SOFTWARE FOUNDATIONS

VOLUME 2: PROGRAMMING LANGUAGE FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

REFERENCES

TYPING MUTABLE REFERENCES

Up to this point, we have considered a variety of *pure* language features, including functional abstraction, basic types such as numbers and booleans, and structured types such as records and variants. These features form the backbone of most programming languages — including purely functional languages such as Haskell and "mostly functional" languages such as ML, as well as imperative languages such as C and object-oriented languages such as Java, C#, and Scala.

However, most practical languages also include various *impure* features that cannot be described in the simple semantic framework we have used so far. In particular, besides just yielding results, computation in these languages may assign to mutable variables (reference cells, arrays, mutable record fields, etc.); perform input and output to files, displays, or network connections; make non-local transfers of control via exceptions, jumps, or continuations; engage in inter-process synchronization and communication; and so on. In the literature on programming languages, such "side effects" of computation are collectively referred to as *computational effects*.

In this chapter, we'll see how one sort of computational effect — mutable references — can be added to the calculi we have studied. The main extension will be dealing explicitly with a *store* (or *heap*) and *pointers* that name store locations. This extension is fairly straightforward to define; the most interesting part is the refinement we need to make to the statement of the type preservation theorem.

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

Definitions

Pretty much every programming language provides some form of assignment operation that changes the contents of a previously allocated piece of storage. (Coq's internal language Gallina is a rare exception!)

In some languages — notably ML and its relatives — the mechanisms for name-binding and those for assignment are kept separate. We can have a variable `x` whose *value* is the number `5`, or we can have a variable `y` whose value is a *reference* (or *pointer*) to a mutable cell whose current contents is `5`. These are different things, and the difference is visible to the programmer. We can add `x` to another number, but not assign to it. We can use `y` to assign a new value to the cell that it points to (by writing `y:=84`), but we cannot use `y` directly as an argument to an operation like `+`. Instead, we must explicitly *dereference* it, writing `!y` to obtain its current contents.

In most other languages — in particular, in all members of the C family, including Java — *every* variable name refers to a mutable cell, and the operation of dereferencing a variable to obtain its current contents is implicit.

For purposes of formal study, it is useful to keep these mechanisms separate. The development in this chapter will closely follow ML's model. Applying the lessons learned here to C-like languages is a straightforward matter of collapsing some distinctions and rendering some operations such as dereferencing implicit instead of explicit.

Syntax

In this chapter, we study adding mutable references to the simply-typed lambda calculus with natural numbers.

```
Module STLCRef.
```

The basic operations on references are *allocation*, *dereferencing*, and *assignment*.

- To allocate a reference, we use the `ref` operator, providing an initial value for the new cell. For example, `ref 5` creates a new cell containing the value `5`, and reduces to a reference to that cell.
- To read the current value of this cell, we use the dereferencing operator `!`; for example, `!(ref 5)` reduces to `5`.
- To change the value stored in a cell, we use the assignment operator. If `r` is a reference, `r := 7` will store the value `7` in the cell referenced by `r`.

Types

We start with the simply typed lambda calculus over the natural numbers. Besides the base natural number type and arrow types, we need to add two more types to deal

with references. First, we need the *unit type*, which we will use as the result type of an assignment operation. We then add *reference types*.

If `T` is a type, then `Ref T` is the type of references to cells holding values of type `T`.

```
T ::= Nat
    | Unit
    | T → T
    | Ref T
```

```
Inductive ty : Type :=
  | TNat : ty
  | TUnit : ty
  | TArrow : ty → ty → ty
  | TRef : ty → ty.
```

Terms

Besides variables, abstractions, applications, natural-number-related terms, and `unit`, we need four more sorts of terms in order to handle mutable references:

```
t ::= ...              Terms
    | ref t               allocation
    | !t                  dereference
    | t := t              assignment
    | l                   location
```

```
Inductive tm : Type :=
  (* STLC with numbers: *)
  | tvar : string → tm
  | tapp : tm → tm → tm
  | tabs : string → ty → tm → tm
  | tnat : nat → tm
  | tsucc : tm → tm
  | tpred : tm → tm
  | tmult : tm → tm → tm
  | tif0 : tm → tm → tm → tm
  (* New terms: *)
  | tunit : tm
  | tref : tm → tm
  | tderef : tm → tm
  | tassign : tm → tm → tm
  | tloc : nat → tm.
```

Intuitively:

- `ref t` (formally, `tref t`) allocates a new reference cell with the value `t` and reduces to the location of the newly allocated cell;
- `!t` (formally, `tderef t`) reduces to the contents of the cell referenced by `t`;
- t_1 `:=` t_2 (formally, `tassign` t_1 t_2) assigns t_2 to the cell referenced by t_1; and

- `l` (formally, `tloc l`) is a reference to the cell at location `l`. We'll discuss locations later.

In informal examples, we'll also freely use the extensions of the STLC developed in the MoreStlc chapter; however, to keep the proofs small, we won't bother formalizing them again here. (It would be easy to do so, since there are no very interesting interactions between those features and references.)

Typing (Preview)

Informally, the typing rules for allocation, dereferencing, and assignment will look like this:

$$\frac{\texttt{Gamma |- } t_1 \texttt{ : } T_1}{\texttt{Gamma |- ref } t_1 \texttt{ : Ref } T_1} \text{ (T\_Ref)}$$

$$\frac{\texttt{Gamma |- } t_1 \texttt{ : Ref } T_{11}}{\texttt{Gamma |- !}t_1 \texttt{ : } T_{11}} \text{ (T\_Deref)}$$

$$\frac{\texttt{Gamma |- } t_1 \texttt{ : Ref } T_{11} \quad \texttt{Gamma |- } t_2 \texttt{ : } T_{11}}{\texttt{Gamma |- } t_1 \texttt{ := } t_2 \texttt{ : Unit}} \text{ (T\_Assign)}$$

The rule for locations will require a bit more machinery, and this will motivate some changes to the other rules; we'll come back to this later.

Values and Substitution

Besides abstractions and numbers, we have two new types of values: the unit value, and locations.

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_nat : ∀ n,
      value (tnat n)
  | v_unit :
      value tunit
  | v_loc : ∀ l,
      value (tloc l).

Hint Constructors value.
```

Extending substitution to handle the new syntax of terms is straightforward.

