Chapter 1

Library nameless

```
Set Implicit Arguments.
Require Import LibLN.
```

1.1 Definitions

1.1.1 Grammars

Grammar of types. We have two constructors, one for type variables and one for arrow types of the form *T1* → *T2*.

```
Inductive typ : Set :=
  | typ_var : var → typ
  | typ_arrow : typ → typ → typ.
```

Grammar of pre-terms. We use a locally nameless representation for the simply-typed lambda calculus, where bound variables are represented as natural numbers (de Bruijn indices) and free variables are represented as atoms. The type *var*, defined in the library LibLN_Var, represents 'names' or 'atoms'. One central assumption is that it is always possible to generate an atom fresh for any given finite set of atoms (lemma *var_fresh*).

```
Inductive trm : Set :=
  | trm_bvar : nat → trm
  | trm_fvar : var → trm
  | trm_abs : trm → trm
  | trm_app : trm → trm → trm.
```

We declare the constructors for indices and variables to be coercions. That way, if Coq sees a nat where it expects an exp, it will implicitly insert an application of bvar; and similarly for atoms. In real metatheory developments, we usually do not need such coercions. However, they will be very useful for carrying out examples in this tutorial.

```
Coercion trm_bvar : nat >-> trm.
```

`Coercion` *trm_fvar* : *var* >-> *trm*.

For example, we can encode the expression (\x. Y x) as below. Because "Y" is free variable in this term, we need to assume an atom for this name.

`Parameter` *Y* : *var*.
`Definition` *demo_rep1* := *trm_abs* (*trm_app* *Y* 0).

Another example: the encoding of (\x. \y. (y x))

`Definition` *demo_rep2* := *trm_abs* (*trm_abs* (*trm_app* 0 1)).

Exercise: convert the following lambda calculus term to locally nameless representation: \s. \z. s(s z)

`Definition` *demo_two* := *trm_abs* (*trm_abs* (*trm_app* 1 (*trm_app* 1 0))).

There are two important advantages of the locally nameless representation:

- Alpha-equivalent terms have a unique representation, we're always working up to alpha-equivalence.
- Operations such as free variable substitution and free variable calculation have simple recursive definitions (and therefore are simple to reason about).

Weighed against these advantages are two drawbacks:

- The *trm* datatype admits terms, such as *trm_abs* 3, where indices are unbound. A term is called "locally closed" when it contains no unbound indices.
- We must define *both* bound variable & free variable substitution and reason about how these operations interact with eachother.

1.1.2 Opening

Opening replaces an index with a term. It corresponds to informal substitution for a bound variable, such as in the rule for beta reduction. Note that only "dangling" indices (those that do not refer to any abstraction) can be opened. Opening has no effect for terms that are locally closed.

Natural numbers are just an inductive datatype with two constructors: O and S, defined in Coq.Init.Datatypes.

We make several simplifying assumptions in defining *open_rec*. First, we assume that the argument u is locally closed. This assumption simplifies the implementation since we do not need to shift indices in u when passing under a binder. Second, we assume that this function is initially called with index zero and that zero is the only unbound index in the term. This eliminates the need to possibly subtract one in the case of indices.

There is no need to worry about variable capture because bound variables are indices.

`Fixpoint` *open_rec* (*k* : *nat*) (*u* : *trm*) (*t* : *trm*) {`struct` *t*} : *trm* :=

```
  match t with
  | trm_bvar i ⇒ If k = i then u else (trm_bvar i)
  | trm_fvar x ⇒ trm_fvar x
  | trm_abs t1 ⇒ trm_abs (open_rec (S k) u t1)
  | trm_app t1 t2 ⇒ trm_app (open_rec k u t1) (open_rec k u t2)
  end.
```

Many common applications of opening replace index zero with an expression or variable. The following definition provides a convenient shorthand for such uses. Note that the order of arguments is switched relative to the definition above. For example, (*open e x*) can be read as "substitute the variable x for index 0 in e" and "open e with the variable x." Recall that the coercions above let us write x in place of (*fvar x*).

```
Definition open t u := open_rec 0 u t.
```

We define notations for *open_rec* and *open*.

```
Notation "{ k ~> u } t" := (open_rec k u t) (at level 67).
Notation "t ^^ u" := (open t u) (at level 67).
```

