t1). omega.

change ((x, t0) :: e) with (nil ++ (x, t0) :: e).

Application of the weakening part of the outer induction hypothesis.

apply HR2 with t1.

simpl in H2. generalize (size_type_pos t2). omega.

simpl. apply H0; eauto. auto. apply H9; eauto.

Part 2: narrowing

assert ($\forall$ *e r s,*

is_subtype e r s $\rightarrow$

$\forall$ *e1 x q e2 p,*

e = e1 ++ (x, q) :: e2 $\rightarrow$ *size_type q* $\leq$ *size* $\rightarrow$ *is_subtype e2 p q* $\rightarrow$

is_subtype (e1 ++ (x, p) :: e2) r s).

Sub-induction on the derivation of *is_subtype e r s*

induction 1; *intros*; *subst e.*

Case *sa_top*

apply sa_top. apply wf_env_extends with q; eauto.
apply wf_type_env_incr with (e1 ++ (x, q) :: e2); auto.
rewrite (dom_env_extends e2 x q p e1). apply incl_refl.

Case *sa_refl_tvar*

apply sa_refl_tvar with (if eq_name x x0 then p else u).
apply wf_env_extends with q; eauto. auto.
rewrite (lookup_env_extends e2 x0 p q x e1 H0).
case (eq_name x x0); auto.

Case sa_trans_tvar

apply sa_trans_tvar with (if eq_name x x0 then p else u).
auto. rewrite (lookup_env_extends e2 x0 p q x e1).
case (eq_name x x0); auto. eauto.
case (eq_name x x0); intro.

sub-case $x = x0$

generalize H1. rewrite (lookup_env_extends e2 x0 q q x e1); eauto.
rewrite e; rewrite eq_name_true; intro EQ; injection EQ; intro; subst u.
apply H with q.
apply sub_weaken with e2; auto.
apply wf_env_extends with q; eauto.
change (e1 ++ (x0, p) :: e2) with (e1 ++ (((x0, p) :: nil) ++ e2)).
rewrite ← app_ass. apply env_concat_weaken.
rewrite app_ass. simpl. apply wf_env_extends with q; eauto.
auto.
apply IHis_subtype with q; auto.

sub-case $x \neq x0$

apply IHis_subtype with q; auto.

Case *sa_arrow*

apply sa_arrow.
apply IHis_subtype1 with q; auto.
apply IHis_subtype2 with q; auto.

Case *sa_forall*

apply sa_all.
apply IHis_subtype with q; auto.
intros.
change ((x0, t1) :: e1 ++ (x, p) :: e2)
with (((x0, t1) :: e1) ++ (x, p) :: e2).
apply H2 with q. auto.
rewrite (dom_env_extends e2 x q p e1). auto.
reflexivity. auto. auto.

Combining the two parts together

split. intros; apply H with q; auto.
intros; apply H0 with (e1 ++ (x, q) :: e2) q; auto.

Qed.

As a corollary, we obtain transitivity of subtyping.

Theorem sub_trans:
  $\forall\ e\ s\ q\ t$,
  $is\_subtype\ e\ s\ q \rightarrow is\_subtype\ e\ q\ t \rightarrow is\_subtype\ e\ s\ t.$
Proof.
  $intros.\ elim\ (sub\_trans\_narrow\ (size\_type\ q));\ intros.$
  $apply\ H1\ with\ q;\ auto.$
Qed.

As well as narrowing. QED.

Theorem sub_narrow:
  $\forall\ x\ e1\ e2\ p\ q\ r\ s$,
  $is\_subtype\ (e1\ ++\ (x, q) :: e2)\ r\ s \rightarrow is\_subtype\ e2\ p\ q \rightarrow$
  $is\_subtype\ (e1\ ++\ (x, p) :: e2)\ r\ s.$
Proof.
  $intros.\ elim\ (sub\_trans\_narrow\ (size\_type\ q));\ intros.$
  $apply\ H2\ with\ q;\ auto.$
Qed.