SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

NORM

NORMALIZATION OF STLC

```
(* Chapter written and maintained by Andrew Tolmach *)
```

This optional chapter is based on chapter 12 of *Types and Programming Languages* (Pierce). It may be useful to look at the two together, as that chapter includes explanations and informal proofs that are not repeated here.

In this chapter, we consider another fundamental theoretical property of the simply typed lambda-calculus: the fact that the evaluation of a well-typed program is guaranteed to halt in a finite number of steps—-i.e., every well-typed term is *normalizable*.

Unlike the type-safety properties we have considered so far, the normalization property does not extend to full-blown programming languages, because these languages nearly always extend the simply typed lambda-calculus with constructs, such as general recursion (see the MoreStlc chapter) or recursive types, that can be used to write nonterminating programs. However, the issue of normalization reappears at the level of *types* when we consider the metatheory of polymorphic versions of the lambda calculus such as System F-omega: in this system, the language of types effectively contains a copy of the simply typed lambda-calculus, and the termination of the typechecking algorithm will hinge on the fact that a "normalization" operation on type expressions is guaranteed to terminate.

Another reason for studying normalization proofs is that they are some of the most beautiful—-and mind-blowing—-mathematics to be found in the type theory literature, often (as here) involving the fundamental proof technique of *logical relations*.

The calculus we shall consider here is the simply typed lambda-calculus over a single base type `bool` and with pairs. We'll give most details of the development for the basic lambda-calculus terms treating `bool` as an uninterpreted base type, and leave the extension to the boolean operators and pairs to the reader. Even for the base calculus, normalization is not entirely trivial to prove, since each reduction of a term can duplicate redexes in subterms.

Exercise: 2 stars (norm_fail)

Where do we fail if we attempt to prove normalization by a straightforward induction on the size of a well-typed term?

```
(* FILL IN HERE *)
```
□

Exercise: 5 stars, recommended (norm)

The best ways to understand an intricate proof like this is are (1) to help fill it in and (2) to extend it. We've left out some parts of the following development, including some proofs of lemmas and the all the cases involving products and conditionals. Fill them in. □

Language

We begin by repeating the relevant language definition, which is similar to those in the MoreStlc chapter, plus supporting results including type preservation and step determinism. (We won't need progress.) You may just wish to skip down to the Normalization section...

Syntax and Operational Semantics

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.Lists.List. Import ListNotations.
Require Import Maps.
Require Import Smallstep.

Hint Constructors multi.

Inductive ty : Type :=
  | TBool : ty
  | TArrow : ty → ty → ty
  | TProd : ty → ty → ty
.

Inductive tm : Type :=
    (* pure STLC *)
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
    (* pairs *)
  | tpair : tm → tm → tm
  | tfst : tm → tm
  | tsnd : tm → tm
    (* booleans *)
  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.
          (* i.e., if t0 then t1 else t2 *)
```

Substitution

```
Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y ⇒ if beq_string x y then s else t
  | tabs y T t1 ⇒
      tabs y T (if beq_string x y then t1 else (subst x s t1))
  | tapp t1 t2 ⇒ tapp (subst x s t1) (subst x s t2)
  | tpair t1 t2 ⇒ tpair (subst x s t1) (subst x s t2)
  | tfst t1 ⇒ tfst (subst x s t1)
  | tsnd t1 ⇒ tsnd (subst x s t1)
  | ttrue ⇒ ttrue
  | tfalse ⇒ tfalse
  | tif t0 t1 t2 ⇒
      tif (subst x s t0) (subst x s t1) (subst x s t2)
  end.

Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Reduction

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T11 t12,
      value (tabs x T11 t12)
  | v_pair : ∀ v1 v2,
      value v1 →
      value v2 →
      value (tpair v1 v2)
  | v_true : value ttrue
  | v_false : value tfalse
.

Hint Constructors value.