We also define a notation for the specialization of *open* to the case where the argument is a free variable. This notation is not needed when *trm_fvar* is declared as a coercion like we do in this tutorial, but it is very handy when we don't want to have such a coercion. (Coercions are very convenient for simple developments, but they can make things very obscur when it comes to scaling up to larger developments.)

```
Notation "t ^ x" := (open t (trm_fvar x)).
```

This next demo shows the operation of *open*. For example, the locally nameless representation of the term (\y. (\x. (y x)) y) is *abs* (*app* (*abs* (*app* 1 0)) 0). To look at the body without the outer abstraction, we need to replace the indices that refer to that abstraction with a name. Therefore, we show that we can open the body of the abstraction above with Y to produce *app* (*abs* (*app* *Y* 0)) *Y*). The tactic *case_if* is used to simplify conditionals.

```
Lemma demo_open :
  open (trm_app (trm_abs (trm_app 1 0)) 0) Y =
       (trm_app (trm_abs (trm_app Y 0)) Y).
Proof.
  unfold open. unfold open_rec. case_if. case_if. case_if. auto.
Qed.
```

1.1.3 Local closure

Recall that *trm* admits terms that contain unbound indices. We say that a term is locally closed, when no indices appearing in it are unbound. The proposition *term e* holds when an expression e is locally closed.

The inductive definition below formalizes local closure such that the resulting induction principle serves as the structural induction principle over (locally closed) expressions. In

particular, unlike induction for type *trm*, there is no cases for bound variables. Thus, the induction principle corresponds more closely to informal practice than the one arising from the definition of pre-terms.

```
Inductive term : trm → Prop :=
  | term_var : ∀ x,
      term (trm_fvar x)
  | term_abs : ∀ L t1,
      (∀ x, x \notin L → term (t1 ^ x)) →
      term (trm_abs t1)
  | term_app : ∀ t1 t2,
      term t1 →
      term t2 →
      term (trm_app t1 t2).
```

For tactics to work well, it is very important that lists of names to avoid, such as L, appear as first argument of the constructors.

1.1.4 Semantics

We now define the semantics of our call-by-value lambda calculus. We define values and small-step reduction. Note the hypotheses which ensure that the relations hold only for locally closed terms.

```
Inductive value : trm → Prop :=
  | value_abs : ∀ t1,
      term (trm_abs t1) → value (trm_abs t1).

Inductive red : trm → trm → Prop :=
  | red_beta : ∀ t1 t2,
      term (trm_abs t1) →
      value t2 →
      red (trm_app (trm_abs t1) t2) (t1 ^^ t2)
  | red_app_1 : ∀ t1 t1' t2,
      term t2 →
      red t1 t1' →
      red (trm_app t1 t2) (trm_app t1' t2)
  | red_app_2 : ∀ t1 t2 t2',
      value t1 →
      red t2 t2' →
      red (trm_app t1 t2) (trm_app t1 t2').
```

We use the notation t –> t' to denote small step reduction.

```
Notation "t –> t'" := (red t t') (at level 68).
```

1.1.5 Environments

Environments are isomorphic to association lists (lists of pairs of keys and values) whose keys are *var*s. To print environments in a pretty way and, in particular, to ensure that new bindings are added to the right of existing environments, we do not use the type *list* directly but instead use an abstract data type called *env*. More precisely, the type *env A* is isomorphic to *list* (*var* × *A*). Environments are defined in the file *LibEnv*.

Here, environments bind *var*s to *typ*s. So, we define *ctx* as a shorthand for *env typ*.

```
Definition ctx := env typ.
```

If E and F are two contexts, then E & F denotes their concatenation. If x is a variable and T is a type, then $x \neg T$ denotes a singleton environment where x is bound to T. In particular, E & $x \neg T$ denotes a context E extended with a binding from x to T. The empty environment is called *empty*.

The function *dom* computes the domain of an environment, returning a finite set of *var*s.

The unary predicate *ok* holds when each atom is bound at most once in an environment. This property is defined inductively.

The ternary predicate *binds* holds when a given binding is present in an environment. More specifically, *binds x T E* holds when the last binding of x binds x to the type T.