```
Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' ⇒
      if beq_string x x' then s else t
  | tapp t1 t2 ⇒
      tapp (subst x s t1) (subst x s t2)
```

```
  | tabs x' T t1 ⇒
      if beq_string x x' then t else tabs x' T (subst x s t1)
  | tnat n ⇒
      t
  | tsucc t1 ⇒
      tsucc (subst x s t1)
  | tpred t1 ⇒
      tpred (subst x s t1)
  | tmult t1 t2 ⇒
      tmult (subst x s t1) (subst x s t2)
  | tif0 t1 t2 t3 ⇒
      tif0 (subst x s t1) (subst x s t2) (subst x s t3)
  | tunit ⇒
      t
  | tref t1 ⇒
      tref (subst x s t1)
  | tderef t1 ⇒
      tderef (subst x s t1)
  | tassign t1 t2 ⇒
      tassign (subst x s t1) (subst x s t2)
  | tloc _ ⇒
      t
  end.

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

Pragmatics

Side Effects and Sequencing

The fact that we've chosen the result of an assignment expression to be the trivial value `unit` allows a nice abbreviation for *sequencing*. For example, we can write

```
r:=succ(!r); !r
```

as an abbreviation for

```
(\x:Unit. !r) (r:=succ(!r)).
```

This has the effect of reducing two expressions in order and returning the value of the second. Restricting the type of the first expression to `Unit` helps the typechecker to catch some silly errors by permitting us to throw away the first value only if it is really guaranteed to be trivial.

Notice that, if the second expression is also an assignment, then the type of the whole sequence will be `Unit`, so we can validly place it to the left of another `;` to build longer sequences of assignments:

```
r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r
```

Formally, we introduce sequencing as a *derived form* `tseq` that expands into an abstraction and an application.

```
Definition tseq t1 t2 :=
  tapp (tabs "x" TUnit t2) t1.
```

References and Aliasing

It is important to bear in mind the difference between the *reference* that is bound to some variable `r` and the *cell* in the store that is pointed to by this reference.

If we make a copy of `r`, for example by binding its value to another variable `s`, what gets copied is only the *reference*, not the contents of the cell itself.

For example, after reducing

```
let r = ref 5 in
let s = r in
s := 82;
(!r)+1
```

the cell referenced by `r` will contain the value `82`, while the result of the whole expression will be `83`. The references `r` and `s` are said to be *aliases* for the same cell.

The possibility of aliasing can make programs with references quite tricky to reason about. For example, the expression

```
r := 5; r := !s
```

assigns `5` to `r` and then immediately overwrites it with `s`'s current value; this has exactly the same effect as the single assignment

```
r := !s
```

unless we happen to do it in a context where `r` and `s` are aliases for the same cell!

Shared State

Of course, aliasing is also a large part of what makes references useful. In particular, it allows us to set up "implicit communication channels" — shared state — between different parts of a program. For example, suppose we define a reference cell and two functions that manipulate its contents:

```
let c = ref 0 in
let incc = \_:Unit. (c := succ (!c); !c) in
let decc = \_:Unit. (c := pred (!c); !c) in
...
```

Note that, since their argument types are `Unit`, the arguments to the abstractions in the definitions of `incc` and `decc` are not providing any useful information to the bodies of these functions (using the wildcard _ as the name of the bound variable is a reminder of this). Instead, their purpose of these abstractions is to "slow down" the execution of the function bodies. Since function abstractions are values, the two `lets` are executed simply by binding these functions to the names `incc` and `decc`, rather than by actually incrementing or decrementing `c`. Later, each call to one of these functions results in its body being executed once and performing the appropriate mutation on `c`. Such functions are often called *thunks*.

In the context of these declarations, calling `incc` results in changes to `c` that can be observed by calling `decc`. For example, if we replace the `...` with `(incc unit; incc unit; decc unit)`, the result of the whole program will be `1`.

Objects

We can go a step further and write a *function* that creates `c`, `incc`, and `decc`, packages `incc` and `decc` together into a record, and returns this record:

```
newcounter =
    \_:Unit.
       let c = ref 0 in
       let incc = \_:Unit. (c := succ (!c); !c) in
       let decc = \_:Unit. (c := pred (!c); !c) in
       {i=incc, d=decc}
```

Now, each time we call `newcounter`, we get a new record of functions that share access to the same storage cell `c`. The caller of `newcounter` can't get at this storage cell directly, but can affect it indirectly by calling the two functions. In other words, we've created a simple form of *object*.

```
let c1 = newcounter unit in
let c2 = newcounter unit in
// Note that we've allocated two separate storage cells now!
let r1 = c1.i unit in
let r2 = c2.i unit in
r2  // yields 1, not 2!
```

Exercise: 1 star, optional (store_draw)

Draw (on paper) the contents of the store at the point in execution where the first two `lets` have finished and the third one is about to begin.

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

□

References to Compound Types

A reference cell need not contain just a number: the primitives we've defined above allow us to create references to values of any type, including functions. For example, we can use references to functions to give an (inefficient) implementation of arrays of numbers, as follows. Write `NatArray` for the type `Ref (Nat→Nat)`.

Recall the `equal` function from the MoreStlc chapter:

```
equal =
  fix
    (\eq:Nat->Nat->Bool.
       \m:Nat. \n:Nat.
         if m=0 then iszero n
```

```
                else if n=0 then false
                else eq (pred m) (pred n))
```

To build a new array, we allocate a reference cell and fill it with a function that, when given an index, always returns `0`.

```
      newarray = \_:Unit. ref (\n:Nat.0)
```

To look up an element of an array, we simply apply the function to the desired index.

```
      lookup = \a:NatArray. \n:Nat. (!a) n
```

The interesting part of the encoding is the `update` function. It takes an array, an index, and a new value to be stored at that index, and does its job by creating (and storing in the reference) a new function that, when it is asked for the value at this very index, returns the new value that was given to `update`, while on all other indices it passes the lookup to the function that was previously stored in the reference.

```
      update = \a:NatArray. \m:Nat. \v:Nat.
                  let oldf = !a in
                  a := (\n:Nat. if equal m n then v else oldf n);
```

References to values containing other references can also be very useful, allowing us to define data structures such as mutable lists and trees.

Exercise: 2 stars, recommended (compact_update)

If we defined `update` more compactly like this

```
      update = \a:NatArray. \m:Nat. \v:Nat.
                  a := (\n:Nat. if equal m n then v else (!a) n)
```

would it behave the same?

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

□

Null References

There is one final significant difference between our references and C-style mutable variables: in C-like languages, variables holding pointers into the heap may sometimes have the value `NULL`. Dereferencing such a "null pointer" is an error, and results either in a clean exception (Java and C#) or in arbitrary and possibly insecure behavior (C and relatives like C++). Null pointers cause significant trouble in C-like languages: the fact that any pointer might be null means that any dereference operation in the program can potentially fail.

