

Formal type theory

Lecture notes 2009

Lauri Alanko

lealanko@cs.helsinki.fi

December 2, 2009

At the core of Coq lies a purely functional language with a really funky type system. Before we delve into the logical aspects of the language, we go through a quick overview of Coq's syntax from the computational perspective.

1 Inductive definitions

In Coq, everything is either a function or data. All data is represented with inductive datatypes. The inductive datatypes of Coq are similar to the datatypes found in e.g. Haskell or ML, but they have more expressive power, as we shall eventually see.

There are no "primitive" data types in Coq. All data types, even numbers, are defined inductively in user-level code.

1.1 Booleans

We begin with the standard boolean type.

```
Inductive bool :=  
  | true : bool  
  | false : bool  
.  
(* ← Note the period! *)
```

Inductive types are created with the `Inductive` command. A Coq source file is composed from a sequence of commands, each of them terminated with a period.

The above command introduces three new variables into the current scope: *bool*, the newly defined type, and *true* and *false*, the two constructors

of the type. We can verify that both of them are of the type *bool* by using the `Check` command:

```
Check true.  
Check false.
```

Now that we have defined the booleans, we can operate on them. Here is the basic negation operation:

```
Definition negb b :=  
  match b with  
  | true => false  
  | false => true  
end. (* ← Note the period once again! *)
```

There are a number of new things here. Firstly, a `Definition` command defines a new variable. Here we define the function *negb* which takes one parameter, *b*. In the body of the function, *b* is matched so that if *b* is *true*, we return *false*, and vice versa. In a `match` expression we must handle all the possible cases, after which the `match` expression must be terminated with the `end` keyword. In simple cases like this one, pattern matching works just like it does in Haskell or ML.

Pattern matching relies on the fact that `Inductive` creates the *smallest* type that has the given constructors. We know not only that *true* and *false* are in *bool*, but also that no other values are in *bool*. Hence, when our function handles both the case that *b* is *true*, and the case that *b* is *false*, it handles every possible *b* such that *b* : *bool*.

Incidentally, in the above definition we did not say that the type of *b* is *bool*. `Coq` can often infer types of variables when we don't give them explicitly. We can use the `Check` command to see the type that `Coq` inferred for *negb*.

```
Check negb.
```

In addition, the `Print` command can be used to show the entire definition of a variable:

```
Print negb.
```

In the following, we often leave out type annotations when they are obvious from the context both to `Coq` and (hopefully) the reader. Whenever you are uncertain, use the `Check` command to see the inferred types.

1.2 Function application

Function application is expressed with the plain juxtaposition syntax that is common in many functional languages: to apply function f to argument a , we simply write $f a$. In many contexts, though, surrounding parentheses are needed: $(f a)$.

Check $(negb true)$.

Notice that the Check command didn't evaluate the expression. In general, Coq doesn't evaluate anything until we explicitly request it. There are good reasons for this: our purpose is ultimately to *reason* about the terms, and evaluation doesn't always make reasoning easier.

We can ask Coq to show us the result of an evaluation with the Eval command:

Eval compute in $(negb true)$.

Eval compute in $(negb false)$.

Eval compute in $(negb (negb false))$.

We can also write multi-parameter functions:

```
Definition andb (b1 : bool) (b2 : bool) : bool :=
  match b1 with
  | false => false
  | true  => b2
  end.
```

Note the type:

Check *andb*.

This actually means $bool \rightarrow (bool \rightarrow bool)$, i.e. a two-parameter function is actually an ordinary function that takes one parameter and returns a new function that takes another parameter.

The parameters in a Definition command are actually just syntactic sugar for a fun-expression, which creates a function that doesn't need to be named.

```
Definition orb : bool → bool → bool :=
  fun b1 =>
    fun b2 =>
      match b1 with
      | false => b2
      | true  => true
      end.
```

Finally, we can match multiple expressions at the same time, and we can also use the wildcard underscore `_` that matches all the remaining cases:

```
Definition xorb (b1 b2 : bool) : bool :=
  match b1, b2 with
  | true, false => true
  | false, true => true
  | _, _ => false
end.
```

2 Natural numbers

The boolean type is finite: there are only two elements, *true* and *false*. In general, we are much more interested in (potentially) infinite structures. The simplest of these is the type of natural numbers.

We use the standard formulation. There is a number *O* (zero), and for every number *n*, there is another number *S n* (the successor of *n*, or $n + 1$). So e.g. the number four would be written as *S (S (S (S O)))*. The corresponding inductive data type is as follows:

```
Module Nat.
Inductive nat :=
  | O : nat
  | S : nat → nat
.
End Nat.
```

We here enclosed the definition in a `Module`. This prevents it from shadowing Coq's standard definition of natural numbers, which is identical to this one, except that it has some special built-in syntactic support: we can use ordinary decimal numerals in expressions and Coq parses them as a repeated application of *S* to *O*. In particular, *0* is a synonym for *O*. In a similar fashion, Coq also prints values of type *nat* with ordinary numerals.

Sometimes Coq's syntactic conveniences can be confusing. They can be disabled with `Set Printing All`, and re-enabled with `Unset Printing All`.

```
Check (S (S (S O))).
Check 7.
Set Printing All.
Check (S (S (S O))).
Check 7.
```

Unset *Printing All*.

In our definition of *nat*, we indicate that *S* is a constructor that takes a single *nat* as an argument by explicitly giving the full type of *S*. This is a bit different from common functional languages like Haskell or ML. In Haskell, we'd write the above as:

```
data Nat = 0 | S Nat
```

There is a good reason for the syntax that Coq uses for inductive definitions, since it can be generalized in ways that the Haskell style cannot. We shall see this later on.

As with booleans, we can operate on natural numbers using simple pattern matching. The following function returns the predecessor of a number, or *O* if the argument is 0:

```
Definition pred n :=  
  match n with  
  | O => O  
  | S m => m  
  end.
```

3 Structural recursion

The number of things we can do on numbers with plain pattern matching is limited. To do anything really interesting, we have to use recursion over the structure of the numbers.

In Coq, recursive functions have to be explicitly declared as such with the `Fixpoint` command. An ordinary function defined with a `Definition` cannot call itself. Apart from the different keyword, the syntax of a recursive definition is outwardly the same.

The following function returns *true* if the argument is even, and *false* if it is odd:

```
Fixpoint evenb n :=  
  match n with  
  | O => true  
  | S m => negb (evenb m)  
  end.
```

We can also define multi-parameter functions such as addition and multiplication:

```

Fixpoint plus n m :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.

```

```

Fixpoint mult n m :=
  match n with
  | O => O
  | S n' => plus m (mult n' m)
  end.

```

Notation " $x + y$ " := (plus x y) : nat_scope.

Notation " $x * y$ " := (mult x y) : nat_scope.

We can check that these indeed do what they're supposed to.

Eval compute in plus 2 3.

Eval compute in mult 2 3.

Comparing two numbers for equality is also straightforward: O and O are equal, and $S n$ and $S m$ are equal exactly when n and m are equal. $S n$ and O are not equal.

```

Fixpoint beq_nat n m :=
  match n, m with
  | O, O => true
  | S n', S m' => beq_nat n' m'
  | -, - => false
  end.

```

There's a subtle but extremely important difference between Coq and ordinary programming languages regarding recursion. In Coq, the recursive call must be applied to a *strictly smaller* argument. That is, in the above definition of *beq_nat*, it's crucial that we pattern match on n to get n' , which is structurally smaller than n , and apply the recursive call to n' .

Suppose we try to find the square root of a number n by searching through different values m , beginning from zero, until $m * m = n$. We would define it like this:

```

Fixpoint sqrt_aux m n :=
  match beq_nat (mult m m) n with
  | true => m
  | false => sqrt_aux (S m) n
  end.

```

Definition $\text{sqrt } n := \text{sqrt_aux } 0 \ n$.

However, Coq won't accept this definition, because $S \ m$, the recursive argument to sqrt_aux is not smaller than m , the original parameter.

We can still implement this function by searching *downwards*, from m to O :

```
Fixpoint sqrt_aux m n :=
  match beq_nat (mult m m) n with
  | true => m
  | false =>
    match m with
    | O => O
    | S m' => sqrt_aux m' n
    end
  end.
```

Definition $\text{sqrt } n := \text{sqrt_aux } n \ n$.

We see that this indeed does the right thing:

Eval compute in $\text{sqrt } 4$.

Eval compute in $\text{sqrt } 25$.

However, we were forced to define our function so that if n is not the square of any natural number, then $\text{sqrt } n$ evaluates to 0:

Eval compute in $\text{sqrt } 17$.

Thus, for *any* argument, our function will return *something*. When given a non-square number, it won't go into an infinite loop like the original version that would be rejected by Coq.

This, in fact, is one of the most fundamental differences between Coq and ordinary programming languages. Coq is a total language: the evaluation of every well-typed term will eventually terminate. There won't be any infinite loops, nor any run-time errors. This is absolutely essential for the logical aspects of Coq to work properly, but it also means that some kinds of programs become harder or even impossible to write.

Require Import *natbool*.

4 Polymorphism

Previously, our $\text{sqrt } n$ function was defined (more or less arbitrarily) to return 0 when n is not the square of any natural number. It's usually more desirable to have a distinct value to represent a "no result" outcome. We can easily define a new type for this purpose:

```
Inductive natoption :=  
  | SomeNat : nat → natoption  
  | NoNat : natoption  
.
```

That is, a *natoption* is either *SomeNat* n where n is some number, or else it is *NoNat*.

However, this is such a useful construct that we will often want to use it with elements of other types in addition to *nat*. This is achieved by parameterizing the definition by the type of the "some" element.

```
Inductive option A :=  
  | Some : A → option A  
  | None : option A  
.
```

This type is also known as *option* in ML, and *Maybe* in Haskell.

At this point it's instructive to look at the type of not only the constructors, but of *option* itself.

Check *option*.

Check *Some*.

Check *None*.

We see that *option* is actually a function, although it is a function between *types*: it takes a type A of elements as a parameter, and returns the type *option* A , whose values are of the form *None* A and *Some* A a , where $a : A$.

The types of *Some* and *None* are also interesting: they take a type A as their first argument, and the types of (the second argument and) the result *depend* on the particular choice of A , in the sense that A occurs in those types. The \forall syntax is used to express a function type with this kind of a dependence. In fact, an ordinary function types written as $A \rightarrow B$ is only syntactic sugar for $\forall _ : A, B$.

4.1 Implicit arguments

Consider the term `Some nat 42`. The argument `nat` is needed in order to specify what the type of the second argument and the result should be. However, `42` by itself is already of type `nat`, so specifying the first argument is redundant. In fact, Coq can infer it:

```
Check (Some _ 42).
```

In a term (and not a pattern), the underscore `_` is a way of telling Coq: “infer this argument”. However, when we want Coq to *always* infer an argument (and we often do), we can make it “implicit” as follows:

```
Implicit Arguments Some [A].
```

```
Implicit Arguments None [A].
```

Now we can just elide the first argument completely and Coq will infer it. When we *do* need to give the first argument explicitly, we can do so by prepending the name of the function with `@`:

```
Check (Some 42).
```

```
Check (Some true).
```

```
Check (@Some nat 42).
```

```
Check (@None nat).
```

From the above discussion, it should be clear that types are not very much different from other kinds of terms in Coq. A function can take a type as a parameter, and a function can be applied to a type, and a type even has its own type, `Type`. Nearly everything that we can do with ordinary terms, we can also do with types, and this is part of what gives Coq its great expressive power.

5 Pairs

Continuing our quick survey on how common functional concepts are realized in Coq, we now turn to pairs. A pair of `A` and `B` is straightforward enough to define: the type has *one* constructor, and that takes *two* arguments, of types `A` and `B`, respectively.

An inductive type with a single constructor represents a kind of a record or structure: it doesn’t encode any information about choices between alternatives, it is just a collection of elements. Coq provides a special syntax for defining inductive types like these:

```
Structure prod A B := pair {  
  fst : A;
```

```
  snd : B (* ← no semicolon at the end allowed *)
}
```

The Structure syntax lets us name the single constructor, *pair*, and it also directly defines simple accessors to the fields of the structure. When we print out their definitions both with pretty-printing enabled and without it, we can see how a structure is just an inductive type. We also see the `let`-syntax that allows easier pattern matching on single-constructor types.

```
Print prod.
Print fst.
Set Printing All.
Print prod.
Print fst.
Unset Printing All.
```

Again, we don't want to give the type arguments explicitly all the time since they can usually be inferred.

```
Implicit Arguments pair [A B].
Implicit Arguments fst [A B].
Implicit Arguments snd [A B].
```

The type of pairs is often called a Cartesian product, hence the name *prod*. We provide a nice infix syntax for both the product type and the pairs themselves. Although the syntax $A * B$ is already used for multiplication, there's no ambiguity, because this new $A * B$ is only used with types.

```
Notation "A * B" := (prod A B) : type_scope.
Notation "( a , b )" := (pair a b).
```

Just for fun, here is a simple function that reverses the elements of a pair. It demonstrates the `let`-syntax, and one further language feature: the parameters in braces $\{A B\}$ are immediately made implicit.

```
Definition swap {A B} (p : A * B) : B * A :=
  let (a, b) := p in (b, a).
```

Let's try it out:

```
Eval compute in swap (42, true).
```

5.1 Sums

As a counterpart to the product types, there are sum types. An $A + B$ is either an A or a B . In Haskell, the equivalent type is known as *Either A B*.

```
Inductive sum A B :=
```

```
| inr : B → sum A B
| inl : A → sum A B
```

Implicit Arguments *inl* [A].

Implicit Arguments *inr* [B].

Notation "A + B" := (sum A B) : type_scope.

Require Import *natbool*.

Require Import *poly*.

6 Constructive logic

The logical aspects of Coq are based on *constructive* logic. In constructive logic, the proof of a sentence should always be constructed from the proofs of its constituents. More precisely:

- a proof of $A \wedge B$ is a proof of A combined with a proof of B
- a proof of $A \vee B$ is either a proof of A or a proof of B
- a proof of $A \rightarrow B$ is process by which, given a proof of A , we can construct a proof of B
- a proof of $\neg A$ is a process by which, if it were ever given a proof of A , we could construct a proof of an impossible proposition
- a proof of $\forall x. P(x)$ is a process by which, given some object a , we can construct a proof of $P(a)$
- a proof of $\exists x. P(x)$ is some object a combined with a proof of $P(a)$

This is different from classical logic. In classical logic:

- $A \vee B$ is equivalent to $\sim(\sim A \wedge \sim B)$, and in particular $A \vee \neg A$ is a theorem
- $A \rightarrow B$ is equivalent to $\neg(A \wedge \sim B)$
- $\exists x. P(x)$ is equivalent to $\sim\text{forall } x. \sim P(x)$

In constructive logic, the implications hold from left to right, but not in the other direction.

Constructive proofs are more informative than classical ones. A constructive proof of $A \vee B$ tells not only that one of A and B holds, but it also tells *which* one. A constructive proof of $\exists x. P(x)$ tells not only that there is some object for which P holds, but it also gives an example of such an object. Sometimes we can't give a constructive proof for a sentence we could prove classically.

The "processes" above should be "effective methods" of deriving one proof from another. In other words, computable functions. This immediately suggests that we can represent proofs as objects of a programming language, and processes as functions. This is the approach taken by Coq.

Once we have functions and other objects that are meant to represent proofs, we still have to answer what they are proofs *of*. We use types to represent propositions, and type checking makes sure that our objects are actually proofs of the propositions they are intended to prove.

Hence, in Coq proofs are just programs, no different from the "ordinary" programs we use for purely computational tasks. The difference is simply in how we interpret the programs and their types.

6.1 The Prop sort

We could in principle take the above at face value and e.g. prove the distributivity of conjunction and disjunction by writing a function with the type $\forall A B C : \text{Type}, A * (B + C) \rightarrow (A * B) + (A * C)$. However, this is not quite ideal, because then it's not obvious that this is intended to be a proof instead of a computational function.

To help make this distinction, Coq has `Prop`, which is a kind of an alternative to `Type`. There are some technical differences between them, but for us the most important one is in intent: if for some type P we have $P : \text{Prop}$, then we know that P is intended to be interpreted as a proposition. In particular, we are only interested in whether there is some term (proof) with type P or not, but we are not very interested in *what* the term is: one proof serves as well as another.

We can now define conjunction and disjunction. These inductive types are exactly similar to *prod* and *sum*, except that deal with Props, not Types.

```
Structure and (A B : Prop) : Prop := conj {
  proj1 : A;
  proj2 : B
}.
Implicit Arguments conj [A B].
```

Notation " $A \wedge B$ " := (*and* $A B$) : *type_scope*.

Inductive *or* ($A B$: Prop) : Prop :=

| *or_introl* : $A \rightarrow \text{or } A B$

| *or_intror* : $B \rightarrow \text{or } A B$

.

Implicit Arguments *or_introl* [$A B$].

Implicit Arguments *or_intror* [$A B$].

Notation " $A \vee B$ " := (*or* $A B$) : *type_scope*.

We also often need the notion of equivalence between propositions, i.e. each implies the other. This is simply a conjunction of two implications.

Definition *iff* ($A B$: Prop) : Prop := $(A \rightarrow B) \wedge (B \rightarrow A)$.

Notation " $A \leftrightarrow B$ " := (*iff* $A B$) (at level 95, no associativity) : *type_scope*.