1.1.6 Typing

The definition of the typing relation is straightforward. In order to ensure that the relation holds for only well-formed environments, we check in the *typing_var* case that the environment is *ok*. The structure of typing derivations implicitly ensures that the relation holds only for locally closed expressions. Finally, note the use of cofinite quantification in the *typing_abs* case.

```
Reserved Notation "E |= t ~: T" (at level 69).

Inductive typing : ctx → trm → typ → Prop :=
  | typing_var : ∀ E x T,
      ok E →
      binds x T E →
      E |= (trm_fvar x) ~: T
  | typing_abs : ∀ L E U T t1,
      (∀ x, x \notin L →
        (E & x ¬ U) |= t1 ^ x ~: T) →
      E |= (trm_abs t1) ~: (typ_arrow U T)
  | typing_app : ∀ S T E t1 t2,
      E |= t1 ~: (typ_arrow S T) →
      E |= t2 ~: S →
      E |= (trm_app t1 t2) ~: T
```

```
where "E |= t ~: T" := (typing E t T).
```

1.1.7 Statement of theorems

At this point we can state the theorems that we want to prove. Preservation states that if a well-typed term takes a reduction step then it produces another well-typed term. Progress states that a well-typed term is either a value or it can take a step of reduction.

```
Definition preservation_statement := ∀ E t t' T,
  E |= t ~: T →
  t –> t' →
  E |= t' ~: T.

Definition progress_statement := ∀ t T,
  empty |= t ~: T →
    value t
  ∨ ∃ t', t –> t'.
```

We here reach the end the "trusted base". If we got definitions wrong above that point, then we might not be proving what we intend to. If, however, we got the definitions right, then no matter how ugly our proofs might be, if we reach the Qed of the two theorems that we have stated, then we know that our type system is sound.

1.2 Infrastructure

Before we start getting into the proofs, we need to set up a few more things.

- functions definining free variables and substitution,
- tactics to pick fresh names and to handle freshness-related goals,
- a few axioms about the behavior of operations on terms.

We will purposely introduce some axioms, so that we can go straight to the proofs that we are interested in. Once we are finished with the main proofs, we will do a second pass in order to turn the axioms into proper lemmas.

1.2.1 Free variables

The function *fv*, defined below, calculates the set of free variables in an expression. Because we are using locally nameless representation, where bound variables are represented as indices, any name we see is a free variable of a term. In particular, this makes the *trm_abs* case simple.

```
Fixpoint fv (t : trm) {struct t} : vars :=
  match t with
```

```
  | trm_bvar i ⇒ \{}
  | trm_fvar x ⇒ \{x}
  | trm_abs t1 ⇒ (fv t1)
  | trm_app t1 t2 ⇒ (fv t1) \u (fv t2)
  end.
```

1.2.2 Substitution

Substitution replaces a free variable with a term. The definition below is simple for two reasons:

- Because bound variables are represented using indices, there is no need to worry about variable capture.
- We assume that the term being substituted in is locally closed. Thus, there is no need to shift indices when passing under a binder.

```
Fixpoint subst (z : var) (u : trm) (t : trm) {struct t} : trm :=
  match t with
  | trm_bvar i ⇒ trm_bvar i
  | trm_fvar x ⇒ If x = z then u else (trm_fvar x)
  | trm_abs t1 ⇒ trm_abs (subst z u t1)
  | trm_app t1 t2 ⇒ trm_app (subst z u t1) (subst z u t2)
  end.
```

We define a notation for free variable substitution that mimics standard mathematical notation.

```
Notation "[ z ~> u ] t" := (subst z u t) (at level 68).
```

Below is a demo.

```
Parameter Z : var.
Lemma demo_subst1: [Y ~> Z] (trm_abs (trm_app 0 Y)) = (trm_abs (trm_app 0 Z)).
Proof. simpl. case_if. auto. Qed.
```

1.2.3 Tactics

When picking a fresh atom or applying a rule that uses cofinite quantification, choosing a set of atoms to be fresh for can be tedious. In practice, it is simpler to use a tactic to choose the set to be as large as possible.