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : ∀ x T11 t12 v2,
         value v2 →
         (tapp (tabs x T11 t12) v2) ==> [x:=v2]t12
  | ST_App1 : ∀ t1 t1' t2,
         t1 ==> t1' →
         (tapp t1 t2) ==> (tapp t1' t2)
  | ST_App2 : ∀ v1 t2 t2',
         value v1 →
         t2 ==> t2' →
         (tapp v1 t2) ==> (tapp v1 t2')
  (* pairs *)
  | ST_Pair1 : ∀ t1 t1' t2,
        t1 ==> t1' →
```

```
        (tpair t₁ t₂) ==> (tpair t₁' t₂)
  | ST_Pair2 : ∀ v₁ t₂ t₂',
        value v₁ →
        t₂ ==> t₂' →
        (tpair v₁ t₂) ==> (tpair v₁ t₂')
  | ST_Fst : ∀ t₁ t₁',
        t₁ ==> t₁' →
        (tfst t₁) ==> (tfst t₁')
  | ST_FstPair : ∀ v₁ v₂,
        value v₁ →
        value v₂ →
        (tfst (tpair v₁ v₂)) ==> v₁
  | ST_Snd : ∀ t₁ t₁',
        t₁ ==> t₁' →
        (tsnd t₁) ==> (tsnd t₁')
  | ST_SndPair : ∀ v₁ v₂,
        value v₁ →
        value v₂ →
        (tsnd (tpair v₁ v₂)) ==> v₂
  (* booleans *)
  | ST_IfTrue : ∀ t₁ t₂,
        (tif ttrue t₁ t₂) ==> t₁
  | ST_IfFalse : ∀ t₁ t₂,
        (tif tfalse t₁ t₂) ==> t₂
  | ST_If : ∀ t₀ t₀' t₁ t₂,
        t₀ ==> t₀' →
        (tif t₀ t₁ t₂) ==> (tif t₀' t₁ t₂)

where "t₁ '==>' t₂" := (step t₁ t₂).

Notation multistep := (multi step).
Notation "t₁ '==>*' t₂" := (multistep t₁ t₂) (at level 40).

Hint Constructors step.

Notation step_normal_form := (normal_form step).

Lemma value__normal : ∀ t, value t → step_normal_form t.
 +
```

Typing

```
Definition context := partial_map ty.

Inductive has_type : context → tm → ty → Prop :=
  (* Typing rules for proper terms *)
  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      has_type Gamma (tvar x) T
```

```
  | T_Abs : ∀ Gamma x T11 T12 t12,
      has_type (update Gamma x T11) t12 T12 →
      has_type Gamma (tabs x T11 t12) (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma t1 t2,
      has_type Gamma t1 (TArrow T1 T2) →
      has_type Gamma t2 T1 →
      has_type Gamma (tapp t1 t2) T2
  (* pairs *)
  | T_Pair : ∀ Gamma t1 t2 T1 T2,
      has_type Gamma t1 T1 →
      has_type Gamma t2 T2 →
      has_type Gamma (tpair t1 t2) (TProd T1 T2)
  | T_Fst : ∀ Gamma t T1 T2,
      has_type Gamma t (TProd T1 T2) →
      has_type Gamma (tfst t) T1
  | T_Snd : ∀ Gamma t T1 T2,
      has_type Gamma t (TProd T1 T2) →
      has_type Gamma (tsnd t) T2
  (* booleans *)
  | T_True : ∀ Gamma,
      has_type Gamma ttrue TBool
  | T_False : ∀ Gamma,
      has_type Gamma tfalse TBool
  | T_If : ∀ Gamma t0 t1 t2 T,
      has_type Gamma t0 TBool →
      has_type Gamma t1 T →
      has_type Gamma t2 T →
      has_type Gamma (tif t0 t1 t2) T
.

Hint Constructors has_type.