Even in ML-like languages, there are occasionally situations where we may or may not have a valid pointer in our hands. Fortunately, there is no need to extend the basic mechanisms of references to represent such situations: the sum types introduced in the MoreStlc chapter already give us what we need.

First, we can use sums to build an analog of the `option` types introduced in the Lists chapter of *Logical Foundations*. Define `Option T` to be an abbreviation for `Unit + T`.

Then a "nullable reference to a `T`" is simply an element of the type `Option (Ref T)`.

Garbage Collection

A last issue that we should mention before we move on with formalizing references is storage *de*-allocation. We have not provided any primitives for freeing reference cells when they are no longer needed. Instead, like many modern languages (including ML and Java) we rely on the run-time system to perform *garbage collection*, automatically identifying and reusing cells that can no longer be reached by the program.

This is *not* just a question of taste in language design: it is extremely difficult to achieve type safety in the presence of an explicit deallocation operation. One reason for this is the familiar *dangling reference* problem: we allocate a cell holding a number, save a reference to it in some data structure, use it for a while, then deallocate it and allocate a new cell holding a boolean, possibly reusing the same storage. Now we can have two names for the same storage cell — one with type `Ref Nat` and the other with type `Ref Bool`.

Exercise: 2 stars (type_safety_violation)

Show how this can lead to a violation of type safety.

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

□

Operational Semantics

Locations

The most subtle aspect of the treatment of references appears when we consider how to formalize their operational behavior. One way to see why is to ask, "What should be the *values* of type `Ref T`?" The crucial observation that we need to take into account is that reducing a `ref` operator should *do* something — namely, allocate some storage — and the result of the operation should be a reference to this storage.

What, then, is a reference?

The run-time store in most programming-language implementations is essentially just a big array of bytes. The run-time system keeps track of which parts of this array are currently in use; when we need to allocate a new reference cell, we allocate a large enough segment from the free region of the store (4 bytes for integer cells, 8 bytes for cells storing `Floats`, etc.), record somewhere that it is being used, and return the index (typically, a 32- or 64-bit integer) of the start of the newly allocated region. These indices are references.

For present purposes, there is no need to be quite so concrete. We can think of the store as an array of *values*, rather than an array of bytes, abstracting away from the different sizes of the run-time representations of different values. A reference, then, is simply an index into the store. (If we like, we can even abstract away from the fact that

these indices are numbers, but for purposes of formalization in Coq it is convenient to use numbers.) We use the word *location* instead of *reference* or *pointer* to emphasize this abstract quality.

Treating locations abstractly in this way will prevent us from modeling the *pointer arithmetic* found in low-level languages such as C. This limitation is intentional. While pointer arithmetic is occasionally very useful, especially for implementing low-level services such as garbage collectors, it cannot be tracked by most type systems: knowing that location `n` in the store contains a `float` doesn't tell us anything useful about the type of location `n+4`. In C, pointer arithmetic is a notorious source of type-safety violations.

Stores

Recall that, in the small-step operational semantics for IMP, the step relation needed to carry along an auxiliary state in addition to the program being executed. In the same way, once we have added reference cells to the STLC, our step relation must carry along a store to keep track of the contents of reference cells.

We could re-use the same functional representation we used for states in IMP, but for carrying out the proofs in this chapter it is actually more convenient to represent a store simply as a *list* of values. (The reason we didn't use this representation before is that, in IMP, a program could modify any location at any time, so states had to be ready to map *any* variable to a value. However, in the STLC with references, the only way to create a reference cell is with `tref` t_1, which puts the value of t_1 in a new reference cell and reduces to the location of the newly created reference cell. When reducing such an expression, we can just add a new reference cell to the end of the list representing the store.)

```
Definition store := list tm.
```

We use `store_lookup n st` to retrieve the value of the reference cell at location `n` in the store `st`. Note that we must give a default value to `nth` in case we try looking up an index which is too large. (In fact, we will never actually do this, but proving that we don't will require a bit of work.)

```
Definition store_lookup (n:nat) (st:store) :=
  nth n st tunit.
```

To update the store, we use the `replace` function, which replaces the contents of a cell at a particular index.

```
Fixpoint replace {A:Type} (n:nat) (x:A) (l:list A) : list A :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒
    match n with
    | O ⇒ x :: t
    | S n' ⇒ h :: replace n' x t
```

```
    end
  end.
```

As might be expected, we will also need some technical lemmas about `replace`; they are straightforward to prove.

```
Lemma replace_nil : ∀ A n (x:A),
  replace n x nil = nil.
+

Lemma length_replace : ∀ A n x (l:list A),
  length (replace n x l) = length l.
+

Lemma lookup_replace_eq : ∀ l t st,
  l < length st →
  store_lookup l (replace l t st) = t.
+

Lemma lookup_replace_neq : ∀ l1 l2 t st,
  l1 ≠ l2 →
  store_lookup l1 (replace l2 t st) = store_lookup l1 st.
+
```

Reduction

Next, we need to extend the operational semantics to take stores into account. Since the result of reducing an expression will in general depend on the contents of the store in which it is reduced, the evaluation rules should take not just a term but also a store as argument. Furthermore, since the reduction of a term can cause side effects on the store, and these may affect the reduction of other terms in the future, the reduction rules need to return a new store. Thus, the shape of the single-step reduction relation needs to change from `t ==> t'` to `t / st ==> t' / st'`, where `st` and `st'` are the starting and ending states of the store.

To carry through this change, we first need to augment all of our existing reduction rules with stores:

$$\frac{\texttt{value } v_2}{\texttt{(\textbackslash x:T.t12) } v_2 \texttt{ / st ==> [x:=}v_2\texttt{]}t_{12} \texttt{ / st}} \quad \text{(ST\_AppAbs)}$$

$$\frac{t_1 \texttt{ / st ==> } t_1' \texttt{ / st'}}{t_1\ t_2 \texttt{ / st ==> } t_1'\ t_2 \texttt{ / st'}} \quad \text{(ST\_App1)}$$

$$\frac{\texttt{value } v_1\ t_2 \texttt{ / st ==> } t_2' \texttt{ / st'}}{v_1\ t_2 \texttt{ / st ==> } v_1\ t_2' \texttt{ / st'}} \quad \text{(ST\_App2)}$$

Note that the first rule here returns the store unchanged, since function application, in itself, has no side effects. The other two rules simply propagate side effects from premise to conclusion.