7 Negation

Recall that the constructive notion of a proof of $\neg P$ is a process that shows that if we had a proof of P , we could construct a proof of a falsehood, an impossible proposition. What is this impossible proposition? We could choose arbitrarily some proposition that we believe to be false, but there is a more principled approach available.

A logic is *consistent* if not all propositions in it are provable. An inconsistent logic would be quite useless, so we have to assume that Coq is consistent (and in fact this has been proven). Hence, it is impossible that we could prove every proposition.

This suggests the following definition of falsehood:

Definition *Absurd* : Prop := $\forall P$: Prop, P .

This would in fact work just fine, but in Coq it is usually preferable to use inductive types when possible. This is just a matter of style and convenience.

What should the inductive type be like? We want it to be one that is *impossible to construct*. Hence, it should have *no constructors*:

Inductive *False* : Prop := .

This is equivalent to the above definition of *Absurd*. From *False*, we can derive any proposition. Recall that when pattern matching a value of inductive type A to produce a term of type B , we have to give a term of type B for every possible constructor of A . Since *False* has zero constructors, we don't need to give any such terms when matching it:

```

Definition ex_falso_quodlibet :  $\forall P : \text{Prop}, \text{False} \rightarrow P :=$ 
  fun P fal  $\Rightarrow$  match fal with
    end.

```

Now that we have a suitable definition of falsehood, we can define a neat syntax for negations.

```

Definition not (P : Prop) : Prop := P  $\rightarrow$  False.

```

```

Notation " $\neg$  P" := (not P) : type_scope.

```

As an example of propositions involving negation, let's consider one of De Morgan's laws: for any propositions P and Q , $\neg(P \vee Q)$ is equivalent with $\neg P \wedge \neg Q$. Recall that equivalence means that each implies the other. Let's handle each direction separately.

To prove $\neg(P \vee Q) \rightarrow \neg P \wedge \neg Q$ informally, the argument is simple. Suppose that $P \vee Q$ is false. Then P can't be true, because if P were true, $P \vee Q$ would be true, and that would lead to contradiction. By a similar argument, Q can't be true either. Hence $\neg P \wedge \neg Q$.

Formally, we need to construct a term with type $\neg(P \vee Q) \rightarrow \neg P \wedge \neg Q$, as the first step, we need a process (function) for proving $\neg P \wedge \neg Q$, when given $\neg(P \vee Q)$.

Let $npvq$ be a proof of $\neg(P \vee Q)$, i.e. $(P \vee Q) \rightarrow \text{False}$. We need to prove the conjunction $\neg P \wedge \neg Q$. We construct such a proof with *conj*. We now need proofs of $\neg P$ and $\neg Q$ as its arguments.

To prove $\neg P$, we let p be a proof of P , and we need to give a proof of *False*. We can get one by applying $npvq$ and giving it a proof of $P \vee Q$. We can get that by applying *or_intror* to the proof of P . Similarly for $\neg Q$, except we use *or_intror*.

We end up with the following function:

```

Definition de_morgan1 P Q :  $\neg(P \vee Q) \rightarrow \neg P \wedge \neg Q :=$ 
  fun npvq  $\Rightarrow$ 
    conj (fun p  $\Rightarrow$  npvq (or_intror p)) (fun q  $\Rightarrow$  npvq (or_intror q)).

```

The other direction is left as an exercise.

There is another De Morgan's law: the equivalence between $\neg P \vee \neg Q$ and $\neg(P \wedge Q)$. Here is a proof from left to right:

```

Definition de_morgan2 P Q : ( $\neg P \vee \neg Q$ )  $\rightarrow$   $\neg(P \wedge Q) :=$ 
  fun npvnq  $\Rightarrow$ 
    fun paq  $\Rightarrow$ 
      let (p, q) := paq
      in match npvnq with
        | or_intror np  $\Rightarrow$  np p

```

| *or_intror* $nq \Rightarrow nq$
end.

However, the other direction cannot be proven in constructive logic.

8 Truth

Every now and then we have need for “just some true proposition”. While in principle we could pick some arbitrary proposition that we have proven, it’s clearer to have a dedicated type for this purpose.

Inductive *True* : Prop := I : *True*.

A proof of *True* is always of the form *I*. It does not tell us anything interesting, but it is trivial to prove.

9 Equality

What does it mean for two objects to be equal? A common definition is the so-called *Leibniz equality*: two objects are equal, if they have exactly the same properties. That is, *a* and *b* are equal, if for any property *P*, we have $P\ a \leftrightarrow P\ b$. It turns out that it is in fact sufficient to have just one-way implication, and the other direction can be derived. (This is left as an exercise.)

Hence, we get the following definition for Leibniz equality:

Definition *leq* {*A*} (*a b* : *A*) : Prop :=
 $\forall P : A \rightarrow \text{Prop}, P\ a \rightarrow P\ b$.

There is something special about this definition. So far, we have had three different kinds of functions:

- functions from terms to terms, e.g. *pred* : *nat* → *nat*
- functions from types to types, e.g. *option* : Type → Type
- functions from types to terms, e.g. *None* : $\forall A : \text{Type}, \text{option } A$

These correspond to the Haskell and ML concepts of ordinary functions, type constructors (polymorphic types), and polymorphic terms. However, *leq* takes terms (of type *A*), and returns a *type* (recall that propositions are just special kinds of types). In its definition, *P* does the same thing. This cannot be done (directly) in traditional programming languages.

This feature of the language, *dependent types* proper, allows us to have types (propositions) that talk about individual terms (objects), and gives Coq the power of predicate logic.

What can we do with Leibniz equality? For one thing, we can prove that it is reflexive, i.e. every object is equal to itself:

```
Definition refl_leq {A} (a : A) : leq a a :=
  fun (P : A → Prop) (pa : P a) => pa.
```

We can use reflexivity to prove *computational* equalities trivially.

```
Definition two_plus_two_equal_four : leq (3 + 5) (2 * 4) := refl_leq 8.
```

This type-checks because the type checker evaluates $3 + 5$, $2 * 4$ and 8 to see that they are all the same. This is one reason why it's so crucial that the evaluation of terms always terminates: types can contain terms, and we need to evaluate them during type checking, and we definitely want type checking to terminate.

As long as we are discussing definite, concrete objects, like above, equality is quite boring. We don't need to prove a theorem to say that $2 + 2 = 4$, since it is trivially true by computation. Equality becomes much more interesting when used in more general contexts, where the terms in the equality contain parameter variables. We can't just compute them away.

As an example, let's consider the sentence $\forall n m, leq n m \rightarrow leq (S n) (S m)$, i.e. successors of equals are equal. To prove this, we use the equality between n and m to show that a property that holds for n (its successor is equal to $S n$ by reflexivity) also holds for m .

```
Definition S_preserves_equality : ∀ n m, leq n m → leq (S n) (S m) :=
  fun n m (nqm : leq n m) =>
    nqm (fun x => leq (S n) (S x)) (refl_leq (S n)).
```

Another way of thinking about Leibniz equality is that it means *substitutability*: if a and b are equal, then whenever we have a proposition of the form $\dots a \dots$, then we also have a proposition of the form $\dots b \dots$, where a has just been substituted with b . We get it by invoking the equality by using a property of the form $\text{fun } x \Rightarrow \dots x \dots$. We did this above.

However, the power of equality is not limited to plain substitutions. Recall that type checking in Coq involves computation. Hence, to show that Q implies R , given $leq a b$, we just need some property P such that $P a$ evaluates to Q , and $P b$ evaluates to R .

We use this technique in the proof of the injectivity of the successor function, below. $S n$ has the property that its predecessor is equal to n . If $S m$ is equal to $S n$, then it has the property as well. Since *by computation* $pred$

$(S\ m)$ evaluates to m , this means that m is equal to n .

```
Definition S_is_injective :  $\forall\ n\ m,\ leq\ (S\ n)\ (S\ m) \rightarrow leq\ n\ m :=$   
  fun n m snqsm  $\Rightarrow$   
    snqsm (fun x  $\Rightarrow leq\ n\ (pred\ x))\ (refl\_leq\ n)$ .
```

9.1 Standard equality

The definition of *leq* above is a perfectly reasonable way to define equality. However, like in the case of *Absurd* previously, it's not the "Coq way". Instead, an inductive type is used. Coq's infrastructure is heavily tied to the standard library definition of equality, so we will henceforth use that instead of defining our own. The standard definition looks approximately like this (in its own module, to avoid shadowing):

```
Module Eq.  
Inductive eq {A} (a : A) : A  $\rightarrow$  Prop :=  
  refl_equal : eq a a.  
End Eq.
```

There is also a standard notation by which *eq a b* can be written as $a = b$.

The definition of *eq* gives us, for each a , the property of "being equal to a ", and states that the only object that has this property is a . Note that here, too, we have a type that depends on terms, though now the type is inductive.

In fact *eq* is equivalent with our definition of *leq* earlier. Given *eq a b*, and some property P such that $P\ a$, we can get $P\ b$. However, the proof of this may be somewhat befuddling:

```
Definition eq_subst {A} (a b : A) (aqb : a = b) P (pa : P a) : P b :=  
  match aqb with  
  | refl_equal  $\Rightarrow pa$   
  end.
```

What happened here? We had $pa : P\ a$, pattern matched on something with just a single zero-argument constructor, and then the type checker accepted our pa as having the type $P\ b$. What happened was that Coq did some deep magic under the hood. Some of it can be seen by printing out the above definition in the form in which Coq ultimately sees it:

```
Print eq_subst.
```

At this point we are on the verge of delving too deeply into the intricacies of Coq's type system. It becomes obvious that as we begin using

standard equality, our proof terms become so hairy that writing them by hand becomes too difficult, and we need another way to develop proofs.

```
Require Import natbool.  
Require Import poly.  
Require Import logic.  
Require Import util.
```

10 Tactics

Recall that in Coq, a proof is represented by a term of its internal programming language. A proposition is represented by a type, and the correctness of a proof is verified by checking that a term has the declared type.

Ultimately, then, proving in Coq is just a matter of programming: in principle, we could do all our proofs by just writing out a proposition as a type, and then writing a term with that type.

However, this tends to be impractical in the real world. Firstly, in order to be able to express complex propositions, Coq's type system is very complicated. This makes writing larger terms with it is very difficult. Second, many of proofs (or parts of proofs) are conceptually trivial, but still require lots of tedious code.

To make proof development more feasible, Coq provides a way to construct proofs interactively using *tactics*, which are semi-automatic directions for filling out parts of a proof. A *proof script* is a list of tactics, which Coq executes to create a complete proof term.

The standard style of proof scripts in Coq resembles the transcript of commands issued in a text adventure game. Every command makes sense in the context it is given in, and once you have solved the game, you can replay the script to solve the game again quickly. However, when read by itself, the list of commands makes no sense.

We are not interested only in creating formal proofs by any means available. We are also interested in understanding general proof techniques and the relationship between informal and formal proofs. Thus we would like our proof scripts to read more like stories, with enough context to help the reader make sense of the structure of the proof. Ideally our proofs or proof scripts will look just like informal proofs, only formulated precisely enough that even a mechanical proof checker can verify their correctness.

With this goal in mind, we shall diverge a little from Coq's standard tactics, and use some custom tactics that make the proof script more legible, although it also makes them a bit more verbose.

10.1 Proving equality by definition: reflexivity

The primitive notion of equality in Coq is *convertibility*: roughly, two terms are convertible if they evaluate to the same term. This technical notion captures the idea of equality *by definition*.

Verifying the convertibility of terms is an essential part of type checking: if Coq thinks that a term t has type $T1$, but type $T2$ is expected, then type checking succeeds only if $T1$ and $T2$ are convertible.

The equality relation in Coq captures the notion of convertibility. Consider the only constructor of the eq type, $refl_equal$:

Check $refl_equal$.

Note that the type of the resulting equality proof is always $a = a$, for some a . So if a and b are convertible, $a = b$ can be proven with $refl_equal$. The idiomatic way to do this in a proof script is to use the reflexivity tactic.

Lemma $two_plus_two_equals_four$: $2 + 2 = 4$.

Proof.

 reflexivity.

Qed.

This is our first example of a proof script. Firstly, we use the `Lemma` command to declare the name of the proof and its type, i.e. the proposition want to prove. This is for the most part similar to a `Definition`, except that we don't give a body of the definition, but instead develop the proof with a script, which is delimited with `Proof` and `Qed`. Coq accepts `Qed` only after it announces that the proof has been completed.

While in the middle of interpreting a proof script, Coq shows us the current *goal*, i.e. the proposition that we currently need to solve. In the example, the proof script consists only of a single use of the reflexivity tactic, which solves the goal, since $2 + 2$ is convertible with 4 .

We can use reflexivity to prove many other trivial equalities that are true by definition:

Lemma $six_perfect$: $1 + 2 + 3 = 1 * 2 * 3$.

Proof.

 reflexivity.

Qed.

Lemma $evenb_three_is_false$: $evenb\ 3 = false$.

Proof.

 reflexivity.

Qed.

Reflexivity proofs on concrete terms like these are mostly useful for checking that the functions that we have defined behave as we would expect. But to get more interesting results, we need to be able to generalize.

10.2 Proving implications and universals: assume (and intros)

Recall that a term of type $A \rightarrow B$ is a function, and such functions can be constructed with terms of the form `fun a : A => b`, where a is a parameter of type A and b is a term of type B , which may be constructed with the use of a . This corresponds to the standard introduction rule of implication, which says that $A \rightarrow B$ can be proven if B can be proven from a hypothesis A .

Recall also that $\forall a : A, B$ is exactly the same as $A \rightarrow B$, except that a may appear inside B .

In a tactic script, we often prove implications and universal quantifications by introducing the parameter as a hypothesis, and then proving the conclusion. We do this with the custom `assume` tactic:

```
Lemma plus_0_l :  $\forall n : nat, 0 + n = n$ .
```

```
Proof.
```

```
  assume n : nat.
```

```
  reflexivity.
```

```
Qed.
```

Here the `assume` begins the proof of a universally quantified sentence “for all natural numbers n ” by giving a name to the arbitrary natural number (here n) and verifying that it has the type we expect. After this, n is in our current *context* and can be used like any other variable.

We can then prove $0 + n = n$ again by reflexivity. Note that even though we cannot compute the *value* of n , $0 + n$ and n are still convertible, because `plus 0 n` always evaluates to n , i.e. $0 + n$ is *by definition* n . On the other hand, what `plus n 0` evaluates to depends on n , so proving it is somewhat more complicated.

Coq’s standard tactic for introductions is called `intros`, which is similar, but doesn’t allow us to give the type of the introduced variable explicitly. We use a custom `assume` tactic which allows this in order to make the scripts more readable.

10.3 Finishing proofs: prove

We now introduce a little bit of extra syntax for structuring the proofs. Later on we shall have proof scripts with a number of subgoals, each of which gets proven separately by a proof script fragment. To better keep track of where the proof of a goal is finished, we adopt the convention that we use `prove` to indicate a *terminating* tactic. The general syntax is `prove goal` using *tactic* where *goal* is an optional annotation to let us write down the current goal to make the script more legible, and *tactic* is the tactic which we expect to prove the goal.

Here is another simple proof using the new convention:

Lemma *mult_0_1* : $\forall n : \text{nat}, 0 * n = 0$.

Proof.

 assume *n* : *nat*.

 prove ($0 * n = 0$) using reflexivity.

Qed.

10.4 Rewriting terms: rewrite

Recall that one way of viewing an equality $a = b$ is to say that in any context, *a* can be substituted by *b*. At the level of proof scripts, this principle can be used with the `rewrite` tactic. If we have a hypothesis *H* of the form $a = b$, then we can use it to replace *a* with *b* in the goal or in some other hypothesis. The syntax for this is `rewrite \rightarrow H` or `rewrite \leftarrow H` depending on whether we wish to replace *a* with *b* or the other way around.

Here is an example of rewriting in use.

Lemma *pred_S_example* : $\forall n m : \text{nat}, n = S m \rightarrow \text{pred } n = m$.

 assume *n m* : *nat*.

 assume *H* : ($n = S m$).

 rewrite \rightarrow *H*.

 prove ($\text{pred } (S m) = m$) using reflexivity.

Qed.

10.5 Case analysis: per cases

So far all of our proofs have been uniform in the sense that the structure of the proof is not dependent on exact form of an introduced variable: $0 + n = n$ by definition, regardless of whether *n* is 0 or some other number.

However, often it happens that we can prove some property $P x$ for any *x*, but the way it is proven depends on *x*. In this case, we need to do *case*

analysis on x , and give a different proof for each case. Remember that proof scripts ultimately generate proof terms, which are just programs. At that level, case analysis corresponds to simple pattern matching on x , with a different proof term at each branch of the match expression.

A common situation where case analysis on x is needed is when we are interested in $f x$, where f is some function that performs pattern matching on x . In this situation, if we want to know anything about the result of $f x$, we must know enough about the structure of x to know which branch the pattern matching in f will take.

As the simplest example, let's consider the proposition $\forall b : \text{bool}, \text{negb} (\text{negb } b) = b$. We know that b must be one of *true* and *false*. If it is *true*, then, by definition, $\text{negb} (\text{negb } b)$ evaluates to $\text{negb } \text{false}$ and ultimately *true*. Similarly, if it is *false*, $\text{negb} (\text{negb } b)$ evaluates to $\text{negb } \text{true}$ and *false*. The following proof script formalizes the above reasoning.

Lemma *negb_involutive* : $\forall b : \text{bool}, \text{negb} (\text{negb } b) = b$.