Hint Extern 2 (has_type _ (tapp _ _) _) ⇒ eapply T_App; auto.
Hint Extern 2 (_ = _) ⇒ compute; reflexivity.
```

Context Invariance

```
Inductive appears_free_in : string → tm → Prop :=
  | afi_var : ∀ x,
      appears_free_in x (tvar x)
  | afi_app1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tapp t1 t2)
  | afi_app2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tapp t1 t2)
  | afi_abs : ∀ x y T11 t12,
        y ≠ x →
        appears_free_in x t12 →
        appears_free_in x (tabs y T11 t12)
  (* pairs *)
```

```
  | afi_pair1 : ∀ x t₁ t₂,
      appears_free_in x t₁ →
      appears_free_in x (tpair t₁ t₂)
  | afi_pair2 : ∀ x t₁ t₂,
      appears_free_in x t₂ →
      appears_free_in x (tpair t₁ t₂)
  | afi_fst : ∀ x t,
      appears_free_in x t →
      appears_free_in x (tfst t)
  | afi_snd : ∀ x t,
      appears_free_in x t →
      appears_free_in x (tsnd t)
  (* booleans *)
  | afi_if₀ : ∀ x t₀ t₁ t₂,
      appears_free_in x t₀ →
      appears_free_in x (tif t₀ t₁ t₂)
  | afi_if₁ : ∀ x t₀ t₁ t₂,
      appears_free_in x t₁ →
      appears_free_in x (tif t₀ t₁ t₂)
  | afi_if₂ : ∀ x t₀ t₁ t₂,
      appears_free_in x t₂ →
      appears_free_in x (tif t₀ t₁ t₂)
.

Hint Constructors appears_free_in.

Definition closed (t:tm) :=
  ∀ x, ¬ appears_free_in x t.

Lemma context_invariance : ∀ Gamma Gamma' t S,
     has_type Gamma t S →
     (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
     has_type Gamma' t S.
```
+

```
Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
```
+

```
Corollary typable_empty__closed : ∀ t T,
    has_type empty t T →
    closed t.
```
+

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (update Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
```

```
Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
```

Determinism

```
Lemma step_deterministic :
   deterministic step.
```

Normalization

Now for the actual normalization proof.

Our goal is to prove that every well-typed term reduces to a normal form. In fact, it turns out to be convenient to prove something slightly stronger, namely that every well-typed term reduces to a *value*. This follows from the weaker property anyway via Progress (why?) but otherwise we don't need Progress, and we didn't bother re-proving it above.

Here's the key definition:

```
Definition halts (t:tm) : Prop := ∃ t', t ==>* t' ∧ value t'.
```

A trivial fact:

```
Lemma value_halts : ∀ v, value v → halts v.
```

The key issue in the normalization proof (as in many proofs by induction) is finding a strong enough induction hypothesis. To this end, we begin by defining, for each type `T`, a set `R_T` of closed terms of type `T`. We will specify these sets using a relation `R` and write `R T t` when `t` is in `R_T`. (The sets `R_T` are sometimes called *saturated sets* or *reducibility candidates*.)

Here is the definition of `R` for the base language:

- `R bool t` iff `t` is a closed term of type `bool` and `t` halts in a value
- `R` $(T_1 \to T_2)$ `t` iff `t` is a closed term of type $T_1 \to T_2$ and `t` halts in a value *and* for any term `s` such that `R` T_1 `s`, we have `R` T_2 `(t s)`.

This definition gives us the strengthened induction hypothesis that we need. Our primary goal is to show that all *programs* —-i.e., all closed terms of base type—-halt. But closed terms of base type can contain subterms of functional type, so we need to know something about these as well. Moreover, it is not enough to know that these subterms halt, because the application of a normalized function to a normalized argument involves a substitution, which may enable more reduction steps. So we need a stronger condition for terms of functional type: not only should they halt themselves, but, when applied to halting arguments, they should yield halting results.

The form of `R` is characteristic of the *logical relations* proof technique. (Since we are just dealing with unary relations here, we could perhaps more properly say *logical properties*.) If we want to prove some property `P` of all closed terms of type `A`, we proceed by proving, by induction on types, that all terms of type `A` *possess* property `P`, all terms of type `A→A` *preserve* property `P`, all terms of type `(A→A)->(A→A)` *preserve the property of preserving* property `P`, and so on. We do this by defining a family of properties, indexed by types. For the base type `A`, the property is just `P`. For functional types, it says that the function should map values satisfying the property at the input type to values satisfying the property at the output type.