The first tactic we define, *gather_vars*, is used to collect together all the atoms in the context. It relies on an auxiliary tactic from *LibLN_Tactics*, *gather_vars_with*, which collects together the atoms appearing in objects of a certain type. The argument to *gather_vars_with* is a function that should return the set of vars appearing in its argument.

```
Ltac gather_vars :=
```

```
  let A := gather_vars_with (fun x : vars ⇒ x) in
  let B := gather_vars_with (fun x : var ⇒ \{x}) in
  let C := gather_vars_with (fun x : ctx ⇒ dom x) in
  let D := gather_vars_with (fun x : trm ⇒ fv x) in
  constr:(A \u B \u C \u D).
```

The tactic *pick_fresh_gen L x* creates a new atom fresh from L and called x. Using the tactic *gather_vars*, we can automate the construction of L. The tactic *pick_fresh x* creates a new atom called x that is fresh for "everything" in the context.

```
Ltac pick_fresh x :=
  let L := gather_vars in (pick_fresh_gen L x).
```

The tactic *apply_fresh T* `as` *y* takes a lemma T of the form $\forall$ *L ...*, ($\forall$ *x*, *x \notin L*, *P x*) $\rightarrow$ *...* $\rightarrow$ *Q*. and applies it by instantiating L as the set of variables occuring in the context (L is computed using *gather_vars*). Moreover, for each subgoal of the form $\forall$ *x*, *x \notin L*, *P x* being generated, the tactic automatically introduces the name x as well as the hypothesis *x \notin L*.

```
Tactic Notation "apply_fresh" constr(T) "as" ident(x) :=
  apply_fresh_base T gather_vars x.
```

The tactic *apply_fresh*$\times$ *T* `as` *y* is the same as *apply_fresh T* `as` *y* except that it calls `intuition eauto` subsequently. It is also possible to call *apply_fresh* without specifying the name that should be used.

```
Tactic Notation "apply_fresh" "*" constr(T) "as" ident(x) :=
  apply_fresh T as x; auto×.
Tactic Notation "apply_fresh" constr(T) :=
  apply_fresh_base T gather_vars ltac_no_arg.
Tactic Notation "apply_fresh" "*" constr(T) :=
  apply_fresh T; auto_star.
```

1.2.4 Automation

Automation is crucial for avoiding to have hundreds of subgoals to handle by hand. For the tactics `auto` and `eauto` to be able to derive proof automatically, we need to give explicitely the list of lemmas that the proof search algorithm should try to exploit. The command `Hint Resolve` *lemma* adds a given lemma to the database of proof search. The command `Hint Constructors` *ind* is equivalent to invoking `Hint Resolve` on all of the constructors of the inductive type *ind*. We use `Hint Constructors` for all our inductively-defined predicates.

```
Hint Constructors term value red.
```

1.2.5 Axiomatization of the infrastructure

```
Module AxiomatizedVersion.
```

At the point, we introduce two simple axioms and skip the many uninteresting auxiliary lemmas that would be required to prove them.

The first axiom states that substitution for a variable x commutes with the operation of opening with another variable y.

The second axiom states that the opening of a term t with a term u can be decomposed in two steps: first opening t with a variable x, and second substituting u for x.

```
Axiom subst_open_var : ∀ x y u t,
  y ≠ x → term u →
  ([x ~> u]t) ^ y = [x ~> u] (t ^ y).

Axiom subst_intro : ∀ x t u,
  x \notin (fv t) → term u →
  t ^^ u = [x ~> u](t ^ x).
```

In order to focus our complete attention on the interesting proofs first, we add a meta-axiom to tell Coq that it should admit any subgoal related to well-formedness, i.e., any goal of the form *term t* or *ok E*. We will remove these axioms later on. This meta-axiom takes the form of a hint whose action is *skip*. This hint will be triggered whenever we call `auto`. The *skip* tactics simply admits the current goal.

```
Local Hint Extern 1 (term _) ⇒ skip.
Local Hint Extern 1 (ok _) ⇒ skip.
```

It might also be useful to add an extra meta-axiom, to get rid of all the freshness-related subgoals. We do not need here, though.

```
Hint Extern 1 (_ \notin _) ⇒ skip
```

1.3 Proofs

Weakening states that if an expression is typeable in some environment, then it is typeable in any well-formed extension of that environment. This property is needed to prove the substitution lemma.

As stated below, this lemma is not directly proveable. The natural way to try proving this lemma proceeds by induction on the typing derivation for t.

```
Lemma typing_weaken_0 : ∀ E F t T,
    E |= t ~: T →
    ok (E & F) →
    (E & F) |= t ~: T.
Proof.
  introv Typ. induction Typ; intros Ok; subst.
  apply× typing_var.
  apply_fresh× typing_abs as y. Admitted.
```

We are stuck in the *typing_abs* case because the induction hypothesis *H0* applies only when we weaken the environment at its head. In this case, however, we need to weaken the

environment in the middle; compare the conclusion at the point where we are stuck to the hypothesis H, which comes from the given typing derivation.