Proof.

 assume $b : \text{bool}$.

 per cases on b .

 suppose it is *true*.

 prove $(\text{negb} (\text{negb } \text{true}) = \text{true})$ using reflexivity.

 suppose it is *false*.

 prove $(\text{negb} (\text{negb } \text{false}) = \text{false})$ using reflexivity.

Qed.

NOTE: THIS TACTIC DOESN'T WORK WITH MORE COMPLICATED INDUCTIVE DEFINITIONS. USE THE STANDARD `destruct` TACTIC INSTEAD.

Coq's standard tactics make it hard to see which case we are handling at which time, so we use a custom tactic *per cases*, which attempts to simulate the pattern matching syntax and hopefully makes the structure of the proof clearer.

The *per cases on b* generates two *subgoals*, one for each constructor of *bool*. In each subgoal, b has been replaced by a concrete value of type *bool*. The goals must be proven in the order that the constructors were declared in the definition of the inductive type. In this case, first *true*, then *false*.

The custom tactic does some mystical things to the context and the goal for its internal bookkeeping purposes. The only legal thing to do after a *per cases* tactic is to use *suppose it is* to begin handling a new case. The *suppose* tactic checks that we truly are handling the declared case. Once the subgoal has been proven, handling the next case must again be started

with a suppose declaration.

The *bool* type is extraordinarily simple in that none of its constructors take arguments. More commonly, a constructor takes arguments, and by pattern matching we only get to know that e.g. some number n is actually the successor $S\ m$ of *some* other number m . The suppose syntax supports a primitive form of pattern matching and allows us to bind a variable to the number than n is a successor of.

Here is a simple example:

Lemma *plus_1_neq_0* : $\forall n, \text{beq_nat } (n + 1) 0 = \text{false}$.

Proof.

assume $n : \text{nat}$.

per cases on n .

 suppose it is O .

 prove ($\text{beq_nat } (0 + 1) 0 = \text{false}$) using reflexivity.

 suppose it is $(S\ m)$.

 prove ($\text{beq_nat } (S\ m + 1) 0 = \text{false}$) using reflexivity.

Qed.

Why do we need separate cases here, although we are just using reflexivity in each case? The reason is that in order for us to show that $\text{beq_nat } (n + 1) 0$ evaluates to *false*, we must first show that $n + 1$ evaluates to $S\ n'$, for some n' . It does so, but the reason it does so depends on the value of n . If n is 0, then $n + 1$ evaluates to 1, i.e. $S\ O$, the latter argument. If n is $S\ m$, then $n + 1$ evaluates to $S\ (m + 1)$. So we must consider each case separately to verify this.

10.6 Conversions: change

Sometimes we need to explicitly change the goal or some hypothesis into a convertible but syntactically different form. We may want to do this simply because we think the new form is clearer or more relevant to the proof, or in order to expose a subterm which can then be substituted by another with *rewrite*.

The *change* tactic performs this. The goal is converted with *change goal* where *goal* should be convertible with the current goal. A hypothesis can be converted with *change term* in *hyp* where *hyp* is some current hypothesis whose type is convertible with *term*.

Here is an example of its use:

Lemma *foo* : $\forall m\ n, m * m = n \rightarrow (1 + m) * m = m + n$.

 assume $m\ n : \text{nat}$.

```

assume H : (m * m = n).
change (S m * m = m + n).
change (m + (m * m) = m + n).
rewrite → H.
prove (m + n = m + n) using reflexivity.

```

Qed.

Here the first change is included only for illustration and could be left out. However, the second change is crucial: it converts the goal into a form where $m * m$ appears, and only then can we use `rewrite` to substitute it with n .

There are also tactics for performing conversion steps without specifying beforehand the desired result form. These include `red`, `simpl` and `unfold`. Such tactics are good for investigating what a goal or hypothesis is convertible to, but in a final proof script these tactics are uninformative to the reader, so `change`, which is accompanied with the desired result, should often be preferred.

10.7 Induction

Induction is the logical counterpart of the kind of recursion that Coq supports. Recall that in Coq, a recursive function f on natural numbers practically must be designed so that:

- we specify directly what $f\ 0$ returns
- we specify what $f\ (S\ m)$ returns from the value that $f\ m$ returns

When we think of the function f as a proof that for each n some property $P\ n$ holds, the above principle gets the following interpretation:

- we prove directly that $P\ 0$ holds
- we prove that $P\ (S\ m)$ holds from the assumption that $P\ m$ holds

This is the induction principle for natural numbers.

Proving by induction in Coq is very much like proving by cases, except that in an inductive case (for natural numbers, the $S\ m$ -case) we also get and *inductive hypothesis* stating that the proposition we are trying to prove for $S\ m$ already holds for m .

NOTE: THIS TACTIC DOESN'T WORK WITH MORE COMPLICATED INDUCTIVE DEFINITIONS. USE THE STANDARD `induction` TACTIC INSTEAD.

We use a custom tactic similar to `per cases`, but it is now called `per induction`. The inductive hypothesis can be introduced with the `assume` tactic.

Here is a proof that 0 is the right identity of *plus*. Note that we also make use of a conversion with `change` in order to be able to use the inductive hypothesis in rewriting.

Lemma *plus_0_r* : $\forall n, n + 0 = n$.

Proof.

```
  assume n : nat.
  per induction on n.
  suppose it is O.
    prove (0 + 0 = 0) using reflexivity.
  suppose it is (S m).
    assume H : (m + 0 = m).
    change (S (m + 0) = S m).
    rewrite → H.
    prove (S m = S m) using reflexivity.
```

Qed.

11 Untyped booleans and arithmetic (TAPL Chapter 3)

We next develop some theory for the untyped boolean and arithmetic languages. We follow the exposition of TAPL closely, and concentrate here only on the Coq-specific issues in formalizing the definitions and proofs from TAPL.

```
Require Export Logic.
```

```
Require Export util.
```

```
Require Export notations.
```

We use Coq's module system to keep everything specific to a particular language within an inner module, and general definitions at the "top level" of the file.

```
Module untyped_bool1.
```

```
(* Bring in Ott-generated language definitions and also export them out
   from this module. *)
```

```
Require Export untyped_bool_lang.
```

| | | | |
|-----------|-------|--|---|
| $term, t$ | $::=$ | (t) \mathbf{true} \mathbf{false} $\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3$ | M constant true constant false conditional |
|-----------|-------|--|---|

VAL t

| | |
|-------------------------------|-----------|
| $\frac{}{\mathbf{VAL true}}$ | VAL_TRUE |
| $\frac{}{\mathbf{VAL false}}$ | VAL_FALSE |

$t \longrightarrow t'$ Evaluation

| | |
|---|-----------|
| $\frac{}{\mathbf{if true then } t_2 \mathbf{ else } t_3 \longrightarrow t_2}$ | E_IFTRUE |
| $\frac{}{\mathbf{if false then } t_2 \mathbf{ else } t_3 \longrightarrow t_3}$ | E_IFFALSE |
| $\frac{t_1 \longrightarrow t'_1}{\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 \longrightarrow \mathbf{if } t'_1 \mathbf{ then } t_2 \mathbf{ else } t_3}$ | E_IF |

Figure 1: \mathbb{B} , syntax and evaluation rules

| | | | |
|-----------|-------|---|----------------|
| $term, t$ | $::=$ | | terms: |
| | | (t) | M |
| | | true | constant true |
| | | false | constant false |
| | | if t_1 then t_2 else t_3 | conditional |
| | | 0 | constant zero |
| | | succ t | successor |
| | | pred t | predecessor |
| | | iszero t | zero test |

NV t

$$\frac{}{\mathbf{NV} 0} \text{ NV_ZERO}$$

$$\frac{\mathbf{NV} t}{\mathbf{NV}(\mathbf{succ} t)} \text{ NV_SUCC}$$

VAL t

$$\frac{}{\mathbf{VAL} \mathbf{true}} \text{ VAL_TRUE}$$

$$\frac{}{\mathbf{VAL} \mathbf{false}} \text{ VAL_FALSE}$$

$$\frac{\mathbf{NV} t}{\mathbf{VAL} t} \text{ VAL_NV}$$

Figure 2: \mathbb{BN} , syntax

$t \longrightarrow t'$ Evaluation

$$\begin{array}{c}
\frac{}{\mathbf{if\ true\ then\ } t_2 \mathbf{\ else\ } t_3 \longrightarrow t_2} \text{E_IFTRUE} \\
\frac{}{\mathbf{if\ false\ then\ } t_2 \mathbf{\ else\ } t_3 \longrightarrow t_3} \text{E_IFFALSE} \\
\frac{t_1 \longrightarrow t'_1}{\mathbf{if\ } t_1 \mathbf{\ then\ } t_2 \mathbf{\ else\ } t_3 \longrightarrow \mathbf{if\ } t'_1 \mathbf{\ then\ } t_2 \mathbf{\ else\ } t_3} \text{E_IF} \\
\frac{t_1 \longrightarrow t'_1}{\mathbf{succ\ } t_1 \longrightarrow \mathbf{succ\ } t'_1} \text{E_SUCC} \\
\frac{}{\mathbf{pred\ } 0 \longrightarrow 0} \text{E_PREDZERO} \\
\frac{\mathbf{NV\ } t_1}{\mathbf{pred\ (succ\ } t_1 \mathbf{)\ } \longrightarrow t_1} \text{E_PREDSUCC} \\
\frac{t_1 \longrightarrow t'_1}{\mathbf{pred\ } t_1 \longrightarrow \mathbf{pred\ } t'_1} \text{E_PRED} \\
\frac{}{\mathbf{iszero\ } 0 \longrightarrow \mathbf{true}} \text{E_ISZEROZERO} \\
\frac{\mathbf{NV\ } t_1}{\mathbf{iszero\ (succ\ } t_1 \mathbf{)\ } \longrightarrow \mathbf{false}} \text{E_ISZEROSUCC} \\
\frac{t_1 \longrightarrow t'_1}{\mathbf{iszero\ } t_1 \longrightarrow \mathbf{iszero\ } t'_1} \text{E_ISZERO}
\end{array}$$

Figure 3: \mathbb{BN} , evaluation rules

11.1 Determinacy of one-step evaluation (TAPL 3.5.4)

This proof follows the structure of the informal proof in TAPL quite closely.

Lemma *eval_deterministic* :

$$\forall t t' t'', t \longrightarrow t' \rightarrow t \longrightarrow t'' \rightarrow t' = t''.$$

Proof.

`intros t t' t'' tEt' tEt''.`

At this point we encounter a very common and important issue with induction: we need to make sure that the inductive hypothesis is strong enough. In order for the induction to succeed, we want to prove inductively the relatively general sentence $\forall y2, x \longrightarrow y2 \rightarrow y1 = y2$ so that inductive hypotheses are also of this form. If we only try to prove $x \longrightarrow y2 \rightarrow y1 = y2$, or $y1 = y2$, then we will get stuck during the proof.

To make sure that we get an induction hypothesis that is general enough, we specify in the `in` clause of the `induction` tactic the hypotheses which we want to generalize. These should be dependent on either the value or the type of the term we are inducing on.

We provide explicit names for the introduced variables to make sure that they match with those used in the proof in TAPL.

`induction tEt' as [t2 t3 | t2 t3 | t1 t2 t3 t1' t1Et1'] in t'', tEt'' ⊢ *.`

All the evaluation rules are of the form `(if t1 then t2 else t3)%tm → t4`, so we know that x has the form of a conditional. The question is, what is the form of $t1$? That is determined by the exact rule used to derive $x \longrightarrow y1$.

Case `"tEt' = E.IfTrue t2 t3"`.

We are now considering the case that $xRy1$ was formed by `E.IfTrue`, and hence $x = \text{true}\%tm$. We then use `inversion` to show that $xRy2$ must also have been derived by `E.IfTrue`.

`inversion tEt'' as [t2_ t3_ | t2_ t3_ | t1_ t2_ t3_ t1'_].`

`SCase "tEt'' = E.IfTrue t2_t3_".`

When the same rule is used for each case, then trivially the result of the evaluation is the same.

`reflexivity.`

The `E.IfFalse` rule is immediately ruled out by the `inversion` tactic so we don't need to consider it at all.

`SCase "tEt'' = E.If t1_t2_t3_t1'_".`

The E_If rule isn't immediately ruled out because it could in principle also be used *if* there were some rule that evaluated *true* to something else. There is no such rule, of course, so we use inversion on the assumption that there is, and that proves our goal with the *ex falso quodlibet* principle by showing that the assumption is impossible.

inversion $H3$.

Case " $tEt' = E_IfFalse\ t2\ t3$ ".

The $E_IfFalse$ case is handled analogously.

inversion tEt' as $[t2_t3_ | t2_t3_ | t1_t2_t3_t1']$.

SCase " $E_IfFalse$ ".

reflexivity.

SCase " E_If ".

inversion $H3$.

Case " $tEt = E_If\ t1\ t2\ t3\ t1''$ ".

Here begins the interesting bit. We now know that there is some $t1'$ and some derivation of $t1 \rightarrow t1'$. We show that this implies that $t \rightarrow t''$ was also derived with E_If , since *true* and *false* cannot evaluate any further.

inversion tEt' as $[t2_t3_Ht1\ Ht''$
 $| t2_t3_Ht1\ Ht''$
 $| t1_t2_t3_t1''\ Ht1Et1'']$.

SCase " $tEt' = E_IfTrue\ t2_t3_$ ".

If $t \rightarrow t''$ were derived with E_IfTrue , then $t1$ would be *true*. This is clearly impossible.

rewrite $\leftarrow Ht1$ in $t1Et1'$.

inversion $t1Et1'$.

SCase " $tEt' = E_IfFalse\ t2_t3_$ ".

Similarly for $E_IfFalse$.

rewrite $\leftarrow Ht1$ in $t1Et1'$.

inversion $t1Et1'$.

SCase " $tEt' = E_If\ t1_t2_t3_t1''$ ".

At this point we know that both $t \rightarrow t'$ and $t \rightarrow t''$ have been derived by E_If , so we know that $t1 \rightarrow t1'$ and $t1 \rightarrow t1''$ for some $t1'$ and $t1''$. We need to show that these are the same. At this point our generalized inductive hypothesis pays off: we know that for *any* t'' , now $t1 \rightarrow t'' \rightarrow$

$t1' = t''$. We can thus use it with $t1''$ which we got by inversion on $t \longrightarrow t''$ (for our *original* t'').

```
rewrite → (IHt1Et1' t1'' Ht1Et1'').
```

```
reflexivity.
```

Qed.

End *untyped_bool1*.

11.2 Normal forms (TAPL 3.5.6)

We define normal forms relative to a relation, so we can use the same definition with future languages as well.

A normal form can be defined in either of two ways. The version that uses \exists is more natural, but a bit more cumbersome to use. The \forall version is nicer, so we use that.

Definition *normal_form* $\{A : \text{Type}\} (R : \text{relation } A) (t : A) :=$
 $\forall t', \sim(R t t')$.

Definition *normal_form'* $\{A : \text{Type}\} (R : \text{relation } A) (t : A) :=$
 $\sim\text{exists } t', R t t'$.

Lemma *normal_forms_equiv* $\{A\} (R : \text{relation } A) (t : A) :$
 $\text{normal_form } R t \leftrightarrow \text{normal_form}' R t$.

Proof.

```
intros A R t.
```

```
split.
```

```
intros nft [t' tEt'].
```

```
apply nft with t', tEt'.
```

```
intros nft' t' tEt'.
```

```
eapply nft'.
```

```
 $\exists t'$ .
```

```
apply tEt'.
```

Qed.

Module *untyped_bool2*.

Export *untyped_bool1*.

11.3 Every value is a normal form (TAPL 3.5.7)

The proof simply considers all possible values and shows by inversion that no rule can evaluate the values further.

Lemma *value_normal* : $\forall t, \text{val } t \rightarrow \text{normal_form } \text{eval } t$.

```

intros t vt u tEu.
induction vt.
  Case "Val_True".
  inversion tEu.
  Case "Val_False".
  inversion tEu.

```

Qed.

11.4 Every normal form is a value (TAPL 3.5.7)

We first define a constructively slightly stronger progress theorem: every term is either a value, or then there is some term that it evaluates to.

Lemma progress : $\forall t, \text{val } t \vee \exists t', t \longrightarrow t'$.

```
intro t.
```

We prove the lemma with structural induction on t .

```
induction t.
```

If t is `true%tm` or `false%tm`, it is obviously a value.

```
Case "true%tm".
```

```
apply or_introl, Val_True.
```

```
Case "false%tm".
```

```
apply or_introl, Val_False.
```

```
Case "(if t1 then t2 else t3)%tm".
```

If t is a conditional, we use the inductive hypothesis on $t1$ to show that t can be evaluated further.

```
apply or_intror.
```

```
destruct IHt1 as [vt1 | [t1' t1Et1']].
```

```
  SCase "val t1".
```

If $t1$ is a value, it is either `true%tm` or `false%tm` and thus t can be evaluated with `E_IfTrue` or `E_IfFalse`

```
  destruct vt1.
```

```
    SSCase "Val_True".
```

```
     $\exists t2$ .
```

```
    apply E_IfTrue.
```

```
    SSCase "Val_False".
```

```
     $\exists t3$ .
```

```
    apply E_IfFalse.
```

SCase "exists $t1', t1 \longrightarrow t1''$ ".