When we come to formalize the definition of `R` in Coq, we hit a problem. The most obvious formulation would be as a parameterized Inductive proposition like this:

```
Inductive R : ty → tm → Prop :=
| R_bool : ∀ b t, has_type empty t TBool →
                halts t →
                R TBool t
| R_arrow : ∀ T1 T2 t, has_type empty t (TArrow T1 T2) →
                halts t →
                (∀ s, R T1 s → R T2 (tapp t s)) →
                R (TArrow T1 T2) t.
```

Unfortunately, Coq rejects this definition because it violates the *strict positivity requirement* for inductive definitions, which says that the type being defined must not occur to the left of an arrow in the type of a constructor argument. Here, it is the third argument to `R_arrow`, namely `(∀s, R T1 s → R TS (tapp t s))`, and specifically the `R T1 s` part, that violates this rule. (The outermost arrows separating the constructor arguments don't count when applying this rule; otherwise we could never have genuinely inductive properties at all!) The reason for the rule is that types defined with non-positive recursion can be used to build non-terminating functions, which as we know would be a disaster for Coq's logical soundness. Even though the relation we want in this case might be perfectly innocent, Coq still rejects it because it fails the positivity test.

Fortunately, it turns out that we *can* define `R` using a `Fixpoint`:

```
Fixpoint R (T:ty) (t:tm) {struct T} : Prop :=
  has_type empty t T ∧ halts t ∧
  (match T with
   | TBool ⇒ True
   | TArrow T1 T2 ⇒ (∀ s, R T1 s → R T2 (tapp t s))

   (* ... edit the next line when dealing with products *)
```

```
    | TProd T1 T2 ⇒ False
    end).
```

As immediate consequences of this definition, we have that every element of every set `R_T` halts in a value and is closed with type `t` :

```
Lemma R_halts : ∀ {T} {t}, R T t → halts t.
```

+

```
Lemma R_typable_empty : ∀ {T} {t}, R T t → has_type empty t T.
```

+

Now we proceed to show the main result, which is that every well-typed term of type `T` is an element of `R_T`. Together with `R_halts`, that will show that every well-typed term halts in a value.

Membership in `R_T` Is Invariant Under Reduction

We start with a preliminary lemma that shows a kind of strong preservation property, namely that membership in `R_T` is *invariant* under reduction. We will need this property in both directions, i.e., both to show that a term in `R_T` stays in `R_T` when it takes a forward step, and to show that any term that ends up in `R_T` after a step must have been in `R_T` to begin with.

First of all, an easy preliminary lemma. Note that in the forward direction the proof depends on the fact that our language is determinstic. This lemma might still be true for nondeterministic languages, but the proof would be harder!

```
Lemma step_preserves_halting : ∀ t t', (t ==> t') → (halts t ↔
halts t').
```

+

Now the main lemma, which comes in two parts, one for each direction. Each proceeds by induction on the structure of the type `T`. In fact, this is where we make fundamental use of the structure of types.

One requirement for staying in `R_T` is to stay in type `T`. In the forward direction, we get this from ordinary type Preservation.

```
Lemma step_preserves_R : ∀ T t t', (t ==> t') → R T t → R T t'.
```

+

The generalization to multiple steps is trivial:

```
Lemma multistep_preserves_R : ∀ T t t',
  (t ==>* t') → R T t → R T t'.
```

+

In the reverse direction, we must add the fact that `t` has type `T` before stepping as an additional hypothesis.

```
Lemma step_preserves_R' : ∀ T t t',
  has_type empty t T → (t ==> t') → R T t' → R T t.
```

+

```
Lemma multistep_preserves_R' : ∀ T t t',
  has_type empty t T → (t ==>* t') → R T t' → R T t.
```

+

Closed Instances of Terms of Type `t` Belong to `R_T`

Now we proceed to show that every term of type `T` belongs to `R_T`. Here, the induction will be on typing derivations (it would be surprising to see a proof about well-typed terms that did not somewhere involve induction on typing derivations!). The only technical difficulty here is in dealing with the abstraction case. Since we are arguing by induction, the demonstration that a term `tabs x` T_1 t_2 belongs to `R_(`T_1→T_2`)` should involve applying the induction hypothesis to show that t_2 belongs to `R_(`T_2`)`. But `R_(`T_2`)` is defined to be a set of *closed* terms, while t_2 may contain `x` free, so this does not make sense.