We can obtain a more useful induction hypothesis by changing the statement to insert new bindings into the middle of the environment, instead of at the head. However, the proof still gets stuck, as can be seen by examining each of the cases in the proof below.

```
Lemma typing_weaken_2 : ∀ G E F t T,
    (E & G) |= t ~: T →
    ok (E & F & G) →
    (E & F & G) |= t ~: T.
Proof.
  introv Typ.
  gen_eq H: (E & G). gen G.
  induction Typ; intros G EQ Ok; subst.
  apply× typing_var. apply× binds_weaken.
  let L := gather_vars in sets L': L.
  apply (@typing_abs L').
  intros y Fry. subst L'.
  rewrite ← concat_assoc.
  apply H0.
    auto.     rewrite concat_assoc. auto.
    rewrite concat_assoc. auto.
  apply× typing_app.
Qed.
```

Using the tactic *apply_fresh* introduced earlier, as well as the tactic *apply_ih_bind* which is specialized for applying an induction hypothesis up to rewriting of associativity in contexts, we obtain a nice and short proof.

```
Lemma typing_weaken : ∀ G E F t T,
    (E & G) |= t ~: T →
    ok (E & F & G) →
    (E & F & G) |= t ~: T.
Proof.
  introv Typ. gen_eq H: (E & G). gen G.
  induction Typ; intros G EQ Ok; subst.
  apply× typing_var. apply× binds_weaken.
  apply_fresh× typing_abs as y. apply_ih_bind× H0.
  apply× typing_app.
Qed.
```

Proving that typing is preserved by substitution involves very similar techniques. The only non trivial part concerns the case analysis in the variable case. For that, we use the tactics *binds_get* and *binds_cases* which extract information from *binds* hypotheses.

```
Lemma typing_subst_1 : ∀ F E t T z u U,
```

```
  (E & z ¬ U & F) |= t ~: T →
  E |= u ~: U →
  (E & F) |= [z ~> u]t ~: T.
Proof.
  introv Typt Typu. gen_eq G: (E & z ¬ U & F). gen F.
  induction Typt; intros G Equ; subst; simpl subst.
  case_var.      binds_get H0.          lets M: (@typing_weaken empty E G u U).
      do 2 rewrite concat_empty_r in M.
      apply× M.
    binds_cases H0.
      apply× typing_var.          apply× typing_var.     apply_fresh typing_abs as y.
   rewrite× subst_open_var. apply_ih_bind× H0.
  apply× typing_app.
Qed.
```

As we have seen in the proof above, specializing lemmas on empty environments can be quite tedious. Fortunately, the metatheory library includes tactic that greatly helps. Calling *apply_empty lemma* is almost equivalent to calling `apply` (@*lemma empty*) except that it rewrites away the empty environments on the fly. The proof becomes as follows.

```
Lemma typing_subst : ∀ F E t T z u U,
  (E & z ¬ U & F) |= t ~: T →
  E |= u ~: U →
  (E & F) |= [z ~> u]t ~: T.
Proof.
  introv Typt Typu. gen_eq G: (E & z ¬ U & F). gen F.
  induction Typt; intros G Equ; subst; simpl subst.
  case_var.
    binds_get H0. apply_empty× typing_weaken.
    binds_cases H0; apply× typing_var.
  apply_fresh typing_abs as y.
   rewrite× subst_open_var. apply_ih_bind× H0.
  apply× typing_app.
Qed.
```

The proof of preservation appears below. Proof sketch: By induction on the typing derivation for *t*.

- *typing_var* case: Variables don't step.
- *typing_abs* case: Abstractions don't step.
- *typing_app* case: By case analysis on how *t* steps. The *eval_beta* case is interesting, since it follows by the substitution lemma. The others follow directly from the induction hypotheses.

```
Lemma preservation_result : preservation_statement.
Proof.
  introv Typ. gen t'.
  induction Typ; intros t' Red; inversions Red.
  inversions Typ1. pick_fresh x. rewrite× (@subst_intro x).
   apply_empty× typing_subst.
  apply× typing_app.
  apply× typing_app.
Qed.
```

The proof of progress appears below. Proof sketch: By induction on the typing derivation for t.