On the other hand, if $t1 \longrightarrow t1'$ for some $t1'$, then t can also be evaluated further with E_If .

eexists. (* This allows the witness object to be inferred later. *)

eapply E_If .
apply $t1Et1'$.

Qed.

That every normal form is a value comes as a direct corollary from the above.

Corollary *normal_value* : $\forall t, \text{normal_form } eval\ t \rightarrow val\ t$.

intros $t\ nft$.

destruct (progress t) as [vt | [$t' tEt'$]].

apply vt .

elimtype *False*. (* We can prove anything by proving *False*. *)

eapply nft, tEt' .

Qed.

End *untyped_bool2*.

11.5 Reflexive-transitive closures (TAPL 2.2.5, 3.5.9–10)

In principle, the reflexive-transitive closure of a relation can be defined inductively as follows:

```
Inductive RTC {A : Type} (R : relation A) : relation A :=
| rtc_R a b : R a b → RTC R a b
| rtc_refl a : RTC R a a
| rtc_trans a b c : RTC R a b → RTC R b c → RTC R a c
.
```

The above, however, is not a very practical definition for proof purposes. Intuitively, we usually reason about transitive closures as a chains of relations, but the proof of $RTC\ R\ a\ b$ is a *tree*, and there can be a number of trees with different structures even though the actual chain of R -relations was exactly the same. This makes formal reasoning about them much more difficult than it ought to be.

Instead, it's more practical to explicitly define a "reflexive-stepping closure" $RSC\ R$ where every proof of $RSC\ R\ a\ b$ that is constructed by chaining $R\ a\ x1, R\ x1\ x2, \dots, R\ xn\ b$ has a unique proof.

```
Inductive RSC {A : Type} (R : relation A) : relation A :=
```

```

| rsc_refl a : RSC R a a
| rsc_step a b c : R a b → RSC R b c → RSC R a c
.

```

$RSR R$ is equivalent with $RTC R$, as the following developments show.

Lemma *rtc_rsc1* {A} (R : relation A) a b c : $RTC R a b \rightarrow RSC R b c \rightarrow RSC R a c$.

Proof.

```

intros A R a b c RTCab RSCbc.
induction RTCab in RSCbc ⊢ *.
  eapply rsc_step.
  apply H.
  apply RSCbc.
  apply RSCbc.
  apply IHRTCab1.
  apply IHRTCab2.
  apply RSCbc.

```

Qed.

In the future we shall begin using automation tactics. Just as a sample, here is how the above could also be proven:

Lemma *rtc_rsc1'* {A} (R : relation A) a b c : $RTC R a b \rightarrow RSC R b c \rightarrow RSC R a c$.

Proof.

```

intros A R a b c RTCab RSCbc.
induction RTCab in RSCbc ⊢ *; eauto using @rsc_step.

```

Qed.

Lemma *rtc_sub_rsc* {A : Type} (R : relation A) : *subrelation* (RTC R) (RSC R).

Proof.

```

intros A R.
red.
intros x y RTCxy.
eapply rtc_rsc1.
apply RTCxy.
apply rsc_refl.

```

Qed.

Lemma *rsc_sub_rtrc* {A : Type} (R : relation A) : *subrelation* (RSC R) (RTC R).

Proof.

```

intros A R a b RSCab.
induction RSCab.
  apply rtc_refl.

```

```

eapply rtc_trans.
  apply rtc_R.
  apply H.
  apply IHRSCab.

```

Qed.

We will later need the fact that $RSC\ R$ is transitive.

Lemma *rsc_trans* $\{A\} (R : relation\ A) : transitive\ (RSC\ R)$.

Proof.

```

intros A R a b c RSCab RSCbc.

induction RSCab as [a | a x b Rax RSCxb] in RSCbc ⊢ *.
  Case "RSCab = rsc_refl a".
  apply RSCbc.
  Case "RSCab = rsc_step a x b Rax RSCxb".
  eapply rsc_step.
  apply Rax.
  apply IHRSCxb.
  apply RSCbc.

```

Qed.

Module *untyped_bool3*.

Export *untyped_bool2*.

11.6 The multi-step evaluation relation (TAPL 3.5.9)

Notation " $t1 \longrightarrow^* t2$ " :=
(RSC eval t1 t2) (at level 80, no associativity) : type_scope.

11.7 Uniqueness of normal forms (TAPL 3.5.11)

Although TAPL calls this a simple "corollary" of *eval_deterministic*, we actually have to do some work in Coq to prove this. Still, using the *RSC* closures the proof is not too complex. With *RTC* proving this would be much harder, since we would have to reason about two proof trees that could be structurally quite different.

The proof proceeds very much like the proof of *eval_deterministic*: we consider various possible structures of the proofs of $t \longrightarrow^* u$ and $t \longrightarrow^* u'$ and conclude that they always have to have the same structure.

Lemma *meval_unique_normal* : $\forall t\ u\ u'$,

$t \longrightarrow^* u \rightarrow t \longrightarrow^* u' \rightarrow \text{normal_form eval } u \rightarrow \text{normal_form eval } u' \rightarrow u = u'$.

Proof.

```

intros t u u' tMu tMu' nfu nfu'.
induction tMu as [t | t x u tEx xMu].
  Case "tMu = rsc_refl t".
  induction tMu' as [t | t y u tEy yMu].
    SCase "tMu' = rsc_refl t".
    reflexivity.

    SCase "tMu' = rsc_step t y u tEy yMu".
    elimtype False.
    eapply nfu, tEy.

  Case "tMu = rsc_step t x u tEx xMu".
  induction tMu' as [t | t y u' tEy yMu'].
    SCase "tMu' = rsc_refl t".
    elimtype False.
    eapply nfu', tEx.

    SCase "tMu' = rsc_step t y u' tEy yMu'".

```

This is the interesting case. we know that $t \longrightarrow x$ and $t \longrightarrow y$, and we use *eval_deterministic* to show that therefore also $x = y$, and this along with the inductive hypothesis helps us prove the goal.

```

  apply IHxMu.
  rewrite → (eval_deterministic _ _ _ tEx tEy).
  apply yMu'.
  apply nfu.

```

Qed.

11.8 Termination of evaluation (TAPL 3.5.12)

Before we start proving the actual normalization theorem, we need an auxiliary congruence lemma about multi-step evaluation. This is very straightforward, and we could think of this as just mapping *E_If* over the evaluation chain to produce a new chain.

Lemma *meval_if_congr* : $\forall t1 t1' t2 t3,$

$t1 \longrightarrow^* t1' \rightarrow$

$(\text{if } t1 \text{ then } t2 \text{ else } t3)\%tm \longrightarrow^* (\text{if } t1' \text{ then } t2 \text{ else } t3)\%tm.$

Proof.

```

intros t1 t1' t2 t3 t1Mt1'.

```

```

induction t1Mt1' as [t1 | t1 u t1' t1Eu uMt1'].
  Case "t1Mt1' = rsc_refl t1".
    apply rsc_refl.
  Case "t1Mt1' = rsc_step t1 u t1' t1Eu uMt1'".
    eapply rsc_step.
    apply E_If, t1Eu.
    apply IHuMt1'.

```

Qed.

End *untyped_bool3*.

We define normalization outside of any modules so we can also speak of the normalization of other languages.

Definition *normalizing* {A} (R : relation A) :=
 $\forall t : A, \exists t', RSC\ R\ t\ t' \wedge normal_form\ R\ t'$.

Module *untyped_bool4*.

Export *untyped_bool3*.

The proof of normalization for the boolean language is relatively straightforward but there is lots of duplication which makes the proof script longer than it could be. In the future we shall look at ways of shortening the scripts.

Lemma *eval_normalizing* : *normalizing eval*.

Proof.

```

red.
intro t.

```

For every t , we need to find some normal form t' such that $t \longrightarrow^* t'$. We proceed by induction on t .

```

induction t as [ | | t1 IHt1 t2 IHt2 t3 IHt3].
  Case "t = true%tm".

```

If $t = true\%tm$, it is in normal form already, and we can get from t to itself in zero evaluation steps.

```

   $\exists (true)\%tm.$ 
  split.
  apply rsc_refl.

```

We make use of the previously proven fact that every value is a normal form, since it is trivial to show that $true\%tm$ is a value.

```

  apply value_normal.

```

apply *Val_True*.

Case "t = false%tm".

The *false%tm* case is handled analogously.

\exists (*false%tm*).

split.

apply *rsc_refl*.

apply *value_normal*.

apply *Val_False*.

Case "t = (if t1 then t2 else t3)%tm".

If *t* is a conditional, we make use of the inductive hypothesis which states that *t1* evaluates to some normal form *t1'*.

destruct *IHt1* as [*t1'* [*t1Et1'* *nft1'*]].

Since *t1'* is a normal form, it is also a value.

apply *normal_value* in *nft1'*.

Now just consider which value it is.

destruct *nft1'*.

SCase "Val_True".

If *t1'* is *true%tm*, we can construct the following derivations:

$t \longrightarrow^*$ (if *t1'* then *t2* else *t3*) (with *eval_ifcongr*) (if *t1'* then *t2* else *t3*) \longrightarrow *t2* (with *E_IfTrue*) *t2* \longrightarrow^* *t2'* (with an inductive hypothesis)

We simply chain these together to derive $t \longrightarrow^* t2'$, where *t2'* is a normal form.

destruct *IHt2* as [*t2'* [*t2Et2'* *nft2'*]].

\exists *t2'*.

split.

SSCase "t \longrightarrow^* t2''".

eapply *rsc_trans*.

apply *meval_if_congr*.

apply *t1Et1'*.

eapply *rsc_step*.

apply *E_IfTrue*.

apply *t2Et2'*.

SSCase "normal_form eval t2''".

apply *nft2'*.

SCase "Val_False".

For $t1' = false\%tm$, the proof is analogous.

```
destruct IHt3 as [t'3 [t3Et3' nft3']].
∃ t'3.
split.
  eapply rsc_trans.
  apply meval_if_congr.
  apply t1Et1'.
  eapply rsc_step.
  apply E_IfFalse.
  apply t3Et3'.
  apply nft3'.
```

Qed.

End *untyped_bool4*.

Module *untyped_arith1*.

Require Export *untyped_arith_lang*.

11.9 Stuck terms (TAPL 3.5.15)

Definition *stuck* ($t : term$) := *normal_form eval t* $\wedge \sim(\text{val } t)$.

In the boolean-arithmetic language, unlike the boolean language, there are terms that can get stuck.

Lemma *can_get_stuck* : $\exists t, stuck\ t$.

$\exists (succ\ false)\%tm$.

split.

Case "t is in normal form".

intros $t' tEt'$.

inversion tEt' .

SCase " $tEt' = E_Succ\ t1\ t1''$ ".

inversion $H0$.

Case "t is not a value".

intro vt .

inversion vt as [| | $t\ nvt$].

SCase " $vt = Val_NV\ t\ nvt$ ".

inversion nvt as [| $f\ nvf$].

SSCase " $nvt = NV_Succ\ f\ nvf$ ".

inversion nvf .

Qed.

End *untyped_arith1*.

| | | | | |
|-----------------|-------|-----------------|-----|-------------------------|
| $type, T, S, U$ | $::=$ | (T) | M | types: |
| | | \mathbf{Bool} | | type of booleans |
| | | \mathbf{Nat} | | type of natural numbers |

Figure 4: \mathbb{BN} , type syntax

```
Module untyped_bool := untyped_bool4.
Module untyped_arith := untyped_arith1.
```

12 Typed arithmetic expressions (TAPL Chapter 8)

```
Require Export tapl_ch3.
Module typed_arith1.
Require Export typed_arith_lang.
```

12.1 Uniqueness of typing (TAPL 8.2.4)

We prove that a term can have at most one type. The idea is simple: we do induction on the term t , and consider all the various forms that t can have. For each of the forms, there is exactly one typing rule that applies to terms of that form, so by inversion we see that $t : T$ and $t : T'$ must be formed by the same typing rule and they must assign the same type to T and T' .

Lemma *typing_unique* : $\forall t T T', \vdash t : T \rightarrow \vdash t : T' \rightarrow T = T'$.

Proof.

```
intros t T T' t_T t_T'.
induction t.
  Case "true%tm".
    inversion t_T.
    inversion t_T'.
    reflexivity.
  Case "false%tm".
    inversion t_T.
    inversion t_T'.
    reflexivity.
```

This is the only non-trivial case. A conditional expression can only be

$t : T$ Typing

$$\begin{array}{c} \frac{}{\mathbf{true} : \mathbf{Bool}} \quad \text{T_TRUE} \\ \frac{}{\mathbf{false} : \mathbf{Bool}} \quad \text{T_FALSE} \\ \frac{t_1 : \mathbf{Bool} \quad t_2 : T \quad t_3 : T}{\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 : T} \quad \text{T_IF} \\ \frac{}{0 : \mathbf{Nat}} \quad \text{T_ZERO} \\ \frac{t_1 : \mathbf{Nat}}{\mathbf{succ } t_1 : \mathbf{Nat}} \quad \text{T_SUCC} \\ \frac{t_1 : \mathbf{Nat}}{\mathbf{pred } t_1 : \mathbf{Nat}} \quad \text{T_PRED} \\ \frac{t_1 : \mathbf{Nat}}{\mathbf{iszero } t_1 : \mathbf{Bool}} \quad \text{T_ISZERO} \end{array}$$

Figure 5: \mathbb{BN} , typing rules

typed by the T_If rule, but the rule does not directly tell us what the type of the conditional term t is. However, it does tell us that the type of $t2$ is the same as the type of t . So from the inversions we get $t2 : T$ and $t2 : T'$. We can then use the inductive hypothesis to show that $T = T'$.

Case " $t = (\text{if } t1 \text{ then } t2 \text{ else } t3)$ ".

inversion t_T .

inversion t_T' .

subst.

apply $IHt2$.

 apply $H4$.

 apply $H11$.

Case " $t = 0$ ".

inversion t_T .

inversion t_T' .

reflexivity.

We could continue like this through the rest of the term constructors, but by this point it should be obvious that handling each case separately is just tedious, since we are doing the same thing every time. So let's abort and try again.

Abort.

Lemma *typing_unique* : $\forall t T T', \vdash t : T \rightarrow \vdash t : T' \rightarrow T = T'$.

Proof.

 intros $t T T' t_T t_T'$.

In the previous proof, in every case we did inversion on t_T and t_T' , and we could then solve all but one of the cases by reflexivity. We can use two features of Coq's tactic language to make a proof like this easier:

- a tactic of the form $tac1 ; tac2$ applies $tac2$ to *all* the subgoals generated by $tac1$. If $tac2$ fails on any of the subgoals, the whole tactic fails.
- a tactic `try $tac1$` attempts to apply the tactic $tac1$, but will not fail even if $tac1$ fails.

We can now directly say "after induction, do inversion on both typing judgements in all subgoals, and try to solve them by reflexivity".

 induction t ; inversion t_T ; inversion t_T' ; try reflexivity.

We are then left with a single subgoal that couldn't be solved by reflexivity. Even here, though, we can use the `;` syntax: after we apply the

inductive hypothesis, both of the subgoals can be solved by assumptions that are found in the context. The `assumption` tactic, when used without arguments, tries to find a suitable assumption from the context. This tactic is suitable for solving both subgoals, so we can use it with `;` after applying the inductive hypothesis.

```
apply IHt2; assumption.
```

Qed.

12.2 Canonical forms (TAPL 8.3.1)

The canonical forms lemma for booleans states that a value of type `Bool` is either `true%tm` or `false%tm`. Since we already have a `nv` predicate for numeric values, we might as well define a separate predicate for boolean values, and use that in our formalization of the lemma.

```
Inductive bv : term → Prop :=
```

```
| bv_true : bv true
```

```
| bv_false : bv false
```

```
.
```

```
Lemma canonical_forms_bool : ∀ v, val v → ⊢ v : Bool → bv v.
```

Proof.

```
intros v val_v v_Boot.
```

The idea of the proof is that we consider possible values and possible ways of assigning the type `Bool` to the value, and show that for values other than `true%tm` and `false%tm`, it's impossible for them to have type `Bool`.

We do this by first using `inversion`, and then applying the `subst` tactic, which "rewrites away" as many equalities from the context as it can. At that point we are likely to have an impossible equality in the context (impossible in the sense that the equated terms are formed by distinct constructors). The `discriminate` tactic then searches for an impossible equality in the context, and if it finds one, it can prove anything from that.

So we end up with this strategy: do `inversion` on the value and typing judgements, then rewrite away as many equalities as we can, and then, if we are dealing with `true%tm` or `false%tm`, show that they are indeed boolean values, or if that doesn't succeed, try to show that the context is impossible.

To implement this, we need one more tactic: `solve [tac1 | tac2 | ...]` tries to solve the goal first by `tac1`, and if it fails then it tries `tac2` and so on, and if none of the subtactics solves the goal, the entire tactic will fail. We prepend `try` in order to make this failing non-fatal.

It is perhaps easier to get a better idea of what's going on by first manually stepping through the different cases by using individual tactic invocations, and then using the below tactic which solves the whole thing almost directly.

```
inversion val_v; inversion v_Bool; subst;
  try solve [apply bv_true | apply bv_false | discriminate].
```

We are left with two subgoals that weren't solved automatically. Conditional expressions and `iszero` tests can have the type `Bool`, so that is not a contradictory assumption. Nor is the assumption that they are values, *directly*. Both `iszero`- and `if`-terms could be values *if* they were numerical values. That is of course impossible, but a single inversion on `val_v` didn't reveal that. So we have to do a new inversion on the assumption `nv v`.

```
Case "v = (if t1 then t2 else t3)%tm".
```

```
inversion H.
```

```
Case "v = (iszero t1)%tm".
```

```
inversion H.
```

Qed.