Now, the result of reducing a `ref` expression will be a fresh location; this is why we included locations in the syntax of terms and in the set of values. It is crucial to note that making this extension to the syntax of terms does not mean that we intend *programmers* to write terms involving explicit, concrete locations: such terms will arise only as intermediate results during reduction. This may seem odd, but it follows naturally from our design decision to represent the result of every reduction step by a modified *term*. If we had chosen a more "machine-like" model, e.g., with an explicit stack to contain values of bound identifiers, then the idea of adding locations to the set of allowed values might seem more obvious.

In terms of this expanded syntax, we can state reduction rules for the new constructs that manipulate locations and the store. First, to reduce a dereferencing expression $!t_1$, we must first reduce t_1 until it becomes a value:

$$\frac{t_1 \;/\; st \Longrightarrow t_1' \;/\; st'}{!t_1 \;/\; st \Longrightarrow !t_1' \;/\; st'} \quad \text{(ST\_Deref)}$$

Once t_1 has finished reducing, we should have an expression of the form $!l$, where l is some location. (A term that attempts to dereference any other sort of value, such as a function or `unit`, is erroneous, as is a term that tries to dereference a location that is larger than the size $|st|$ of the currently allocated store; the reduction rules simply get stuck in this case. The type-safety properties established below assure us that well-typed terms will never misbehave in this way.)

$$\frac{l < |st|}{!(\text{loc } l) \;/\; st \Longrightarrow \text{lookup } l \; st \;/\; st} \quad \text{(ST\_DerefLoc)}$$

Next, to reduce an assignment expression $t_1 := t_2$, we must first reduce t_1 until it becomes a value (a location), and then reduce t_2 until it becomes a value (of any sort):

$$\frac{t_1 \;/\; st \Longrightarrow t_1' \;/\; st'}{t_1 := t_2 \;/\; st \Longrightarrow t_1' := t_2 \;/\; st'} \quad \text{(ST\_Assign1)}$$

$$\frac{t_2 \;/\; st \Longrightarrow t_2' \;/\; st'}{v_1 := t_2 \;/\; st \Longrightarrow v_1 := t_2' \;/\; st'} \quad \text{(ST\_Assign2)}$$

Once we have finished with t_1 and t_2, we have an expression of the form $l := v_2$, which we execute by updating the store to make location l contain v_2:

$$\frac{l < |st|}{\text{loc } l := v_2 \;/\; st \Longrightarrow \text{unit} \;/\; [l := v_2]st} \quad \text{(ST\_Assign)}$$

The notation $[l := v_2]st$ means "the store that maps l to v_2 and maps all other locations to the same thing as st." Note that the term resulting from this reduction step is just `unit`; the interesting result is the updated store.

Finally, to reduct an expression of the form `ref t1`, we first reduce `t1` until it becomes a value:

$$\frac{\texttt{t}_1 \texttt{ / st ==> } \texttt{t}_1\texttt{' / st'}}{\texttt{ref t}_1 \texttt{ / st ==> ref t}_1\texttt{' / st'}} \quad \text{(ST\_Ref)}$$

Then, to reduce the `ref` itself, we choose a fresh location at the end of the current store — i.e., location `|st|` — and yield a new store that extends `st` with the new value `v1`.

$$\frac{}{\texttt{ref v}_1 \texttt{ / st ==> loc |st| / st,v}_1} \quad \text{(ST\_RefValue)}$$

The value resulting from this step is the newly allocated location itself. (Formally, `st,v1` means `st ++ v1::nil` — i.e., to add a new reference cell to the store, we append it to the end.)

Note that these reduction rules do not perform any kind of garbage collection: we simply allow the store to keep growing without bound as reduction proceeds. This does not affect the correctness of the results of reduction (after all, the definition of "garbage" is precisely parts of the store that are no longer reachable and so cannot play any further role in reduction), but it means that a naive implementation of our evaluator might run out of memory where a more sophisticated evaluator would be able to continue by reusing locations whose contents have become garbage.

Here are the rules again, formally:

```
Reserved Notation "t1 '/' st1 '==>' t2 '/' st2"
  (at level 40, st1 at level 39, t2 at level 39).

Import ListNotations.

Inductive step : tm * store → tm * store → Prop :=
  | ST_AppAbs : ∀ x T t12 v2 st,
        value v2 →
        tapp (tabs x T t12) v2 / st ==> [x:=v2]t12 / st
  | ST_App1 : ∀ t1 t1' t2 st st',
        t1 / st ==> t1' / st' →
        tapp t1 t2 / st ==> tapp t1' t2 / st'
  | ST_App2 : ∀ v1 t2 t2' st st',
        value v1 →
        t2 / st ==> t2' / st' →
        tapp v1 t2 / st ==> tapp v1 t2'/ st'
  | ST_SuccNat : ∀ n st,
        tsucc (tnat n) / st ==> tnat (S n) / st
  | ST_Succ : ∀ t1 t1' st st',
        t1 / st ==> t1' / st' →
        tsucc t1 / st ==> tsucc t1' / st'
  | ST_PredNat : ∀ n st,
```

```
      tpred (tnat n) / st ==> tnat (pred n) / st
  | ST_Pred : ∀ t1 t1' st st',
      t1 / st ==> t1' / st' →
      tpred t1 / st ==> tpred t1' / st'
  | ST_MultNats : ∀ n1 n2 st,
      tmult (tnat n1) (tnat n2) / st ==> tnat (mult n1 n2) /
st
  | ST_Mult1 : ∀ t1 t2 t1' st st',
      t1 / st ==> t1' / st' →
      tmult t1 t2 / st ==> tmult t1' t2 / st'
  | ST_Mult2 : ∀ v1 t2 t2' st st',
      value v1 →
      t2 / st ==> t2' / st' →
      tmult v1 t2 / st ==> tmult v1 t2' / st'
  | ST_If0 : ∀ t1 t1' t2 t3 st st',
      t1 / st ==> t1' / st' →
      tif0 t1 t2 t3 / st ==> tif0 t1' t2 t3 / st'
  | ST_If0_Zero : ∀ t2 t3 st,
      tif0 (tnat 0) t2 t3 / st ==> t2 / st
  | ST_If0_Nonzero : ∀ n t2 t3 st,
      tif0 (tnat (S n)) t2 t3 / st ==> t3 / st
  | ST_RefValue : ∀ v1 st,
      value v1 →
      tref v1 / st ==> tloc (length st) / (st ++ v1::nil)
  | ST_Ref : ∀ t1 t1' st st',
      t1 / st ==> t1' / st' →
      tref t1 / st ==> tref t1' / st'
  | ST_DerefLoc : ∀ st l,
      l < length st →
      tderef (tloc l) / st ==> store_lookup l st / st
  | ST_Deref : ∀ t1 t1' st st',
      t1 / st ==> t1' / st' →
      tderef t1 / st ==> tderef t1' / st'
  | ST_Assign : ∀ v2 l st,
      value v2 →
      l < length st →
      tassign (tloc l) v2 / st ==> tunit / replace l v2 st
  | ST_Assign1 : ∀ t1 t1' t2 st st',
      t1 / st ==> t1' / st' →
      tassign t1 t2 / st ==> tassign t1' t2 / st'
  | ST_Assign2 : ∀ v1 t2 t2' st st',
      value v1 →
      t2 / st ==> t2' / st' →
      tassign v1 t2 / st ==> tassign v1 t2' / st'

where "t1 '/' st1 '==>' t2 '/' st2" := (step (t1,st1) (t2,st2)).
```