- *typing_var* case: Can't happen; the empty environment doesn't bind anything.
- *typing_abs* case: Abstractions are values.
- *typing_app* case: Applications reduce. The result follows from an exhaustive case analysis on whether the two components of the application step or are values and the fact that a value must be an abstraction.

```
Lemma progress_result : progress_statement.
Proof.
  introv Typ. gen_eq E: (empty:ctx). lets Typ': Typ.
  induction Typ; intros; subst.
  false× binds_empty_inv.
  left×.
  right. destruct¬ IHTyp1 as [Val1 | [t1' Red1]].
    destruct¬ IHTyp2 as [Val2 | [t2' Red2]].
      inversions Typ1; inversions Val1. ∃× (t0 ^^ t2).
      ∃× (trm_app t1 t2').
    ∃× (trm_app t1' t2).
Qed.

End AxiomatizedVersion.
```

1.4 Removing all axioms

```
Module CompleteVersion.
```

At this point we come back to the infrastructure part and try to prove all remaining axioms and meta-axioms. We will need to re-check all our proofs. This is usually done in-place in the file, however in this tutorial we have copy-pasted all the proofs.

1.4.1 Proving the two axioms

We first set up four lemmas, and then we can prove our two axioms.

The first lemma is a technical auxiliary lemma which do not want and do not need to read.

```
Lemma open_rec_term_core :∀ t j v i u, i ≠ j →
  {j ~> v}t = {i ~> u}({j ~> v}t) → t = {i ~> u}t.
Proof.
  induction t; introv Neq Equ; simpls; inversion× Equ; fequals×.
  case_nat×. case_nat×.
Qed.
```

Substitution on indices is identity on well-formed terms.

```
Lemma open_rec_term : ∀ t u,
  term t → ∀ k, t = {k ~> u}t.
Proof.
  induction 1; intros; simpl; fequals×. unfolds open.
  pick_fresh x. apply× (@open_rec_term_core t1 0 (trm_fvar x)).
Qed.
```

Substitution for a fresh name is identity.

```
Lemma subst_fresh : ∀ x t u,
  x \notin fv t → [x ~> u] t = t.
Proof. intros. induction t; simpls; fequals×. case_var×. Qed.
```

Substitution distributes on the open operation.

```
Lemma subst_open : ∀ x u t1 t2, term u →
  [x ~> u] (t1 ^^ t2) = ([x ~> u]t1) ^^ ([x ~> u]t2).
Proof.
  intros. unfold open. generalize 0.
  induction t1; intros; simpl; fequals×.
  case_nat×. case_var×. apply× open_rec_term.
Qed.
```

Substitution and open_var for distinct names commute.

```
Lemma subst_open_var : ∀ x y u t, y ≠ x → term u →
  ([x ~> u]t) ^ y = [x ~> u] (t ^ y).
Proof. introv Neq Wu. rewrite× subst_open. simpl. case_var×. Qed.
```

Opening up an abstraction of body t with a term u is the same as opening up the abstraction with a fresh name x and then substituting u for x.

```
Lemma subst_intro : ∀ x t u,
  x \notin (fv t) → term u →
  t ^^ u = [x ~> u](t ^ x).
Proof.
```

```
  introv Fr Wu. rewrite× subst_open.
  rewrite× subst_fresh. simpl. case_var×.
Qed.
```

1.4.2 Preservation of local closure

The goal of this section is to set up the appropriate lemmas for proving goals of the form *term t*. First, we defined a predicate capturing that a term t is the body of a locally closed abstraction.

```
Definition body t :=
  ∃ L, ∀ x, x \notin L → term (t ^ x).
```

We then show how to introduce and eliminate *body t*.

```
Lemma term_abs_to_body : ∀ t1,
  term (trm_abs t1) → body t1.
Proof. intros. unfold body. inversion× H. Qed.

Lemma body_to_term_abs : ∀ t1,
  body t1 → term (trm_abs t1).
Proof. intros. inversion× H. Qed.

Hint Resolve term_abs_to_body body_to_term_abs.
```

We prove that terms are stable by substitution

```
Lemma subst_term : ∀ t z u,
  term u → term t → term ([z ~> u]t).
Proof.
  induction 2; simpls×.
  case_var×.
  apply_fresh term_abs. rewrite× subst_open_var.
Qed.

Hint Resolve subst_term.
```