The proof of the canonical forms lemma for numbers is similar. A difference is that doing induction on `val v` will immediately give us `nv v` in one case, and then we can prove the goal by assumption.

Again, it may be illustrative to try to prove this step by step to see what is going on. The `;` syntax is powerful, but compound tactics do many things in a single step so they are harder to follow.

Lemma `canonical_forms_nat` : $\forall v, \text{val } v \rightarrow \vdash v : \text{Nat} \rightarrow \text{nv } v$.

Proof.

```
intros v val_v v_Nat.
inversion val_v; inversion v_Nat; subst;
  try solve [assumption | discriminate].
```

Qed.

12.3 Progress (TAPL 8.3.2)

We show that if a term t is well-typed (i.e. has some type T), then either it is a value or then it can be evaluated further. We do induction on the proof of $\vdash t : T$. The cases `T_True`, `T_False` and `T_Zero` directly indicate that t is a value. In the other cases t is a compound expression, and then we make use of the inductive hypothesis as described below.

Lemma `progress` : $\forall t T, \vdash t : T \rightarrow \text{val } t \vee \exists t', t \longrightarrow t'$.

Proof.

```
intros t T t_T.  
induction t_T.  
Case "T_True".  
left. (* Applies the first constructor, here or_introl *)  
constructor. (* Applies some suitable constructor, here Val_True *)
```

```
Case "T_False".  
left.  
constructor.
```

Now t is of the form $(\text{if } t1 \text{ then } t2 \text{ else } t3)\%tm$. By the inductive hypothesis we know that either $t1$ is a value or it can be evaluated further.

If $t1$ is a value, then, since we know from the typing rule T_If that $t1 : Bool$, we can use the canonical forms lemma to show that $t1$ is either $true\%tm$ or $false\%tm$, and then we can then show that t evaluates further by either of the computation rules E_IfTrue or $E_IfFalse$.

Otherwise, $t1 \longrightarrow t1'$ for some $t1'$. Then t evaluates further by the congruence rule E_If .

```
Case "T_If".  
right.
```

The complicated introduction pattern after `as` means the following:

- if IHt_T1 was created using the first constructor (or_introl), bind the constructor argument to val_t1
- if IHt_T1 was created using the second constructor (or_intror), then destruct its argument (the proof of an existential formula), and bind its constructor arguments to $t1'$ and $t1Et1'$. These will then be the witness object and a proof that $t1 \longrightarrow t1'$.

```
destruct IHt_T1 as [val_t1 | [t1' t1Et1']].  
SCase "val t1".
```

The case tactic is like `destruct`, but it doesn't clear from the context the assumptions that the argument depends on.

```
case (canonical_forms_bool t1 val_t1 t_T1).  
SSCase "BV_True".  
 $\exists t2$ .  
constructor.
```

```

      SSCase "BV_False".
      ∃ t3.
      constructor.
    SCase "t1 → t1'".
    eexists.
    apply E_If.
    apply t1Et1'.
  Case "T_Zero".
  left.
  constructor.
  constructor.
  Case "T_Succ".
  destruct IHt_T as [val_t1 | [t1' t1Et1']].
  SCase "val t1".
  left.
  constructor.
  constructor.
  apply canonical_forms_nat; assumption.
  SCase "t1 → t1'".
  right.
  eexists.
  apply E_Succ.
  apply t1Et1'.

```

The treatment of other compound expressions is analogous to *T_If*.

```

  Case "T_Pred".
  right.
  destruct IHt_T as [val_t1 | [t1' t1Et1']].
  SCase "val t1".
  case (canonical_forms_nat t1 val_t1 t_T).
  SSCase "NV_Zero".
  ∃ 0%tm.
  apply E_PredZero.
  SSCase "NV_Succ".
  intros n nvn.
  ∃ n.
  apply E_PredSucc.
  apply nvn.
  SCase "t1 → t1'".

```

```

    eexists.
    apply E_Pred.
    apply t1Et1'.
Case "T_IsZero".
right.
destruct IHt_T as [val_t1 | [t1' t1Et1']].
  SCase "val t1".
  case (canonical_forms_nat t1 val_t1 t_T).
    SScase "NV_Zero".
     $\exists$  true%tm.
    apply E_IsZeroZero.
    SScase "NV_Succ".
    intros n nvn.
     $\exists$  false%tm.
    apply E_IsZeroSucc.
    apply nvn.
  SCase "t1  $\longrightarrow$  t1'".
  eexists.
  apply E_IsZero.
  apply t1Et1'.

```

Qed.

12.4 Preservation (TAPL 8.3.3)

We then show that evaluation of a well-typed term preserves its type.

First a straightforward lemma to point out that all numeric values have the type *Nat*.

Lemma *nv_in_Nat* : $\forall n, nv\ n \rightarrow \vdash n : Nat$.

Proof.

```

  intros n nvn.
  induction nvn as [| n' nvn'].
  apply T_Zero.
  apply T_Succ.
  apply IHnvn'.

```

Qed.

The actual type preservation proof proceeds again by induction on the typing derivation.

(Why do we always do induction on the typing derivation? Because

then we get information both about the form of a term and the form of a type, and this is often much more useful than doing induction on terms, types or evaluation derivations.)

Some typing derivations constrain t to be a value, and then the assumption $t \longrightarrow t'$ is impossible, which we show by *inversion*.

Other typing derivations constrain t to be e.g. of the form (if $t1$ then $t2$ else $t3$). Then inversion on the derivation of $t \longrightarrow t'$ shows that there are only a couple of rules that can evaluate such terms. We consider each of those cases separately and see that they can all be proved straightforwardly: for instance, if the evaluation is by $E_IfTrue\ t2\ t3$: (if *true* then $t2$ else $t3$) $\longrightarrow t2$ then we know by the typing rule T_If that $t2$ has the same type as t .

Lemma *preservation* : $\forall t\ t'\ T, \vdash t : T \rightarrow t \longrightarrow t' \rightarrow \vdash t' : T$.

Proof.

```

intros t t' T t_T tEt'.
induction t_T in t', tEt' ⊢ *.
  Case "T_Bool".
    inversion tEt'.
  Case "T_False".
    inversion tEt'.
  Case "T_If".
    inversion tEt'; subst.
    SCase "E_IfTrue".
      apply t_T2.
    SCase "E_IfFalse".
      apply t_T3.
    SCase "E_If".
      apply T_If.
        apply IHt_T1.
        apply H3.
        apply t_T2.
        apply t_T3.
  Case "T_Zero".
    inversion tEt'.
  Case "T_Succ".
    inversion tEt'; subst.
    SCase "E_Succ".
      apply T_Succ.

```

```

    apply IHt_T.
    apply H0.
  Case "T_Pred".
  inversion tEt'; subst.
  SCase "E_PredZero".
  apply T_Zero.

```

In this case we know that the evaluation is of the form $\text{pred}(\text{succ } v) \longrightarrow v$ where v is a numeric value. We use the previously proven lemma to show that v indeed has type Nat , just like t .

```

  SCase "E_PredSucc".
  apply nv_in_Nat.
  apply H0.
  SCase "E_Pred".
  apply T_Pred.
  apply IHt_T.
  apply H0.
  Case "T_IsZero".
  inversion tEt'; subst.
  SCase "E_IsZeroZero".
  apply T_True.
  SCase "E_IsZeroSucc".
  apply T_False.
  SCase "E_IsZero".
  apply T_IsZero.
  apply IHt_T.
  apply H0.

```

Qed.

12.5 Automation

In the progress and preservation theorems above, we could see lots of very tedious and fairly trivial reasoning: for the most part, we were simply applying lemmas, assumptions and constructors.

The `auto` and `eauto` tactics can be used to automate these relatively simple parts of proofs (and in fact they can be customized to be much more powerful). In their basic form, the tactics try to solve the current goal by introductions, trying assumptions in the context, and applying various lemmas whose result type matches the goal. The difference between `auto` and

eauto is that eauto will use eapply and not just apply.

The list of lemmas to try can be given to the tactics with a using clause, or else by specifying with a with clause a *hint database* where relevant lemmas are collected.

New lemmas are added to the database with the Hint command. Hint Constructors adds all the constructors of a given inductive type, and Hint Resolve adds individual lemmas.

We here build a hint database named *typed_arith* and add some useful hints to it.

```
Hint Constructors ex typing eval val nv or and : typed_arith.
```

```
Hint Resolve canonical_forms_bool canonical_forms_nat nv_in_Nat : typed_arith.
```

We can now solve the progress theorem much easier than previously.

If we start a proof script with Proof with *tac1*. then wherever we finish a tactic with ..., the tactic *tac1* is tried to solve the generated subgoals. This makes it easy to apply an automation tactic in multiple places.

```
Lemma progress' :  $\forall t T, \vdash t : T \rightarrow \text{val } t \vee \exists t', t \longrightarrow t'$ .
```

```
Proof with eauto with typed_arith.
```

```
  intros t T t_T.
```

```
  induction t_T...
```

The automation tactic now solved the trivial cases directly. What it did not know to do was to examine the inductive hypotheses to check whether subterms are values or if they can be evaluated further. This we now have to tell explicitly.

```
  Case "T_If".
```

```
  destruct IHt_T1 as [val_t1 | [t1' t1Et1']]...
```

The automation again solved one of the subgoals (where $t1 \longrightarrow t1'$) directly. In the remaining case where *t1* is a value, we still need to do the analysis to see which boolean value *t1* is. We could again apply *canonical_forms_nat* directly, but since it is in our hint database, we can try an easier approach.

The elimtype tactic eliminates (does induction on) a term with the specified type. We have to produce a term with that type in a new subgoal. So elimtype (*bv t1*) means: "consider the cases where *t1* is *true*%tm and *t1* is *false*%tm, and try to prove *bv t1*". It turns out that the automation can solve all these three subgoals: the first two by applying the computation rules *E_IfTrue* and *E_IfFalse*, and the last one by applying *canonical_forms_nat*, since all of these are in the hint database.

```
  SCase "val t1".
```

```

    elimtype (bv t1)...
Case "T_Succ".
destruct IHt_T as [val_t1 | [t1' t1Et1']...]
Case "T_Pred".
destruct IHt_T as [val_t1 | [t1' t1Et1']...]
  SCase "val t1".
    elimtype (nv t1)...
Case "T_IsZero".
destruct IHt_T as [val_t1 | [t1' t1Et1']...]
  SCase "val t1".
    elimtype (nv t1)...

```

Qed.

Finally, the preservation theorem can now be proven almost completely automatically.

Lemma *preservation'* : $\forall t t' T, \vdash t : T \rightarrow t \rightarrow t' \rightarrow \vdash t' : T$.

Proof.

```

  intros t t' T t_T tEt'.
  induction t_T in t', tEt' ⊢ *;
  inversion tEt'; subst; auto with typed_arith.

```

Qed.

It is easy to get drunk with the power of automation, but preferably automation should only be used to prove things that are reasonably obvious to a human. In particular, it is a bad idea to make the machine prove things that human couldn't directly prove.

End *typed_arith1*.

Module *typed_arith* := *typed_arith1*.

13 Typed lambda calculus (TAPL Chapter 9)

We now turn to languages with functions. The presence of variables, typing contexts and substitutions complicates formal reasoning about the languages.

Require Export *Logic*.

Require Export *notations*.

| | | | | |
|---------------------------------|-----|---|---|--|
| <i>term, t</i> | ::= | (t) \mathbf{true} \mathbf{false} $\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3$ x $\lambda x : T . t$ $t_1 t_2$ $[x \mapsto t'] t$ | M bind x in t M | terms: constant true constant false conditional variable abstraction application |
| <i>type, T, S, U</i> | ::= | (T) \mathbf{Bool} $T \rightarrow U$ | M | types: type of booleans function type |
| <i>ctx, Γ</i> | ::= | $\{ \}$ $\Gamma, x : T$ | | typing contexts |

VAL t

$$\frac{}{\mathbf{VAL true}} \text{ VAL_TRUE}$$

$$\frac{}{\mathbf{VAL false}} \text{ VAL_FALSE}$$

$$\frac{}{\mathbf{VAL}(\lambda x : T . t)} \text{ VAL_ABS}$$

Figure 6: $\rightarrow \mathbb{B}$, syntax

$t \longrightarrow t'$ Evaluation

$$\begin{array}{c}
\frac{}{\mathbf{if\ true\ then\ } t_2 \mathbf{\ else\ } t_3 \longrightarrow t_2} \text{E_IFTRUE} \\
\frac{}{\mathbf{if\ false\ then\ } t_2 \mathbf{\ else\ } t_3 \longrightarrow t_3} \text{E_IFFALSE} \\
\frac{t_1 \longrightarrow t'_1}{\mathbf{if\ } t_1 \mathbf{\ then\ } t_2 \mathbf{\ else\ } t_3 \longrightarrow \mathbf{if\ } t'_1 \mathbf{\ then\ } t_2 \mathbf{\ else\ } t_3} \text{E_IF} \\
\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \text{E_APP1} \\
\frac{\mathbf{VAL\ } t_1 \quad t_2 \longrightarrow t'_2}{t_1 t_2 \longrightarrow t_1 t'_2} \text{E_APP2} \\
\frac{\mathbf{VAL\ } t_2}{(\lambda x : T. t_{12}) t_2 \longrightarrow [x \mapsto t_2] t_{12}} \text{E_APPABS}
\end{array}$$

Figure 7: $\longrightarrow \mathbb{B}$, evaluation rules

13.1 Decidable equality for numbers

We have chosen to represent variable identifiers simply as natural numbers. This is reasonable enough, since the only thing we want from identifiers is that there is an infinite number of them and that we can effectively compare them for equality.

This latter point is worth explicating. An effective method for comparing numbers for equality is a function that can tell us whether two numbers are the same or not. We already have such a function: *beq_nat*. We now have to prove that *beq_nat n m* actually returns *true* if and only if $n = m$.

Theorem *beq_nat_eq* : $\forall n m, \text{beq_nat } n m = \text{true} \leftrightarrow n = m$.

Proof.

Exercise *Admitted*.

Given the above result, it directly follows that equality of numbers is decidable.

Corollary *nat_eq_dec* : $\forall n m : \text{nat}, n = m \vee n \neq m$.

Proof.

intros *n m*.

$x : T \text{ in } \Gamma$ bound

$$\frac{}{x : T \text{ in } \Gamma, x : T} \text{ B_HERE}$$

$$\frac{x \langle \rangle y \quad x : T \text{ in } \Gamma}{x : T \text{ in } \Gamma, y : T'} \text{ B_NEXT}$$

$\mathbf{FVS } tx$

$$\frac{\mathbf{FVS } t_1 x}{\mathbf{FVS } (\text{if } t_1 \text{ then } t_2 \text{ else } t_3) x} \text{ FVS_IF1}$$

$$\frac{\mathbf{FVS } t_2 x}{\mathbf{FVS } (\text{if } t_1 \text{ then } t_2 \text{ else } t_3) x} \text{ FVS_IF2}$$

$$\frac{\mathbf{FVS } t_3 x}{\mathbf{FVS } (\text{if } t_1 \text{ then } t_2 \text{ else } t_3) x} \text{ FVS_IF3}$$

$$\frac{}{\mathbf{FVS } x x} \text{ FVS_VAR}$$

$$\frac{x \langle \rangle y \quad \mathbf{FVS } t x}{\mathbf{FVS } (\lambda y : T. t) x} \text{ FVS_ABS}$$

$$\frac{\mathbf{FVS } t_1 x}{\mathbf{FVS } (t_1 t_2) x} \text{ FVS_APP1}$$

$$\frac{\mathbf{FVS } t_2 x}{\mathbf{FVS } (t_1 t_2) x} \text{ FVS_APP2}$$

Figure 8: $\rightarrow \mathbb{B}$, type contexts and free variables

$\boxed{\Gamma \vdash t : T}$ Typing

$$\begin{array}{c} \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \quad \text{T_TRUE} \\ \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}} \quad \text{T_FALSE} \\ \frac{\Gamma \vdash t_1 : \mathbf{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 : T} \quad \text{T_IF} \\ \frac{x : T \mathbf{ in } \Gamma}{\Gamma \vdash x : T} \quad \text{T_VAR} \\ \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad \text{T_ABS} \\ \frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad \text{T_APP} \end{array}$$

Figure 9: $\rightarrow \mathbb{B}$, typing rules

```

destruct (beq_nat_eq n m) as [f g].
destruct (beq_nat n m).
  Case "beq_nat n m = true".
  left.
  apply f.
  reflexivity.
  Case "beq_nat n m = false".
  right.
  intro nQm.
  discriminate g.
  apply nQm.

```

Qed.

In practice we often need to track the dependency between the result of an application of *beq_nat* and the equality of its arguments. The following versions of the result make this easier.

Definition *decides b P* := match *b* with *true* ⇒ *P* | *false* ⇒ ¬*P* end.

Corollary *beq_decides_eq_nat* : ∀ *n m*, *decides (beq_nat n m) (n = m)*.

Proof.

```

intros n m.
destruct (beq_nat_eq n m) as [f g].
destruct (beq_nat n m).
  Case "beq_nat n m = true".
  apply f.
  reflexivity.
  Case "beq_nat n m = false".
  intro nQm.
  discriminate g.
  apply nQm.