One slightly ugly point should be noted here: In the `ST_RefValue` rule, we extend the state by writing `st ++ v1::nil` rather than the more natural `st ++ [v1]`. The reason for this is that the notation we've defined for substitution uses square brackets, which clash with the standard library's notation for lists.

```
Hint Constructors step.

Definition multistep := (multi step).
Notation "t1 '/' st '==>*' t2 '/' st'" :=
                (multistep (t1,st) (t2,st'))
                (at level 40, st at level 39, t2 at level 39).
```

Typing

The contexts assigning types to free variables are exactly the same as for the STLC: partial maps from identifiers to types.

```
Definition context := partial_map ty.
```

Store typings

Having extended our syntax and reduction rules to accommodate references, our last job is to write down typing rules for the new constructs (and, of course, to check that these rules are sound!). Naturally, the key question is, "What is the type of a location?"

First of all, notice that this question doesn't arise when typechecking terms that programmers actually write. Concrete location constants arise only in terms that are the intermediate results of reduction; they are not in the language that programmers write. So we only need to determine the type of a location when we're in the middle of a reduction sequence, e.g., trying to apply the progress or preservation lemmas. Thus, even though we normally think of typing as a *static* program property, it makes sense for the typing of locations to depend on the *dynamic* progress of the program too.

As a first try, note that when we reduce a term containing concrete locations, the type of the result depends on the contents of the store that we start with. For example, if we reduce the term `!(loc 1)` in the store `[unit, unit]`, the result is `unit`; if we reduce the same term in the store `[unit, \x:Unit.x]`, the result is `\x:Unit.x`. With respect to the former store, the location `1` has type `Unit`, and with respect to the latter it has type `Unit→Unit`. This observation leads us immediately to a first attempt at a typing rule for locations:

$$\frac{\texttt{Gamma |- lookup l st : T}_1}{\texttt{Gamma |- loc l : Ref T}_1}$$

That is, to find the type of a location `l`, we look up the current contents of `l` in the store and calculate the type $\texttt{T}_1$ of the contents. The type of the location is then `Ref` $\texttt{T}_1$.

Having begun in this way, we need to go a little further to reach a consistent state. In effect, by making the type of a term depend on the store, we have changed the typing relation from a three-place relation (between contexts, terms, and types) to a four-place relation (between contexts, *stores*, terms, and types). Since the store is, intuitively, part of the context in which we calculate the type of a term, let's write this four-place relation with the store to the left of the turnstile: `Gamma; st |- t : T`. Our rule for typing references now has the form

$$\frac{\texttt{Gamma; st |- lookup l st : } T_1}{\texttt{Gamma; st |- loc l : Ref } T_1}$$

and all the rest of the typing rules in the system are extended similarly with stores. (The other rules do not need to do anything interesting with their stores — just pass them from premise to conclusion.)

However, this rule will not quite do. For one thing, typechecking is rather inefficient, since calculating the type of a location `l` involves calculating the type of the current contents `v` of `l`. If `l` appears many times in a term `t`, we will re-calculate the type of `v` many times in the course of constructing a typing derivation for `t`. Worse, if `v` itself contains locations, then we will have to recalculate *their* types each time they appear. Worse yet, the proposed typing rule for locations may not allow us to derive anything at all, if the store contains a *cycle*. For example, there is no finite typing derivation for the location `0` with respect to this store:

```
[\x:Nat. (!(loc 1)) x, \x:Nat. (!(loc 0)) x]
```

Exercise: 2 stars (cyclic_store)

Can you find a term whose reduction will create this particular cyclic store? □

These problems arise from the fact that our proposed typing rule for locations requires us to recalculate the type of a location every time we mention it in a term. But this, intuitively, should not be necessary. After all, when a location is first created, we know the type of the initial value that we are storing into it. Suppose we are willing to enforce the invariant that the type of the value contained in a given location *never changes*; that is, although we may later store other values into this location, those other values will always have the same type as the initial one. In other words, we always have in mind a single, definite type for every location in the store, which is fixed when the location is allocated. Then these intended types can be collected together as a *store typing* — a finite function mapping locations to types.

As with the other type systems we've seen, this conservative typing restriction on allowed updates means that we will rule out as ill-typed some programs that could reduce perfectly well without getting stuck.

Just as we did for stores, we will represent a store type simply as a list of types: the type at index `i` records the type of the values that we expect to be stored in cell `i`.

```
Definition store_ty := list ty.
```

The `store_Tlookup` function retrieves the type at a particular index.

```
Definition store_Tlookup (n:nat) (ST:store_ty) :=
  nth n ST TUnit.
```

Suppose we are given a store typing `ST` describing the store `st` in which some term `t` will be reduced. Then we can use `ST` to calculate the type of the result of `t` without ever looking directly at `st`. For example, if `ST` is `[Unit, Unit→Unit]`, then we can immediately infer that `!(loc 1)` has type `Unit→Unit`. More generally, the typing rule for locations can be reformulated in terms of store typings like this:

$$\frac{\texttt{l < |ST|}}{\texttt{Gamma; ST |- loc l : Ref (lookup l ST)}}$$

That is, as long as `l` is a valid location, we can compute the type of `l` just by looking it up in `ST`. Typing is again a four-place relation, but it is parameterized on a store *typing* rather than a concrete store. The rest of the typing rules are analogously augmented with store typings.