This problem is resolved by using a standard trick to suitably generalize the induction hypothesis: instead of proving a statement involving a closed term, we generalize it to cover all closed *instances* of an open term `t`. Informally, the statement of the lemma will look like this:

If x_1:T_1,..xn:Tn |- t : T and v_1,...,vn are values such that R T_1 v_1, R T_2 v_2, ..., R Tn vn, then R T ([x_1:=v_1][x_2:=v_2]...[xn:=vn]t).

The proof will proceed by induction on the typing derivation x_1:T_1,..xn:Tn |- t : T; the most interesting case will be the one for abstraction.

Multisubstitutions, Multi-Extensions, and Instantiations

However, before we can proceed to formalize the statement and proof of the lemma, we'll need to build some (rather tedious) machinery to deal with the fact that we are performing *multiple* substitutions on term `t` and *multiple* extensions of the typing context. In particular, we must be precise about the order in which the substitutions occur and how they act on each other. Often these details are simply elided in informal paper proofs, but of course Coq won't let us do that. Since here we are substituting closed terms, we don't need to worry about how one substitution might affect the term put in place by another. But we still do need to worry about the *order* of substitutions, because it is quite possible for the same identifier to appear multiple times among the x_1,...xn with different associated `vi` and `Ti`.

To make everything precise, we will assume that environments are extended from left to right, and multiple substitutions are performed from right to left. To see that this is consistent, suppose we have an environment written as `...,y:bool,...,y:nat,...` and a corresponding term substitution written as `...`

[y:=(tbool true)]...[y:=(tnat 3)]...t. Since environments are extended from left to right, the binding y:nat hides the binding y:bool; since substitutions are performed right to left, we do the substitution y:=(tnat 3) first, so that the substitution y:=(tbool true) has no effect. Substitution thus correctly preserves the type of the term.

With these points in mind, the following definitions should make sense.

A *multisubstitution* is the result of applying a list of substitutions, which we call an *environment*.

```
Definition env := list (string * tm).

Fixpoint msubst (ss:env) (t:tm) {struct ss} : tm :=
match ss with
| nil ⇒ t
| ((x,s)::ss') ⇒ msubst ss' ([x:=s]t)
end.
```

We need similar machinery to talk about repeated extension of a typing context using a list of (identifier, type) pairs, which we call a *type assignment*.

```
Definition tass := list (string * ty).

Fixpoint mupdate (Gamma : context) (xts : tass) :=
  match xts with
  | nil ⇒ Gamma
  | ((x,v)::xts') ⇒ update (mupdate Gamma xts') x v
  end.
```

We will need some simple operations that work uniformly on environments and type assigments

```
Fixpoint lookup {X:Set} (k : string) (l : list (string * X))
{struct l}
              : option X :=
  match l with
    | nil ⇒ None
    | (j,x) :: l' ⇒
      if beq_string j k then Some x else lookup k l'
  end.

Fixpoint drop {X:Set} (n:string) (nxs:list (string * X)) {struct
nxs}
            : list (string * X) :=
  match nxs with
    | nil ⇒ nil
    | ((n',x)::nxs') ⇒
        if beq_string n' n then drop n nxs'
        else (n',x)::(drop n nxs')
  end.
```

An *instantiation* combines a type assignment and a value environment with the same domains, where corresponding elements are in R.

```
Inductive instantiation : tass → env → Prop :=
| V_nil :
    instantiation nil nil
| V_cons : ∀ x T v c e,
    value v → R T v →
    instantiation c e →
    instantiation ((x,T)::c) ((x,v)::e).
```

We now proceed to prove various properties of these definitions.