```

Qed.

Corollary *beq_nat_eq_dec* : ∀ *x y*,

$(\text{beq_nat } x \ y = \text{true} \wedge x = y) \vee (\text{beq_nat } x \ y = \text{false} \wedge x \neq y)$.

Proof.

```

intros x y.
assert (Q := beq_decides_eq_nat x y).
destruct (beq_nat x y); simpl in Q ⊢ *; auto using or-introl, or-intror,
conj.

```

Qed.

Module *typed_bool_fun*.

Require Export *typed_bool_fun_lang*.

Hint Constructors *term val eval type typing and or ex bound fvs : bool_fun*.

Definition *well_typed* (*t* : *term*) : Prop :=
 $\exists T, \vdash t : T$.

13.2 Canonical forms (TAPL 9.3.4)

The canonical forms lemmas are exactly similar to those we encountered previously, except now for values of a function type we have the notion of "function values", i.e. abstractions.

Inductive *bv* : *term* \rightarrow Prop :=
 | *bv_true* : *bv true*
 | *bv_false* : *bv false*
 .

Inductive *fv* : *term* \rightarrow Prop :=
 | *fv_abs* *x T t* : *fv* (" *x* : *T* , *t*).

Hint Constructors *bv fv : bool_fun*.

Lemma *canonical_forms_bool* : $\forall v, \text{val } v \rightarrow \vdash v : \text{Bool} \rightarrow \text{bv } v$.

Proof.

 intros *v val_v v_Boot*.
 induction *val_v*; inversion *v_Boot*; subst;
 try inversion *H*; eauto with *bool_fun*.

Qed.

Lemma *canonical_forms_fun* : $\forall v T1 T2, \text{val } v \rightarrow \vdash v : (T1 \rightarrow T2) \rightarrow \text{fv } v$.

Proof.

 intros *v T1 T2 val_v v_F*.
 induction *val_v*; inversion *v_F*; subst;
 try inversion *H*; eauto with *bool_fun*.

Qed.

Hint Resolve *canonical_forms_bool canonical_forms_fun : bool_fun*.

13.3 Progress (TAPL 9.3.5)

The progress theorem for simply typed lambda calculus (with booleans) is similar to the one for the plain boolean language. The new cases to consider are abstractions (which are values immediately), variables (which cannot appear outside an abstraction in an empty typing context), and applications (which are handled very much like conditionals).

A technical complication arises due to the presence of contexts in the typing relation, and in particular due to the fact that the emptiness of the context is crucial for the T_Var case. When we have a hypothesis of the form $\{\} \vdash t : T$, the standard induction tactic forgets the emptiness of the context and makes the T_Var case impossible to prove.

The way to deal with this is to use the `remember` tactic to replace $\{\}$ with Γ in the hypothesis and add a new hypothesis $\Gamma = \{\}$ in the context. Then we can do induction and use `subst` to replace Γ once again with $\{\}$.

It should be noted that this only works because in all the cases we consider, the context in the inductive hypotheses is the same, i.e. $\{\}$. We could not use the inductive hypotheses in the T_Abs case, but thankfully we don't need to, since abstractions are directly values.

Theorem `progress` $t : \text{well_typed } t \rightarrow \text{val } t \vee \exists t', t \longrightarrow t'$.

Proof with `eauto` with `bool_fun`.

```

intros t [T t_T].
remember {}%ctx as  $\Gamma$ .
induction t_T; subst  $\Gamma$ ...

  Case "T_If {} t1 t2 t3 T t_T1 t_T2 t_T3".
  destruct IHt_T1 as [val_t1 | [t1' t1Et1']]...
    SCase "val t1".
    elimtype (bv t1)...

  Case "T_Var {} x T H".
  (* Impossible to have a bound variable in an empty context. *)

  inversion H.

  Case "T_App {} t1 t2 T12 T11 t_T1 t_T2".
  destruct IHt_T1 as [val_t1 | [t1' t1Et1']]...
    SCase "val t1".
    destruct IHt_T2 as [val_t2 | [t2' t2Et2']]...
      SSCase "val t2".
      elimtype (fv t1)...

```

Qed.

13.4 Closed terms

Firstly, a basic definition: a term is *closed*, if no variables appear free in it. (Recall that a variable appears free in a term if it is not bound, i.e. the variable occurs outside of any abstractions that bind the same variable. Hence x appears free in $((\lambda y:T.y) \$ x)$ tm but y does not.)

Definition $closed (t : term) := \forall x, \sim(fvs\ t\ x)$.

There is an obvious connection between free variables and typing: if $\Gamma \vdash t : T$, and a variable x appears free in t , then x must appear in Γ (i.e. it must be bound to some T' in Γ).

Lemma $free_in_context : \forall \Gamma\ t\ T\ x,$
 $(\Gamma \vdash t : T) \rightarrow fvs\ t\ x \rightarrow \exists T', bound\ x\ T'\ \Gamma.$

Proof.

```
intros  $\Gamma\ t\ T\ x\ G\_t\_T\ x\_f\_t$ .
induction  $G\_t\_T$ ; inversion  $x\_f\_t$ ; subst; auto.
Case "T_Var".
  SCase "FVS_Var".
   $\exists T$ .
  apply  $H$ .
Case "T_Abs".
  SCase "FVS_Abs".
  destruct ( $IHG\_t\_T\ H4$ ) as [ $T'\ B$ ].
   $\exists T'$ .
  inversion  $B$ ; subst.
  SSCase "B_Here".
  contradiction  $H3$ .
  reflexivity.
  SSCase "B_Next".
  assumption.
```

Qed.

A direct corollary is that if a term is well-typed (i.e. has some type in an empty context), then it must be closed.

Corollary $well_typed_is_closed : \forall t, well_typed\ t \rightarrow closed\ t$.

Proof.

```
intros  $t\ [T\ t\_T]$ .
red.
red.
intros  $x\ x\_f\_t$ .
destruct ( $free\_in\_context\ \{\}\ t\ T\ x\ t\_T\ x\_f\_t$ ) as [ $T'\ B$ ].
inversion  $B$ .
```

Qed.

13.5 Context invariance

Instead of explicit permutation and weakening lemmas (as in TAPL 9.3.6 and 9.3.7), we prove a somewhat more general property about the typing relation: only the types of the variables that occur free in t matter when typing it. That is, if Γ and Γ' give the same type to all the variables that appear free in t , then if $\Gamma \vdash t : T$, then also $\Gamma' \vdash t : T$.

Lemma *context_invariance* :

```

  ∀ Γ Γ' t T,
  (∀ x T', fvs t x → bound x T' Γ → bound x T' Γ') →
  Γ ⊢ t : T →
  Γ' ⊢ t : T.

```

Proof with `eauto 20` with `bool_fun`.

```

  intros Γ Γ' t T H G_t_T.
  induction G_t_T in Γ', H ⊢ *...

```

The only interesting case is the one for abstractions. After applying the typing rule for abstractions, it remains to show that $\Gamma', x:T1 \vdash t2 : T2$ if $\Gamma, x:T1 \vdash t2 : T2$. We then use the inductive hypothesis, which was generalized over *all* contexts, so we can apply it to the extended contexts above.

Then it remains to show that for any variable $x0$ free in $t2$, $\Gamma', x:T1$ binds it to the same types as $\Gamma, x:T1$. This is proven by considering whether $x0$ was found from the head of $\Gamma, x:T1$. If it was, then $x0 = x$, and both contexts bind it to $T1$. If it is not, then $x0 \neq x$, and we use the original hypothesis that says that Γ' binds variables free in t to the same types as Γ .

Then it remains to show that $x0$, which is free in $t2$, is also free in $t = \lambda x:T1. t2$. This follows from the fact that $x0 \neq x$, so the lambda will not bind $x0$.

It turns out that most of this reasoning can be implemented by automation. Still, it can be instructive exercise to solve everything after the inversion manually.

```

  Case "T_Abs, t = (λx:T1, t2), T = T1 → T2, G, x:T1 ⊢ t2 : T2".
  apply T_Abs.
  apply IHG_t_T.
  intros x' T' x'_f_t2 B.
  inversion B...

```

Qed.

13.6 Weakening in an empty context

As a direct corollary, if t has type T in an empty context (and is therefore closed), it has type T in any context.

Note that we cannot prove the most general form of weakening (if $\Gamma \vdash t : T$ then $\Gamma, x:S \vdash t : T$, because x might appear free in t).

Corollary *empty_weakening* : $\forall \Gamma t T, \vdash t : T \rightarrow \Gamma \vdash t : T$.

Proof with eauto with *bool_fun*.

```
intros  $\Gamma t T t\_T$ .
apply (context_invariance {}).
  intros  $x T' x\_f\_t x\_B\_E$ .
  inversion  $x\_B\_E$ .
```

Qed.

13.7 Shadowed variable removal

An additional lemma that we need is the followign: if the same variable name appears twice in a row in a context, the one that is shadowed by the other can be removed without altering the typing of a term.

The intuitive justification should be clear: if a variable is shadowed by another one with the same name, then it cannot be referenced and no term can depend on its type.

Formally, we prove the lemma by applying context invariance and then showing that the context with the shadowed variable binds all variables to the same type as the context without the shadowed variable. The proof makes for the first time use of the decidability of equality between numbers: When considering contexts $\Gamma, x:S', x:S$ and $\Gamma, x:S$ and looking up a variable x' , then we need to consider both the cases where $x = x'$ and $x \neq x'$. In the first case, both contexts bind x to S . In the second case, both contexts bind x' to whatever Γ binds them to.

For explicitness, the proof script is given manually below. Automation could be used to shorten the proof.

Lemma *shadowed_removable* : $\forall \Gamma x S S' t T,$
 $\{\{\Gamma, x : S'\}, x : S\} \% \text{ctx} \vdash t : T \rightarrow \{\Gamma, x : S\} \vdash t : T$.

Proof.

```
intros  $\Gamma x S S' t T$ .
apply context_invariance.
intros  $x' T' x'_f\_t2 x'_B\_G1$ .
destruct (nat_eq_dec  $x x'$ ) as [ $xQx' \mid nxQx'$ ].
  Case " $x = x'$ ".
```

```

subst x'.
inversion x'_B_G1; subst.
  SCase "B_Here".
  apply B_Here.
  SCase "B_Next".
  contradiction H3.
  reflexivity.
Case "x ≠ x'".
inversion x'_B_G1; subst.
  SCase "B_Here".
  apply B_Here.
  SCase "B_Next".
  inversion H5; subst.
  SSCase "B_Here".
  contradiction H3.
  reflexivity.
  SSCase "B_Next".
  apply B_Next.
  apply H4.
  apply H7.

```

Qed.

13.8 Permutation (TAPL 9.3.6)

Another result we need is the fact that in a context, two adjacent variable typings can switch places. It is a prerequisite, though, that the variables are distinct. The reason for this is that $\Gamma, x:S, x:T \vdash x : T$, but $\Gamma, x:T, x:S \vdash x : S$.

The proof is very similar to the one above, so we use automation to skip the boring parts.

```

Lemma permutation : ∀ Γ x y S U t T,
  x ≠ y →
  { {Γ, x : S}, y : U } % ctx ⊢ t : T →
  { {Γ, y : U}, x : S } % ctx ⊢ t : T.
Proof with eauto with bool_fun.
  intros Γ x y S U t T xNQy.
  apply context_invariance.
  intros x' T' x'_f_t2 x'_B_G1.
  inversion x'_B_G1; subst...

```

```

Case "B_Here".
  apply B_Next...
  red...
Case "B_Next".
  inversion H5...
Qed.

```

13.9 Type preservation under substitution (TAPL 9.3.8)

We now get to the most involved theorem so far, type preservation for substitution. Note that the theorem in TAPL is more general than is really needed. The languages we are considering are designed so that we only substitute *closed values* for variables, so we can assume that the term to be substituted can be typed in an empty context.

The most challenging part of the proof is following the operation of the substitution function. Note that when it encounters a variable or an abstraction, the substitution function compares variables for equality. We need to be able to consider both the case where the comparison is true and the case where it is false. Moreover, we need to know that when the comparison returns true, the compared terms are actually equal, and when it returns false, they are not.

Recall that we use just plain natural numbers as variable identifiers. The previous result about the decidability of their equality now comes handy. First, we use `beq_decides_eq_nat x v` to get $Q : \text{decides } (\text{beq_nat } x \ v) \ (x = v)$. Then we use the `destruct` tactic on `beq_nat x v`. This gives us two subgoals, one where `beq_nat x v` has been replaced with `true` and one where it has been replaced with `false`. Simplifying Q we now get $Q : x = v$ and $Q : x \neq v$ in the subgoals.

```

Theorem subst_preservation :  $\forall \Gamma \ x \ s \ t \ T,$ 
  { $\Gamma, x : S$ }  $\vdash t : T \rightarrow$ 
  {}  $\vdash s : S \rightarrow$ 
   $\Gamma \vdash [x \mapsto s]t : T.$ 

```

```

Proof with eauto with bool_fun.
  intros  $\Gamma \ x \ s \ t \ T \ t\_T \ s\_S.$ 

```

Once again it is crucial that we do a general enough induction: we do induction on terms on the general property that for *any* context Γ and type T such that $\Gamma \vdash t : T$, we have $\Gamma \vdash [x \mapsto s]t$. This allows us to use a different context when applying an inductive hypothesis.

```

induction t in  $\Gamma, T, t\_T \vdash *;$  simpl; inversion t\_T; subst...

```

```

Case "t = TmVar y, T_Var".
rename v into y.
assert (Q := beq_decides_eq_nat x y).
destruct (beq_nat x y); simpl in Q.

```

We are substituting in the variable x . Because the context ends with $x : S$, we know that the type T must be S . We also know that the substitution function replaces x with S so it remains to show that $\Gamma \vdash s : S$. This we get directly from $\vdash s : S$ with weakening.

```

SCase "x = y".
subst y.
inversion H1; subst.
  SSCase "B_Here".
  apply empty_weakening.
  apply s_S.
  SSCase "B_Next".
  contradiction H4.
  reflexivity.

```

We are substituting in some variable that is not x . In that case, the substitution function leaves the original term as it is. It remains to show that $\Gamma \vdash v : T$. This must be true, since $\Gamma, x : S \vdash T$, and $x \neq v$.

```

SCase "x ≠ y".
apply T_Var.
inversion H1; subst.
  SSCase "B_Here".
  contradiction Q.
  reflexivity.
  SSCase "B_Next".
  assumption.

```

```

Case "T_Abs, t = "y:T1, t2, T = T1 → T2".

```

(* Rename the variables to the ones used in TAPL. Note that for some reason TAPL uses $T = T2 \rightarrow T1$ here instead of $T1 \rightarrow T2$. *)

```

rename v into y, T2 into T1, t into T2, t0 into t1.
assert (Q := beq_decides_eq_nat x y).
destruct (beq_nat x y); simpl in Q.

```

We are substituting in an abstraction which binds x . In this case the substitution will leave it as it is and we have to show just that we can remove

the original x from the context and the term remains well-typed. This is done directly by the shadowed variable removal lemma.

```

SCase "x = y".
subst y.
apply T_Abs.
eapply shadowed_removable...

```

We are substituting in an abstraction which does not bind x . (This is the only case that is considered in TAPL.) After applying the T_Abs rule we need to prove

$$\Gamma, y:T2 \vdash [x \mapsto s]t1 : T1$$

We can then use the inductive hypothesis (which, thanks to its generality, can be given the extended context above), and are left with proving

$$\Gamma, y:T2, x:S \vdash t1 : T1$$

This can then be proven by the permutation lemma and our hypothesis (from considering the T_Abs case):

$$\Gamma, x:S, y:T2 \vdash t1 : T1.$$

```

SCase "x ≠ y".
apply T_Abs.
apply IHt.
apply permutation...

```

Qed.

13.10 Type preservation under evaluation (TAPL 9.3.9)

The actual type preservation theorem follows by a relatively simple induction on the derivation of the typing judgement, followed by an inversion on the evaluation judgement. The only interesting case is the one for the T_Abs , E_AppAbs , where we have to use the above substitution preservation lemma.

Theorem preservation : $\forall t t' T,$

$$\begin{aligned} &\vdash t : T \rightarrow \\ &t \longrightarrow t' \rightarrow \\ &\vdash t' : T. \end{aligned}$$

Proof.

Exercise *Admitted*.