The Typing Relation

We can now formalize the typing relation for the STLC with references. Here, again, are the rules we're adding to the base STLC (with numbers and `Unit`):

$$\frac{\texttt{l < |ST|}}{\texttt{Gamma; ST |- loc l : Ref (lookup l ST)}}\ \text{(T\_Loc)}$$

$$\frac{\texttt{Gamma; ST |- } t_1 \texttt{ : } T_1}{\texttt{Gamma; ST |- ref } t_1 \texttt{ : Ref } T_1}\ \text{(T\_Ref)}$$

$$\frac{\texttt{Gamma; ST |- } t_1 \texttt{ : Ref } T_{11}}{\texttt{Gamma; ST |- !} t_1 \texttt{ : } T_{11}}\ \text{(T\_Deref)}$$

$$\frac{\texttt{Gamma; ST |- } t_1 \texttt{ : Ref } T_{11} \qquad \texttt{Gamma; ST |- } t_2 \texttt{ : } T_{11}}{\texttt{Gamma; ST |- } t_1 \texttt{ := } t_2 \texttt{ : Unit}}\ \text{(T\_Assign)}$$

```
Reserved Notation "Gamma ';' ST '|-' t '∈' T" (at level 40).

Inductive has_type : context → store_ty → tm → ty → Prop :=
  | T_Var : ∀ Gamma ST x T,
      Gamma x = Some T →
      Gamma; ST |- (tvar x) ∈ T
  | T_Abs : ∀ Gamma ST x T11 T12 t12,
      (update Gamma x T11); ST |- t12 ∈ T12 →
      Gamma; ST |- (tabs x T11 t12) ∈ (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma ST t1 t2,
      Gamma; ST |- t1 ∈ (TArrow T1 T2) →
```

```
      Gamma; ST |- t2 ∈ T1 →
      Gamma; ST |- (tapp t1 t2) ∈ T2
  | T_Nat : ∀ Gamma ST n,
      Gamma; ST |- (tnat n) ∈ TNat
  | T_Succ : ∀ Gamma ST t1,
      Gamma; ST |- t1 ∈ TNat →
      Gamma; ST |- (tsucc t1) ∈ TNat
  | T_Pred : ∀ Gamma ST t1,
      Gamma; ST |- t1 ∈ TNat →
      Gamma; ST |- (tpred t1) ∈ TNat
  | T_Mult : ∀ Gamma ST t1 t2,
      Gamma; ST |- t1 ∈ TNat →
      Gamma; ST |- t2 ∈ TNat →
      Gamma; ST |- (tmult t1 t2) ∈ TNat
  | T_If0 : ∀ Gamma ST t1 t2 t3 T,
      Gamma; ST |- t1 ∈ TNat →
      Gamma; ST |- t2 ∈ T →
      Gamma; ST |- t3 ∈ T →
      Gamma; ST |- (tif0 t1 t2 t3) ∈ T
  | T_Unit : ∀ Gamma ST,
      Gamma; ST |- tunit ∈ TUnit
  | T_Loc : ∀ Gamma ST l,
      l < length ST →
      Gamma; ST |- (tloc l) ∈ (TRef (store_Tlookup l ST))
  | T_Ref : ∀ Gamma ST t1 T1,
      Gamma; ST |- t1 ∈ T1 →
      Gamma; ST |- (tref t1) ∈ (TRef T1)
  | T_Deref : ∀ Gamma ST t1 T11,
      Gamma; ST |- t1 ∈ (TRef T11) →
      Gamma; ST |- (tderef t1) ∈ T11
  | T_Assign : ∀ Gamma ST t1 t2 T11,
      Gamma; ST |- t1 ∈ (TRef T11) →
      Gamma; ST |- t2 ∈ T11 →
      Gamma; ST |- (tassign t1 t2) ∈ TUnit

where "Gamma ';' ST '|-' t '∈' T" := (has_type Gamma ST t T).

Hint Constructors has_type.
```

Of course, these typing rules will accurately predict the results of reduction only if the concrete store used during reduction actually conforms to the store typing that we assume for purposes of typechecking. This proviso exactly parallels the situation with free variables in the basic STLC: the substitution lemma promises that, if `Gamma |- t : T`, then we can replace the free variables in `t` with values of the types listed in `Gamma` to obtain a closed term of type `T`, which, by the type preservation theorem will reduce to a final result of type `T` if it yields any result at all. We will see below how to formalize an analogous intuition for stores and store typings.

However, for purposes of typechecking the terms that programmers actually write, we do not need to do anything tricky to guess what store typing we should use. Concrete locations arise only in terms that are the intermediate results of reduction; they are not in the language that programmers write. Thus, we can simply typecheck the programmer's terms with respect to the *empty* store typing. As reduction proceeds and new locations are created, we will always be able to see how to extend the store typing by looking at the type of the initial values being placed in newly allocated cells; this intuition is formalized in the statement of the type preservation theorem below.

Properties

Our final task is to check that standard type safety properties continue to hold for the STLC with references. The progress theorem ("well-typed terms are not stuck") can be stated and proved almost as for the STLC; we just need to add a few straightforward cases to the proof to deal with the new constructs. The preservation theorem is a bit more interesting, so let's look at it first.

Well-Typed Stores

Since we have extended both the reduction relation (with initial and final stores) and the typing relation (with a store typing), we need to change the statement of preservation to include these parameters. But clearly we cannot just add stores and store typings without saying anything about how they are related — i.e., this is wrong:

```
Theorem preservation_wrong1 : ∀ ST T t st t' st',
  empty; ST |- t ∈ T →
  t / st ==> t' / st' →
  empty; ST |- t' ∈ T.
Abort.
```

If we typecheck with respect to some set of assumptions about the types of the values in the store and then reduce with respect to a store that violates these assumptions, the result will be disaster. We say that a store `st` is *well typed* with respect a store typing `ST` if the term at each location `l` in `st` has the type at location `l` in `ST`. Since only closed terms ever get stored in locations (why?), it suffices to type them in the empty context. The following definition of `store_well_typed` formalizes this.

```
Definition store_well_typed (ST:store_ty) (st:store) :=
  length ST = length st ∧
  (∀ l, l < length st →
     empty; ST |- (store_lookup l st) ∈ (store_Tlookup l ST)).
```

Informally, we will write `ST |- st` for `store_well_typed ST st`.

Intuitively, a store `st` is consistent with a store typing `ST` if every value in the store has the type predicted by the store typing. The only subtle point is the fact that, when typing the values in the store, we supply the very same store typing to the typing relation. This allows us to type circular stores like the one we saw above.