We prove that opening a body with a term gives a term

```
Lemma open_term : ∀ t u,
  body t → term u → term (t ^^ u).
Proof.
  intros. destruct H. pick_fresh y. rewrite× (@subst_intro y).
Qed.

Hint Resolve open_term.
```

1.4.3 Regularity of relations

The last step to set up the infrastructure consists in proving that relations are "regular". For example, a typing relation can hold only if the environment has no duplicated keys and the

term involved is locally-closed.

```
Lemma typing_regular : ∀ E e T,
  typing E e T → ok E ∧ term e.
Proof.
  split; induction× H.
  pick_fresh y. forwards¬ : (H0 y).
Qed.
```

Similarly, the value predicate only holds on locally-closed terms.

```
Lemma value_regular : ∀ e,
  value e → term e.
Proof. induction 1; auto×. Qed.
```

A reduction relation only holds on pairs of locally-closed terms.

```
Lemma red_regular : ∀ e e',
  red e e' → term e ∧ term e'.
Proof. induction 1; auto× value_regular. Qed.
```

The strength of automation comes from the following custom hints. They are easy to set up because the follow a very regular pattern. These hints indicate that to prove a goal of the form *ok E*, it suffices to find in the goal an hypothesis of the form *typing E t T* and to exploit the regularity lemma *typing_regular* to prove the goal. Similarly, properties of the form *term t* can be extracted out of typing or reduction or value judgments.

```
Hint Extern 1 (ok ?E) ⇒
  match goal with
  | H: typing E _ _ ⊢ _ ⇒ apply (proj1 (typing_regular H))
  end.

Hint Extern 1 (term ?t) ⇒
  match goal with
  | H: typing _ t _ ⊢ _ ⇒ apply (proj2 (typing_regular H))
  | H: red t _ ⊢ _ ⇒ apply (proj1 (red_regular H))
  | H: red _ t ⊢ _ ⇒ apply (proj2 (red_regular H))
  | H: value t ⊢ _ ⇒ apply (value_regular H)
  end.
```

1.4.4 Checking that the main proofs still type-check

We conclude our development by showing that, with the appropriate hints being set up, we can recompile our proofs without changing any single character in them.

```
Lemma typing_weaken : ∀ G E F t T,
    (E & G) |= t ~: T →
    ok (E & F & G) →
    (E & F & G) |= t ~: T.
```

```
Proof.
  introv Typ. gen_eq H: (E & G). gen G.
  induction Typ; intros G EQ Ok; subst.
  apply× typing_var. apply× binds_weaken.
  apply_fresh× typing_abs as y. apply_ih_bind× H0.
  apply× typing_app.
Qed.

Lemma typing_subst : ∀ F E t T z u U,
  (E & z ¬ U & F) |= t ~: T →
  E |= u ~: U →
  (E & F) |= [z ~> u]t ~: T.
Proof.
  introv Typt Typu. gen_eq G: (E & z ¬ U & F). gen F.
  induction Typt; intros G Equ; subst; simpl subst.
  case_var.
    binds_get H0. apply_empty× typing_weaken.
    binds_cases H0; apply× typing_var.
  apply_fresh typing_abs as y.
   rewrite× subst_open_var. apply_ih_bind× H0.
  apply× typing_app.
Qed.

Lemma preservation_result : preservation_statement.
Proof.
  introv Typ. gen t'.
  induction Typ; intros t' Red; inversions Red.
  inversions Typ1. pick_fresh x. rewrite× (@subst_intro x).
   apply_empty× typing_subst.
  apply× typing_app.
  apply× typing_app.
Qed.

Lemma progress_result : progress_statement.
Proof.
  introv Typ. gen_eq E: (empty:ctx). lets Typ': Typ.
  induction Typ; intros; subst.
  false× binds_empty_inv.
  left×.
  right. destruct¬ IHTyp1 as [Val1 | [t1' Red1]].
    destruct¬ IHTyp2 as [Val2 | [t2' Red2]].
      inversions Typ1; inversions Val1. ∃× (t0 ^^ t2).
      ∃× (trm_app t1 t2').
    ∃× (trm_app t1' t2).
Qed.
```

`End` *CompleteVersion.*