End *typed_bool_fun*.

| | | |
|----------------------|---|--|
| <i>term, t</i> | $\begin{array}{l} ::= \\ (t) \quad M \\ x \\ \lambda x : T . t \quad \text{bind } x \text{ in } t \\ t_1 t_2 \\ [x \mapsto t'] t \quad M \end{array}$ | terms: variable abstraction application |
| <i>type, T, S, U</i> | $\begin{array}{l} ::= \\ (T) \quad M \\ \mathbf{A} \\ T \rightarrow U \end{array}$ | types: base type function type |

VAL *t*

$$\frac{}{\mathbf{VAL}(\lambda x : T . t)} \text{ VAL_ABS}$$

Figure 10: $\rightarrow \mathbf{A}$, syntax

$t \longrightarrow t'$ Evaluation

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \text{ E_APP1}$$

$$\frac{\mathbf{VAL} t_1 \quad t_2 \longrightarrow t'_2}{t_1 t_2 \longrightarrow t_1 t'_2} \text{ E_APP2}$$

$$\frac{\mathbf{VAL} t_2}{(\lambda x : T . t_{12}) t_2 \longrightarrow [x \mapsto t_2] t_{12}} \text{ E_APPABS}$$

Figure 11: $\rightarrow \mathbf{A}$, evaluation rules

$\boxed{\mathbf{FVS} tx}$

$$\begin{array}{c} \frac{}{\mathbf{FVS} xx} \text{ FVS_VAR} \\ \frac{x \langle \rangle y}{\mathbf{FVS} tx} \text{ FVS_ABS} \\ \frac{\mathbf{FVS} t_1 x}{\mathbf{FVS} (t_1 t_2) x} \text{ FVS_APP1} \\ \frac{\mathbf{FVS} t_2 x}{\mathbf{FVS} (t_1 t_2) x} \text{ FVS_APP2} \end{array}$$

Figure 12: $\rightarrow \mathbf{A}$, free variables

$\boxed{\Gamma \vdash t : T}$ Typing

$$\begin{array}{c} \frac{x : T \text{ in } \Gamma}{\Gamma \vdash x : T} \text{ T_VAR} \\ \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{ T_ABS} \\ \frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ T_APP} \end{array}$$

Figure 13: $\rightarrow \mathbf{A}$, typing rules

14 Normalization of simply typed lambda calculus (TAPL Chapter 12)

We now formalize the version of Tait’s normalization proof for simply typed lambda calculus that is described in TAPL Chapter 12. The language we study contains only functions and a single base type. The same proof technique could easily be extended to a language with e.g. booleans and numbers, but for simplicity, we here stick to a minimal language.

The developments below differ somewhat from those in TAPL, and the main technical reason for this is the fact that we have no convention that forces all variables to be distinct. This simplifies definitions, but forces us to occasionally check whether two variables are the same or not.

It is customary to present a proof from the bottom up, building complex proofs from simpler ones, and in Coq this is even a technical necessity. However, the actual practice of proving is often different: a number of the auxiliary lemmas below were proven only after noticing, in the middle of another proof, that the proof depends on a non-trivial property, which is better to prove separately.

Note that before the section “Substitution” most of the definitions and proofs are familiar from earlier chapters. They are only included because we are now dealing with a different language (one with just functions and a single base type A), so we have to prove the results again, even though the proof scripts are sometimes word-for-word identical.

Require Import *notations*.

Require Export *tapl_ch8*.

Require Export *tapl_ch9*.

Module *typed_base_fun*.

Require Export *typed_base_fun_lang*.

Hint Constructors *term val eval type typing and or ex bound fvs : base_fun*.

Hint Constructors *RSC : base_fun*.

Hint Unfold *not : base_fun*.

Notation “ $t1 \longrightarrow^* t2$ ” :=

(*RSC eval t1 t2*) (at level 80, no associativity) : *type_scope*.

Definition *closed t* := $\forall x, \sim(\text{fvs } t \ x)$.

Definition *normal_form t* := $\sim\text{exists } t', t \longrightarrow t'$.

Hint Unfold *normal_form : base_fun*.

Definition *well_typed* ($t : term$) : Prop :=
 $\exists T, \vdash t : T$.

Definition *halts* ($t : term$) : Prop :=
 $\exists t', t \longrightarrow^* t' \wedge normal_form\ t'$.

Hint Unfold *halts*.

Inductive *fv* : $term \rightarrow Prop$:=
 $fv_abs\ x\ T\ t : fv\ ("x:T,t)\%tm$.

Hint Constructors *fv* : *base_fun*.

Lemma *fv_is_val* $t : fv\ t \rightarrow val\ t$.

Proof.

intros $t\ fv_t$.
destruct fv_t .
apply *Val_Abs*.

Qed.

14.1 Canonical forms

Lemma *canonical_forms_fun* : $\forall v\ T1\ T2, val\ v \rightarrow \vdash v : (T1 \rightarrow T2) \rightarrow fv\ v$.

Proof.

intros $v\ T1\ T2\ val_v\ v_F$.
induction val_v ; inversion v_F ; subst;
try inversion H ; eauto with *base_fun*.

Qed.

Hint Resolve *canonical_forms_fun* : *base_fun*.

14.2 Progress

Theorem *progress* $t : well_typed\ t \rightarrow val\ t \vee \exists t', t \longrightarrow t'$.

Proof with eauto with *base_fun*.

intros $t\ [T\ t_T]$.
remember $\{\}\%ctx$ as Γ .
induction t_T ; subst Γ ...
Case "*T_Var* $\{\} \times T\ H$ ".
(* Impossible to have a bound variable in an empty context. *)

inversion H .

Case "*T_App* $\{\} t1\ t2\ T12\ T11\ t_T1\ t_T2$ ".

```

destruct IHt_T1 as [val_t1 | [t1' t1Et1']]...
  SCase "val t1".
    destruct IHt_T2 as [val_t2 | [t2' t2Et2']]...
      SSCase "val t2".
        elimtype (fv t1)...

```

Qed.

Corollary *normal_is_value* $t : \text{well_typed } t \rightarrow \text{normal_form } t \rightarrow \text{val } t$.

Proof.

```

intros t wf_t nf_t.
destruct (progress t wf_t).
  trivial.
  contradiction H.

```

Qed.

Lemma *val_is_normal* : $\forall t, \text{val } t \rightarrow \text{normal_form } t$.

Proof with eauto with *base_fun*.

```

intros t vt.
destruct vt.
intros [t' tEt'].
inversion tEt'.

```

Qed.

Hint Resolve *val_is_normal* : *base_fun*.

14.2.1 Evaluation is deterministic

Lemma *eval_deterministic* : $\forall t t' t'', t \longrightarrow t' \rightarrow t \longrightarrow t'' \rightarrow t' = t''$.

Proof with eauto with *base_fun*.

```

intros t t' t'' tEt' tEt''.
induction tEt' in t'', tEt'' ⊢ *; inversion tEt''; subst...
  rewrite → (IHtEt' t1'0)...
  contradiction (val_is_normal t1)...
  inversion tEt'.
  contradiction (val_is_normal t1)...
  rewrite → (IHtEt' t2'0)...
  contradiction (val_is_normal t2)...
  inversion H3.
  contradiction (val_is_normal t2)...

```

Qed.

14.2.2 Only the types of the free variables of a term affect its typing

Lemma *context_invariance* :

```
  ∀ Γ Γ' t T,  
  (∀ x T', fos t x → bound x T' Γ → bound x T' Γ') →  
  Γ ⊢ t : T →  
  Γ' ⊢ t : T.
```

Proof with eauto 20 with *base_fun*.

```
  intros Γ Γ' t T H G_t_T.  
  induction G_t_T in Γ', H ⊢ *...  
  Case "T_Abs".  
  apply T_Abs.  
  apply IHG_t_T.  
  intros x' T' x'_f_t2 B.  
  inversion B...
```

Qed.

A well-typed term has its type in any context

Lemma *empty_weakening* : $\forall \Gamma t T, \vdash t : T \rightarrow \Gamma \vdash t : T$.

Proof with eauto with *base_fun*.

```
  intros Γ t T t_T.  
  apply (context_invariance {})...  
  intros x T' x_f_t x_B_E.  
  inversion x_B_E.
```

Qed.

14.2.3 Substitution preserves typing

Theorem *subst_preservation* : $\forall \Gamma x S s t T$,

```
  {Γ, x : S} ⊢ t : T →  
  {} ⊢ s : S →  
  Γ ⊢ [x ↦ s]t : T.
```

Proof with eauto with *base_fun*.

```
  intros Γ x S s t T t_T s_S.  
  induction t in Γ, T, t_T ⊢ *; simpl; inversion t_T; subst...  
  Case "T_Var".  
  assert (Q := beq_decides_eq_nat x v).  
  destruct (beq_nat x v); simpl in Q.  
  SCase "x = v".  
  subst v.
```

```

inversion H1; subst.
  SSCase "B_Here".
  apply empty_weakening.
  apply s_S.
  SSCase "B_Next".
  contradiction H4.
  reflexivity.
SCase "x ≠ v".
  apply T_Var.
  inversion H1; subst.
  SSCase "B_Here".
  contradiction Q.
  reflexivity.
  SSCase "B_Next".
  assumption.
Case "T_Abs".
assert (Q := beq_decides_eq_nat x v).
destruct (beq_nat x v); simpl in Q.
  SCASE "x = v".
  subst v.
  apply T_Abs.
  apply (context_invariance {{Γ, x : S}, x : t}%ctx)...
  intros x' T' x'_f_t2 x'_B_G1.
  destruct (nat_eq_dec x x') as [xQx' | nxQx'].
  SSCase "x = x'".
  subst x'.
  inversion x'_B_G1; subst...
  contradiction H3...
  SSCase "x ≠ x'".
  inversion x'_B_G1; subst...
  inversion H6; subst...
  contradiction H3...
SCase "x ≠ v".
  apply T_Abs.
  apply IHt.
  apply (context_invariance {{Γ, x : S}, v : t}%ctx)...
  intros x' T' x'_f_t2 x'_B_G1.
  inversion x'_B_G1; subst.

```

```

SSCase "B_Here".
apply B_Next...
SSCase "B_Next".
inversion H6; subst...

```

Qed.

14.2.4 Evaluation preserves typing

Theorem *preservation* : $\forall t t' T,$
 $\vdash t : T \rightarrow$
 $t \longrightarrow t' \rightarrow$
 $\vdash t' : T.$

Proof.

Exercise *Admitted*.

14.2.5 Multi-step evaluation preserves typing

Theorem *meval_preservation* : $\forall t t' T,$
 $\vdash t : T \rightarrow$
 $t \longrightarrow^* t' \rightarrow$
 $\vdash t' : T.$

Proof.

```

intros t t' T tT tMt'.
induction tMt'; eauto using preservation.

```

Qed.

14.3 A free variable of a typable term must be bound in the context

Lemma *free_in_context* : $\forall \Gamma t T x,$
 $(\Gamma \vdash t : T) \rightarrow fvs\ t\ x \rightarrow \exists T', bound\ x\ T'\ \Gamma.$

Proof.

```

intros  $\Gamma$  t T x G_t_T x_f_t.
induction G_t_T; inversion x_f_t; subst; auto.
Case "T_Var".
  SCase "FVS_Var".
   $\exists T.$ 
  apply H.
Case "T_Abs".

```

```

SCase "FVS_Abs".
destruct (IHG_t_T H4) as [T' B].
∃ T'.
inversion B; subst.
  SSCase "B_Here".
  contradiction H3.
  reflexivity.

  SSCase "B_Next".
  assumption.

```

Qed.

Hint Unfold *well_typed* : *base_fun*.

14.4 A well-typed term is closed

Corollary *well_typed_is_closed* : $\forall t, \text{well_typed } t \rightarrow \text{closed } t$.

Proof.

```

intros t [T t_T].
red.
red.
intros x x_f_t.
destruct (free_in_context {} t T x t_T x_f_t) as [T' B].
inversion B.

```

Qed.

Hint Resolve *well_typed_is_closed* : *base_fun*.

14.5 Substitution

The proof of the normalization of the simply-typed lambda calculus depends heavily on various basic properties of the substitution operation $[x \mapsto t']t$ which replaces every free occurrence of x in t with t' . In informal proofs, most of these properties are often considered so self-evident that they are not even worth mentioning. However, when we formalize our reasoning, we often find that various supposedly self-evident facts require a substantial amount of proving. Thankfully, the proofs are usually reasonably simple.

We next prove some basic lemmas about substitution. Almost all the proofs proceed by induction on the term in which we substitute. In the *TmVar* and *TmAbs* cases we have to take into account the equality or inequality of variables which is checked by the substitution function. The

beq_nat_eq_dec lemma allows us to easily check the two possible cases.

14.5.1 Substituting for a non-free variable has no effect

The substitution function has been designed so that it substitutes only in the *free* occurrences of x in t . If there are no free occurrences of x in t , the result of the substitution is the original term unchanged. This fact is proven below.

Lemma *closed_subst_nop* $t x t' : \sim(\text{fvs } t \ x) \rightarrow ([x \mapsto t']t)\%tm = t$.

Proof with *eauto* with *base_fun*.

`intros t x t' nx_f_t.`

`induction t as [y | y T t'' IHt'' | t1 IHt1 t2 IHt2]; simpl.`

`Case "t = 'y'".`

`destruct (beq_nat_eq_dec x y) as [[H Q] | [H Q]]...`

If $x = y$, it contradicts our assumption that x does not appear free in t .

`SCase "x = y".`

`contradiction nx_f_t.`

`rewrite → Q...`

If $x \neq y$, the substitution produces the original term directly.

`SCase "x ≠ y".`

`rewrite → H...`

`Case "t = 'y:T, t'''".`

`destruct (beq_nat_eq_dec x y) as [[H Q] | [H Q]]...`

If $x = y$ then substitution produces the original term directly.

`SCase "x = y".`

`rewrite → H...`

If $x \neq y$ then we apply the inductive hypothesis. Even though y may appear free in t'' , we know that $x \neq y$ so $\sim(\text{fvs } x \ t'')$ is proved easily by automation.

`SCase "x ≠ y".`

`rewrite → H.`

`rewrite → IHt''...`

The *TmApp* case is proved directly with the inductive hypotheses.

`Case "t = t1 $ t2".`

`rewrite → IHt1...`

`rewrite → IHt2...`

Qed.

14.5.2 Repeated substitution of the same variable has no additional effect

If we substitute $[x \mapsto s]t$, and s is closed, then the result of this substitution cannot contain any free occurrences of x . Hence if we then do another substitution of x , it will have no further effect.

Lemma *subst_idem* $\{t\ x\ s\ s'\} : \text{closed } s \rightarrow$
 $([x \mapsto s']([x \mapsto s]t))\%tm = ([x \mapsto s]t)\%tm.$

Proof.

```
intros t x s s' clo_s.
induction t; simpl.
Case "TmVar".
destruct (beq_nat_eq_dec x v) as [[Q H] | [Q H]].
  SCase "x = v".
  rewrite → Q.
  auto using closed_subst_nop.
```

Here we see a reason why we could not simply do `destruct (beq_nat x v)`: the same equality test is done *twice* by each of the substitutions. If we did `destruct` for both cases, we would end up considering cases where `beq_nat x v` returned *true* at one time, and *false* at another. So instead, we have checked the result once and for all and stored it in an equality, and then use it to rewrite uniformly all instances of `beq_nat x v` that we meet.

```
  SCase "x ≠ v".
  rewrite → Q.
  simpl.
  rewrite → Q.
  reflexivity.
Case "TmAbs".
destruct (beq_nat_eq_dec x v) as [[Q H] | [Q H]].
  SCase "x = v".
  rewrite → Q.
  simpl.
  rewrite → Q.
  reflexivity.
  SCase "x ≠ v".
  rewrite → Q.
  simpl.
  rewrite → Q.
  rewrite → IHt.
```

```

    reflexivity.
  Case "TmApp".
  rewrite → IHt1.
  rewrite → IHt2.
  reflexivity.
Qed.

```

14.5.3 Substitution of distinct variables commutes

Suppose we substitute $[x \mapsto s][x' \mapsto s']t$, when $x \neq x'$ and s and s' are both closed. Then both of these substitutions are independent of each other: one replaces the free occurrences of x , and the other replaces the free occurrences of x' . Because s and s' are both closed, they don't contain free occurrences of x' or x , so one substitution cannot produce any subterms which would be affected by the other substitution.

Lemma *beq_nat_refl_false* $x x' : x \neq x' \rightarrow \text{beq_nat } x x' = \text{false}$.

Proof.

```

  intros x x' xNQx'.
  assert (Q' := beq_decides_eq_nat x x').
  destruct (beq_nat x x'); simpl in Q'...
  contradiction xNQx'.
  reflexivity.

```

Qed.

Lemma *subst_exchange* $x s x' s' t :$

```

  x ≠ x' → closed s → closed s' →
  ([x ↦ s]([x' ↦ s'] t))%tm = ([x' ↦ s']([x ↦ s] t))%tm.

```

Proof with auto.

```

  intros x s x' s' t xNQx' clo_s clo_s'.
  assert (beq_nat x x = false) as H1.
  auto using beq_nat_refl_false with base_fun.
  assert (beq_nat x x' = false) as H2.
  auto using beq_nat_refl_false.
  induction t; simpl.
  Case "TmVar".
  assert (Q := beq_decides_eq_nat x v).
  remember (beq_nat x v) as E.
  destruct E; simpl in Q ⊢ *.
  subst v.
  rewrite → H1.

```

```

simpl.
rewrite ← HeqE.
rewrite → closed_subst_nop...
assert (Q' := beq_decides_eq_nat x' v).
remember (beq_nat x' v) as E'.
destruct E'; simpl in Q ⊢ *.
  rewrite → closed_subst_nop...
  rewrite ← HeqE...

```

Case "TmAbs".

```

assert (Q := beq_decides_eq_nat x' v).
remember (beq_nat x' v) as E.
destruct E; simpl in Q ⊢ *.
  subst v.
  rewrite → H2.
  simpl.
  rewrite ← HeqE.
  reflexivity.
  assert (Q' := beq_decides_eq_nat x v).
  remember (beq_nat x v) as E'.
  destruct E'; simpl in Q ⊢ *.
    rewrite ← HeqE.
    reflexivity.
    rewrite ← HeqE.
    rewrite → IHt.
    reflexivity.

```

Case "TmApp".

```

rewrite → IHt1.
rewrite → IHt2.
reflexivity.

```

Qed.