Exercise: 2 stars (store_not_unique)

Can you find a store `st`, and two different store typings ST_1 and ST_2 such that both $ST_1 \vdash st$ and $ST_2 \vdash st$?

```
(* FILL IN HERE *)
```
□

We can now state something closer to the desired preservation property:

```
Theorem preservation_wrong2 : ∀ ST T t st t' st',
  empty; ST |- t ∈ T →
  t / st ==> t' / st' →
  store_well_typed ST st →
  empty; ST |- t' ∈ T.
Abort.
```

This statement is fine for all of the reduction rules except the allocation rule `ST_RefValue`. The problem is that this rule yields a store with a larger domain than the initial store, which falsifies the conclusion of the above statement: if `st'` includes a binding for a fresh location `l`, then `l` cannot be in the domain of `ST`, and it will not be the case that `t'` (which definitely mentions `l`) is typable under `ST`.

Extending Store Typings

Evidently, since the store can increase in size during reduction, we need to allow the store typing to grow as well. This motivates the following definition. We say that the store type `ST'` *extends* `ST` if `ST'` is just `ST` with some new types added to the end.

```
Inductive extends : store_ty → store_ty → Prop :=
  | extends_nil : ∀ ST',
      extends ST' nil
  | extends_cons : ∀ x ST' ST,
      extends ST' ST →
      extends (x::ST') (x::ST).

Hint Constructors extends.
```

We'll need a few technical lemmas about extended contexts.

First, looking up a type in an extended store typing yields the same result as in the original:

```
Lemma extends_lookup : ∀ l ST ST',
  l < length ST →
  extends ST' ST →
  store_Tlookup l ST' = store_Tlookup l ST.
+
```

Next, if `ST'` extends `ST`, the length of `ST'` is at least that of `ST`.

```
Lemma length_extends : ∀ l ST ST',
  l < length ST →
```

```
  extends ST' ST →
  l < length ST'.
+
```

Finally, `ST ++ T` extends `ST`, and `extends` is reflexive.

```
Lemma extends_app : ∀ ST T,
  extends (ST ++ T) ST.
+

Lemma extends_refl : ∀ ST,
  extends ST ST.
+
```

Preservation, Finally

We can now give the final, correct statement of the type preservation property:

```
Definition preservation_theorem := ∀ ST t t' T st st',
  empty; ST |- t ∈ T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     empty; ST' |- t' ∈ T ∧
     store_well_typed ST' st').
```

Note that the preservation theorem merely asserts that there is *some* store typing `ST'` extending `ST` (i.e., agreeing with `ST` on the values of all the old locations) such that the new term `t'` is well typed with respect to `ST'`; it does not tell us exactly what `ST'` is. It is intuitively clear, of course, that `ST'` is either `ST` or else exactly `ST ++ T`$_1$`::nil`, where `T`$_1$ is the type of the value `v`$_1$ in the extended store `st ++ v`$_1$`::nil`, but stating this explicitly would complicate the statement of the theorem without actually making it any more useful: the weaker version above is already in the right form (because its conclusion implies its hypothesis) to "turn the crank" repeatedly and conclude that every *sequence* of reduction steps preserves well-typedness. Combining this with the progress property, we obtain the usual guarantee that "well-typed programs never go wrong."

In order to prove this, we'll need a few lemmas, as usual.

Substitution Lemma

First, we need an easy extension of the standard substitution lemma, along with the same machinery about context invariance that we used in the proof of the substitution lemma for the STLC.

```
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)
  | afi_succ : ∀ x t1,
      appears_free_in x t1 →
      appears_free_in x (tsucc t1)
  | afi_pred : ∀ x t1,
      appears_free_in x t1 →
      appears_free_in x (tpred t1)
  | afi_mult1 : ∀ x t1 t2,
      appears_free_in x t1 →
      appears_free_in x (tmult t1 t2)
  | afi_mult2 : ∀ x t1 t2,
      appears_free_in x t2 →
      appears_free_in x (tmult t1 t2)
  | afi_if0_1 : ∀ x t1 t2 t3,
      appears_free_in x t1 →
      appears_free_in x (tif0 t1 t2 t3)
  | afi_if0_2 : ∀ x t1 t2 t3,
      appears_free_in x t2 →
      appears_free_in x (tif0 t1 t2 t3)
  | afi_if0_3 : ∀ x t1 t2 t3,
      appears_free_in x t3 →
      appears_free_in x (tif0 t1 t2 t3)
  | afi_ref : ∀ x t1,
      appears_free_in x t1 → appears_free_in x (tref t1)
  | afi_deref : ∀ x t1,
      appears_free_in x t1 → appears_free_in x (tderef t1)
  | afi_assign1 : ∀ x t1 t2,
      appears_free_in x t1 → appears_free_in x (tassign t1 t2)
  | afi_assign2 : ∀ x t1 t2,
      appears_free_in x t2 → appears_free_in x (tassign t1 t2).

Hint Constructors appears_free_in.

Lemma free_in_context : ∀ x t T Gamma ST,
   appears_free_in x t →
   Gamma; ST |- t ∈ T →
   ∃ T', Gamma x = Some T'.
+

Lemma context_invariance : ∀ Gamma Gamma' ST t T,
  Gamma; ST |- t ∈ T →
  (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
  Gamma'; ST |- t ∈ T.
```

```
+
```

```
Lemma substitution_preserves_typing : ∀ Gamma ST x s S t T,
  empty; ST |- s ∈ S →
  (update Gamma x S); ST |- t ∈ T →
  Gamma; ST |- ([x:=s]t) ∈ T.
```

```
+
```

Assignment Preserves Store Typing

Next, we must show that replacing the contents of a cell in the store with a new value of appropriate type does not change the overall type of the store. (This is needed for the `ST_Assign` rule.)

```
Lemma assign_pres_store_typing : ∀ ST st l t,
  l < length st →
  store_well_typed ST st →
  empty; ST |- t ∈ (store_Tlookup l ST) →
  store_well_typed ST (replace l t st).
```

```
+
```

Weakening for Stores

Finally, we need a lemma on store typings, stating that, if a store typing is extended with a new location, the extended one still allows us to assign the same types to the same terms as the original.