More Substitution Facts

First we need some additional lemmas on (ordinary) substitution.

```
Lemma vacuous_substitution : ∀ t x,
     ¬ appears_free_in x t →
     ∀ t', [x:=t']t = t.
+
```

```
Lemma subst_closed: ∀ t,
     closed t →
     ∀ x t', [x:=t']t = t.
+
```

```
Lemma subst_not_afi : ∀ t x v,
    closed v → ¬ appears_free_in x ([x:=v]t).
+
```

```
Lemma duplicate_subst : ∀ t' x t v,
  closed v → [x:=t]([x:=v]t') = [x:=v]t'.
+
```

```
Lemma swap_subst : ∀ t x x1 v v1,
    x ≠ x1 →
    closed v → closed v1 →
    [x1:=v1]([x:=v]t) = [x:=v]([x1:=v1]t).
+
```

Properties of Multi-Substitutions

```
Lemma msubst_closed: ∀ t, closed t → ∀ ss, msubst ss t = t.
+
```

Closed environments are those that contain only closed terms.

```
Fixpoint closed_env (env:env) {struct env} :=
  match env with
  | nil ⇒ True
  | (x,t)::env' ⇒ closed t ∧ closed_env env'
  end.
```

Next come a series of lemmas charcterizing how `msubst` of closed terms distributes over `subst` and over each term form

```
Lemma subst_msubst: ∀ env x v t, closed v → closed_env env →
    msubst env ([x:=v]t) = [x:=v](msubst (drop x env) t).
```
+

```
Lemma msubst_var: ∀ ss x, closed_env ss →
   msubst ss (tvar x) =
   match lookup x ss with
   | Some t ⇒ t
   | None ⇒ tvar x
  end.
```
+

```
Lemma msubst_abs: ∀ ss x T t,
  msubst ss (tabs x T t) = tabs x T (msubst (drop x ss) t).
```
+

```
Lemma msubst_app : ∀ ss t1 t2, msubst ss (tapp t1 t2) = tapp
(msubst ss t1) (msubst ss t2).
```
+

You'll need similar functions for the other term constructors.

```
(* FILL IN HERE *)
```

Properties of Multi-Extensions

We need to connect the behavior of type assignments with that of their corresponding contexts.

```
Lemma mupdate_lookup : ∀ (c : tass) (x:string),
    lookup x c = (mupdate empty c) x.
```
+

```
Lemma mupdate_drop : ∀ (c: tass) Gamma x x',
      mupdate Gamma (drop x c) x'
    = if beq_string x x' then Gamma x' else mupdate Gamma c x'.
```
+

Properties of Instantiations

These are strightforward.

```
Lemma instantiation_domains_match: ∀ {c} {e},
    instantiation c e →
    ∀ {x} {T},
      lookup x c = Some T → ∃ t, lookup x e = Some t.
```
+

```
Lemma instantiation_env_closed : ∀ c e,
  instantiation c e → closed_env e.
```

+

```
Lemma instantiation_R : ∀ c e,
    instantiation c e →
    ∀ x t T,
      lookup x c = Some T →
      lookup x e = Some t → R T t.
```

+

```
Lemma instantiation_drop : ∀ c env,
    instantiation c env →
    ∀ x, instantiation (drop x c) (drop x env).
```

+

Congruence Lemmas on Multistep

We'll need just a few of these; add them as the demand arises.

```
Lemma multistep_App2 : ∀ v t t',
  value v → (t ==>* t') → (tapp v t) ==>* (tapp v t').
```

+

```
(* FILL IN HERE *)
```

The R Lemma.

We can finally put everything together.

The key lemma about preservation of typing under substitution can be lifted to multi-substitutions:

```
Lemma msubst_preserves_typing : ∀ c e,
     instantiation c e →
     ∀ Gamma t S, has_type (mupdate Gamma c) t S →
     has_type Gamma (msubst e t) S.
```

+

And at long last, the main lemma.

```
Lemma msubst_R : ∀ c env t T,
    has_type (mupdate empty c) t T →
    instantiation c env →
    R T (msubst env t).
```

+

Normalization Theorem

And the final theorem:

```
Theorem normalization : ∀ t T, has_type empty t T → halts t.
```

+