14.6 Evaluation

Next, two basic facts about evaluation and halting.

14.6.1 Evaluation preserves halting

Lemma *eval_preserves_halts* : $\forall t t', t \longrightarrow t' \rightarrow \text{halts } t \rightarrow \text{halts } t'$.

Proof with `eauto` with `base_fun`.

Exercise *Admitted*.

14.6.2 Evaluation reflects halting

Lemma `eval_reflects_halts` : $\forall t t', t \longrightarrow t' \rightarrow \text{halts } t' \rightarrow \text{halts } t$.

Proof with `eauto` with `base_fun`.

```
intros t t' tEt' [t'' [t'Et'' nft'']]...
```

Qed.

14.6.3 Multi-step evaluation is congruent in the argument position

We are going to need this particular congruence result later on.

Lemma `meval_app2_congr` : $\forall t1 t2 t2'$,

```
val t1 →
```

```
t2 →* t2' → (t1 $ t2)%tm →* (t1 $ t2')%tm.
```

Proof.

```
intros val_t1 t1 t2 t2' t2Mt2'.
```

```
induction t2Mt2'; eauto with base_fun.
```

Qed.

14.7 Recursive halting (TAPL 12.1.2)

After the above preliminary work, we are finally ready to begin the actual normalization proof. The first stage is defining the recursive halting predicate, $R_T(t)$ in TAPL. This formally no different from our previous recursive definitions, except that this time the our recursive function produces a *proposition* instead of a value.

Notice that in TAPL the predicate R_T is defined on "closed terms of type T ". We could include these constraints in the propositions produced by the definition below, or even in the domain of the function, but this would complicate the formalization a little. Instead, we simply add additional constraints for closedness or well-typedness wherever they are needed. This accounts for some of the slight differences between the formalization below and the definition in TAPL.

Fixpoint `rhalts` ($T : \text{type}$) ($t : \text{term}$) : Prop :=

```
match T with
| A%ty ⇒ halts t
| (T1 → T2)%ty ⇒
```

```

    halts t ∧ ∀ s, ⊢ s : T1 → rhalts T1 s → rhalts T2 (t $ s)
end.

```

Hint Unfold *rhalts*.

14.7.1 Recursively halting terms are halting (TAPL 12.1.3)

Lemma *rhalts_then_halts* $T t : rhalts T t \rightarrow halts t$.

Proof.

```

intros T t rTt.
destruct T.
exact rTt.

destruct rTt as [ht _].
exact ht.

```

Qed.

14.7.2 Evaluation preserves and reflects recursive halting (TAPL 12.1.4)

These basic facts are straightforward to prove. Note that it is usually more practical to prove and use each direction of an equivalence as a separate implication lemma. (Sometimes, however, we need a bidirectional equivalence as an inductive hypothesis, so both directions have to be proven simultaneously. That is not the case here, though.)

Lemma *eval_preserves_rhalts* $T t t' :$

$t \longrightarrow t' \rightarrow rhalts T t \rightarrow rhalts T t'$.

Proof with `eauto 10` with `base_fun`.

```

intros T t t' tEt'.
induction T in t, t', tEt' ⊢ *; simpl.
  apply eval_preserves_rhalts...
  intros [ht rt].
  split.
    eapply eval_preserves_rhalts...
    intros s sT1 rT1s.
    eapply IHT2...

```

Qed.

Corollary *meval_preserves_rhalts* $T t t' :$

$t \longrightarrow^* t' \rightarrow rhalts T t \rightarrow rhalts T t'$.

Proof.

```

intros T t t' tMt'.

```

```

induction tMt'; eauto using eval_preserves_rhalts.
Qed.

Lemma eval_reflects_rhalts T t t' :
  t → t' → rhalts T t' → rhalts T t.
Proof with eauto 10 with base_fun.
  intros T t t' tEt'.
  induction T in t, t', tEt' ⊢ *; simpl.
  apply eval_reflects_halts...
  intros [ht' rt'].
  split.
  eapply eval_reflects_halts...
  intros s sT1 rT1s.
  eapply IHT2...

```

Qed.

Corollary meval_reflects_rhalts T t t' :

$t \longrightarrow^* t' \rightarrow rhalts T t' \rightarrow rhalts T t.$

Proof.

```

intros T t t' tMt'.
induction tMt'; eauto using eval_reflects_rhalts.

```

Qed.

14.8 Substitution environments

We now get to the most complicated part of our proof developments. In principle, we would like to show that $\vdash t : T$ implies $rhalts T t$. However, this only makes sense for closed terms. If we try to prove this by induction on the typing derivation (or on the structure of t), then we eventually encounter the open terms that are the bodies of lambda abstractions. The question is: what properties do we want open terms to have in order to prove that closed terms halt?

A direct consequence of our evaluation rules is that if we begin evaluating a closed term t , then it remains closed during every step of the evaluation, i.e. every t' is closed when $t \longrightarrow^* t'$. In particular, every open subterm (and those are always within the bodies of lambda abstractions) has all its free variables replaced by closed terms before it is evaluated.

So the relevant question about the evaluation of open terms is what happens to the result of substituting the free variables of a term with closed terms.

More precisely, if Γ is a context $x1:T1, x2:T2\dots$, and we have $\Gamma \vdash t : T$, the question is what happens to $[x1 \mapsto v1][x2 \mapsto v2]\dots t$, where the values vn are closed and have $\vdash vn : Tn$. Moreover, since we want to prove recursive halting by induction, we can as an inductive hypothesis assume that we have $\text{rhalt} Tn vn$.

Our first step is to formalize this sequence $v1, v2\dots$, which should have the above properties relative to a context Γ . We firstly define an *environment* as simply a list of terms:

```
Inductive env :=
| env_empty
| env_extend : env → term → env
.
```

```
Delimit Scope env_scope with env.
Bind Scope env_scope with env.
```

14.8.1 Typing of an environment with respect to a typing context

We then need to define when exactly an environment E has the desired properties relative to a context Γ . That is, E should have the same length as Γ and its elements should be closed recursively halting values that have the type of the corresponding variable in Γ . We define this correspondence inductively:

```
Inductive env_typing : env → ctx → Prop :=
| ET_empty : env_typing env_empty {}
| ET_extend E Γ x T v :
  ⊢ v : T → val v → rhalt T v → env_typing E Γ →
  env_typing (env_extend E v) {Γ, x : T}
.
```

With a slight abuse of notation, we write $\models E : \Gamma$ to mean that the terms in E are halting values that have the types of the corresponding variables in Γ .

```
Notation "⊨ E : G" := (env_typing E Γ)
  (at level 70, E at next level, no associativity).
```

14.8.2 Substitution of an environment in a term

Finally, we need to define what it means to substitute the variables of a context Γ with the values in an environment E . We proceed recursively,

traversing both the environment and the context, and substituting the variables from right to left.

We have to define the result for the case where the environment and the context have different lengths, but since we will only deal with environments that are well-typed relative to the context, this erroneous situation will never occur.

```
Fixpoint env_subst Γ E t :=
  match E, Γ with
  | env_empty, {}%ctx => t
  | env_extend E' v, {Γ', x:T}%ctx => env_subst Γ' E' ([x ↦ v]t)%tm
  | -, _ => t (* impossible when ⊢ E : Γ *)
  end.
```

Again, we abuse notation and write $[\Gamma \Rightarrow E]t$ for the result of substituting every variable of Γ in t with the corresponding value in E .

Notation " $[\Gamma \Rightarrow E] t' := (env_subst \Gamma E t)$
(at level 7, left associativity) : *term_scope*.

14.8.3 E-Substitution in a closed term has no effect

We then define some basic properties of environment substitution. These are mostly direct corollaries of the similar properties of plain single substitution.

Lemma *closed_env_subst_nop* : $\forall E \Gamma t,$
closed $t \rightarrow \vdash E : \Gamma \rightarrow ([\Gamma \Rightarrow E]t)\%tm = t.$

Proof with *eauto* with *base_fun*.

```
intros E Γ t clo_t E_G.
induction E_G; simpl...
rewrite → closed_subst_nop...
```

Qed.

14.8.4 E-substitution distributes over application

Lemma *env_subst_distr_app* : $\forall E \Gamma t t',$
 $\vdash E : \Gamma \rightarrow$
 $([\Gamma \Rightarrow E](t \$ t'))\%tm = ([\Gamma \Rightarrow E]t \$ [\Gamma \Rightarrow E]t')\%tm.$

Proof.

```
intros E Γ t t' E_G.
induction E_G in t, t' ⊢ *; simpl.
reflexivity.
```

apply *IHE_G*.

Qed.

14.8.5 E-substitution in an abstraction produces an abstraction

Lemma *fv_env_subst_fv* : $\forall E \Gamma t,$
 $\models E : \Gamma \rightarrow fv\ t \rightarrow fv\ ([\Gamma \Rightarrow E]t).$

Proof with eauto with *base_fun*.

intros *E* Γ *t* *E_G* *fv_t*.
destruct *fv_t* as [*x* *T* *t'*].
induction *E_G* in *t'* \vdash *; simpl...
destruct (*beq_nat* *x0* *x*)...

Qed.

14.8.6 E-substitution preserves typing

Lemma *env_subst_preservation* : $\forall E \Gamma t T,$

$\models E : \Gamma \rightarrow$
 $\Gamma \vdash t : T \rightarrow$
 $\vdash [\Gamma \Rightarrow E]t : T.$

Proof with eauto with *base_fun*.

intros *E* Γ *t* *T* *E_G* *G_t_T*.
induction *E_G* in *t*, *T*, *G_t_T* \vdash *; simpl.
apply *G_t_T*.
apply *IHE_G*.
eapply *subst_preservation*.
apply *G_t_T*.
apply *H*.

Qed.

14.8.7 Beta reduction extends substitution environment

This result is somewhat specific to our needs, but it is simpler to prove than prettier, more general-purpose lemmas.

What we want to show here is that when we are evaluating an application of a lambda abstraction “ $x:T, t$ ” to a value v , and the lambda abstraction has closed by an E -substitution, then the result of the application is the body t of the abstraction, but closed by an *extended* substitution, where we add v to the environment and $x:T$ to the context.

Intuitively this makes directly sense, because $(\text{"x:T, t}) v \longrightarrow [x \mapsto v]t$. In practice things are complicated a little by the fact that we must check if x is already in the context or not.

Lemma *beta_ext_env_subst* : $\forall E \Gamma x T t v,$
 $\models E : \Gamma \rightarrow$
 $\vdash v : T \rightarrow$
 $\text{val } v \rightarrow$
 $([\Gamma \Rightarrow E](\text{"x : T, t})) \$ v \longrightarrow [\{\Gamma, x : T\} \Rightarrow \text{env_extend } E v]t.$

Proof with *eauto* with *base_fun*.

```

intros E Γ x T t v E_G v_T val_v.
induction E_G as [| E' Γ' x' T' v' v'_T' val_v' rT'v' E'_G' IHE'_G']
  in t ⊢ *; simpl.
Case "E = env_empty, G = {}".
auto using E_AppAbs.
Case "E = env_extend E' v', G = {G', x' : T'}".
simpl in IHE'_G'.
assert (Q := beq_decides_eq_nat x' x).
destruct (beq_nat x' x); simpl in Q.
  SCase "x' = x".
  subst x'.
  rewrite → subst_idem.
  apply IHE'_G'.
  apply well_typed_is_closed...
  SCase "x' ≠ x".
  rewrite → subst_exchange.
  apply IHE'_G'.
  apply Q.
  apply well_typed_is_closed...
  apply well_typed_is_closed...

```

Qed.

14.9 E-Substitution preserves halting (TAPL 12.1.5)

We now get to the main theorem: *every* term t that has the type T in some context Γ , will recursively halt when its free variables are substituted with recursively halting values that respect the types of Γ .

We consider the case where t is a variable first, since it does not require any inductive hypotheses. The variable case is described as "immediate"

in TAPL, but because we do not require all variables to be distinct (as is done in TAPL), we have to consider a number of cases and eliminate the impossible ones.

Intuitively, however, the proof is simple: if $x:T$ is bound in Γ , and E respects the types of Γ , then at some point during the E -substitution we must replace x by some recursively halting value v of type T . Since v is well-typed, it is closed, and therefore none of the remaining substitutions in E can affect it.

```
Lemma env_subst_var_rhalts :  $\forall E \Gamma x T,$ 
   $\models E : \Gamma \rightarrow$ 
  bound  $x T \Gamma \rightarrow$ 
  rhalts  $T [\Gamma \mapsto E]x$ .
```

Proof with eauto with base_fun.

```
intros E  $\Gamma x T E_G x_T_G$ .
induction E_G as [| E'  $\Gamma' x' T' v' v'_T' val_v' rT'v' E'_G' IHE'_G'$ ]; simpl.
  Case "E = env_empty, G = {}".
    inversion  $x_T_G$ . (* Impossible *)
  Case "E = env_extend E' v', G = {G', x' : T}'".
    assert (Q := beq_decides_eq_nat x' x).
    destruct (beq_nat x' x); simpl in Q.
      SCase "x' = x".
        subst x'.
        inversion  $x_T_G$ ; subst.
          SSCase "B_Here, T = T', x = x'".
            rewrite  $\rightarrow$  closed_env_subst_nop...
          SSCase "B_Next, x  $\neq$  x, bound x T G'".
            contradiction H3...
      SCase "x'  $\neq$  x".
        apply IHE'_G'.
        inversion  $x_T_G$ ; subst.
          SSCase "B_Here, T = T', x = x'".
            contradiction Q...
          SSCase "B_Next, x  $\neq$  x', bound x T G'".
            apply H5.
```

Qed.

We then move to the main theorem. The interesting case is T_Abs , which requires an innovative use of the induction hypothesis.

```
Lemma env_subst_rhalts :  $\forall E \Gamma t T,$ 
```

```

 $\models E : \Gamma \rightarrow$ 
 $\Gamma \vdash t : T \rightarrow$ 
rhalts T [ $\Gamma \Rightarrow E$ ]t.

```

Proof with *eauto*.

```

intros E  $\Gamma$  t T E_G G_t_T.
induction G_t_T in E, E_G  $\vdash$  *; simpl.
  Case "T_Var, t = 'x'".
  apply env_subst_var_rhalts...
  Case "T_Abs, T = T1  $\rightarrow$  T2, t = "x:T1, t2'".
  split.

```

We first prove that the result of the substitution halts. We use the auxiliary result that the result of the substitution is an abstraction, so it is immediately a value and thus a normal form.

```

SSCase "halts ([G  $\Rightarrow$  E]t)".
 $\exists$  ([ $\Gamma \Rightarrow E$ ]("x:T1, t2"))%tm.
split.
  SSSCase "t  $\rightarrow^*$  t'".
  apply rsc_refl.
  SSSCase "normal_form ([G  $\Rightarrow$  E]t)".
  apply val_is_normal.
  apply fv_is_val.
  apply fv_env_subst_fv...
  apply fv_abs.

```

Here begins the most interesting part of the proof. We have to show that given a recursively halting term s of type $T1$, then $[\Gamma \Rightarrow E]t \ \$ \ s$ is also recursively halting.

Because the following proof uses practically all of the lemmas we have defined previously, and is perhaps hard to decipher, we write it unusually in the forward reasoning style, writing out explicitly the intermediate results by which the final goal is proven. This is more verbose than the standard backward reasoning style of Coq, and hopefully easier to follow.

Notice that we are not using any hint databases: when a tactic ends with ellipses (*apply foo...*), it means that the required arguments to *foo* can be found from our current assumptions.

```

intros s sT1 rT1s.
assert (halts s) as [v [sMv nfv]].
  eapply rhalts_then_halts...
assert ( $\vdash v : T1$ ).

```

```

    eapply meval_preservation...
  assert (rhalt T1 v).
    eapply meval_preserves_rhalts...
  assert (val v).
    eapply normal_is_value...
    ∃ T1...
  pose (Γ' := {Γ, x : T1}%ctx).
  pose (E' := env_extend E v).
  assert (⊨ E' : Γ').
    apply ET_extend...
  assert (rhalt T2 [Γ' ⊨ E']t2).
    apply IHG_t_T...
  assert (([Γ ⊨ E]("x : T1, t2) $ s)%tm →* ([Γ' ⊨ E']t2)%tm).
    eapply rsc_trans.
      apply meval_app2_congr.
        apply fv_is_val.
          apply fv_env_subst_fv...
            apply fv_abs.
              apply sMv.
            eapply rsc_step.
              apply beta_ext_env_subst...
                apply rsc_refl.
              assert (rhalt T2 ([Γ ⊨ E]("x:T1, t2) $ s)).
                eapply meval_reflects_rhalts...
              assumption.

```

Compared to abstractions, the application case is relatively straightforward.

```

Case "T_App, T = T11 → T12, t = t1 $ t2".
rewrite → env_subst_distr_app...
assert (rhalt (T11 → T12) [Γ ⊨ E]t1) as [h1 fr1].
  apply IHG_t_T1...
  fold rhalt in fr1. (* just to make the context prettier *)
  apply fr1.
    apply env_subst_preservation...
  apply IHG_t_T2...

```

Qed.

14.10 Normalization (TAPL 12.1.6)

Corollary *normalization* $t T : \vdash t : T \rightarrow \text{halts } t$.

Proof.

intros $t T t_T$.

eapply *rhalts_then_halts*.

apply (*env_subst_rhalts env_empty* {} $t T$).

apply *ET_empty*.

assumption.

Qed.

End *typed_base_fun*.