(The lemma is called `store_weakening` because it resembles the "weakening" lemmas found in proof theory, which show that adding a new assumption to some logical theory does not decrease the set of provable theorems.)

```
Lemma store_weakening : ∀ Gamma ST ST' t T,
  extends ST' ST →
  Gamma; ST |- t ∈ T →
  Gamma; ST' |- t ∈ T.
```

```
+
```

We can use the `store_weakening` lemma to prove that if a store is well typed with respect to a store typing, then the store extended with a new term `t` will still be well typed with respect to the store typing extended with `t`'s type.

```
Lemma store_well_typed_app : ∀ ST st t1 T1,
  store_well_typed ST st →
  empty; ST |- t1 ∈ T1 →
  store_well_typed (ST ++ T1::nil) (st ++ t1::nil).
```

```
+
```

Preservation!

Now that we've got everything set up right, the proof of preservation is actually quite straightforward.

Begin with one technical lemma:

```
Lemma nth_eq_last : ∀ A (l:list A) x d,
  nth (length l) (l ++ x::nil) d = x.
+
```

And here, at last, is the preservation theorem and proof:

```
Theorem preservation : ∀ ST t t' T st st',
  empty; ST |- t ∈ T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     empty; ST' |- t' ∈ T ∧
     store_well_typed ST' st').
+
```

Exercise: 3 stars (preservation_informal)

Write a careful informal proof of the preservation theorem, concentrating on the `T_App`, `T_Deref`, `T_Assign`, and `T_Ref` cases.

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

□

Progress

As we've said, progress for this system is pretty easy to prove; the proof is very similar to the proof of progress for the STLC, with a few new cases for the new syntactic constructs.

```
Theorem progress : ∀ ST t T st,
  empty; ST |- t ∈ T →
  store_well_typed ST st →
  (value t ∨ ∃ t', ∃ st', t / st ==> t' / st').
+
```

References and Nontermination

An important fact about the STLC (proved in chapter Norm) is that it is is *normalizing* — that is, every well-typed term can be reduced to a value in a finite number of steps.

What about STLC + references? Surprisingly, adding references causes us to lose the normalization property: there exist well-typed terms in the STLC + references which can continue to reduce forever, without ever reaching a normal form!

How can we construct such a term? The main idea is to make a function which calls itself. We first make a function which calls another function stored in a reference cell; the trick is that we then smuggle in a reference to itself!

```
(\r:Ref (Unit -> Unit).
     r := (\x:Unit.(!r) unit); (!r) unit)
(ref (\x:Unit.unit))
```

First, `ref (\x:Unit.unit)` creates a reference to a cell of type `Unit → Unit`. We then pass this reference as the argument to a function which binds it to the name `r`, and assigns to it the function `\x:Unit.(!r) unit` — that is, the function which ignores its argument and calls the function stored in `r` on the argument `unit`; but of course, that function is itself! To start the divergent loop, we execute the function stored in the cell by evaluating `(!r) unit`.

Here is the divergent term in Coq:

```
Module ExampleVariables.

Open Scope string_scope.

Definition x := "x".
Definition y := "y".
Definition r := "r".
Definition s := "s".

End ExampleVariables.

Module RefsAndNontermination.
Import ExampleVariables.

Definition loop_fun :=
  tabs x TUnit (tapp (tderef (tvar r)) tunit).

Definition loop :=
  tapp
    (tabs r (TRef (TArrow TUnit TUnit))
      (tseq (tassign (tvar r) loop_fun)
            (tapp (tderef (tvar r)) tunit)))
    (tref (tabs x TUnit tunit)).
```

This term is well typed:

```
Lemma loop_typeable : ∃ T, empty; nil |- loop ∈ T.
+
```

To show formally that the term diverges, we first define the `step_closure` of the single-step reduction relation, written `==>+`. This is just like the reflexive step closure of single-step reduction (which we're been writing `==>*`), except that it is not reflexive: `t ==>+ t'` means that `t` can reach `t'` by *one or more* steps of reduction.

```
Inductive step_closure {X:Type} (R: relation X) : X → X → Prop
:=
  | sc_one : ∀ (x y : X),
                R x y → step_closure R x y
  | sc_step : ∀ (x y z : X),
                R x y →
                step_closure R y z →
                step_closure R x z.

Definition multistep1 := (step_closure step).
Notation "t₁ '/' st '==>+' t₂ '/' st'" :=
        (multistep1 (t₁,st) (t₂,st'))
        (at level 40, st at level 39, t₂ at level 39).
```

Now, we can show that the expression `loop` reduces to the expression `!(loc 0) unit` and the size-one store `[r:=(loc 0)]loop_fun`.

As a convenience, we introduce a slight variant of the `normalize` tactic, called `reduce`, which tries solving the goal with `multi_refl` at each step, instead of waiting until the goal can't be reduced any more. Of course, the whole point is that `loop` doesn't normalize, so the old `normalize` tactic would just go into an infinite loop reducing it forever!

```
Ltac print_goal := match goal with |- ?x ⇒ idtac x end.
Ltac reduce :=
    repeat (print_goal; eapply multi_step ;
            [ (eauto 10; fail) | (instantiate; compute)];
            try solve [apply multi_refl]).
```

Next, we use `reduce` to show that `loop` steps to `!(loc 0) unit`, starting from the empty store.

```
Lemma loop_steps_to_loop_fun :
  loop / nil ==>*
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil.
Proof.
  unfold loop.
  reduce.
Qed.
```

Finally, we show that the latter expression reduces in two steps to itself!

```
Lemma loop_fun_step_self :
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil
==>+
  tapp (tderef (tloc 0)) tunit / cons ([r:=tloc 0]loop_fun) nil.
+
```

Exercise: 4 stars (factorial_ref)

Use the above ideas to implement a factorial function in STLC with references. (There is no need to prove formally that it really behaves like the factorial. Just uncomment the example below to make sure it gives the correct result when applied to the argument `4`.)

```
Definition factorial : tm
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Lemma factorial_type : empty; nil |- factorial ∈ (TArrow TNat
TNat).
Proof with eauto.
  (* FILL IN HERE *) Admitted.
```

If your definition is correct, you should be able to just uncomment the example below; the proof should be fully automatic using the `reduce` tactic.

```
(*
Lemma factorial_4 : exists st,
  tapp factorial (tnat 4) / nil ==>* tnat 24 / st.
Proof.
  eexists. unfold factorial. reduce.
Qed.
*)
```

□

Additional Exercises

Exercise: 5 stars, optional (garabage_collector)

Challenge problem: modify our formalization to include an account of garbage collection, and prove that it satisfies whatever nice properties you can think to prove about it.

□

```
End RefsAndNontermination.
End STLCRef.
```