3 Proving Equalities

In this chapter we concentrate on *equational proof*, one of the most common and useful types of inference. Our first examples will involve simple functions on natural numbers, like addition and multiplication. Later in the chapter we will work with list functions like concatenation and reversal. For both natural numbers and lists, the simplest examples involve only equality chaining, but for proofs of more interesting and useful properties we need some form of induction, corresponding to the way in which natural numbers and lists are defined as datatypes. Such an induction proof divides into cases, but in all of the examples and exercises in this chapter, each proof case can be completed with equality chaining.

3.1 Numeric equations

As we already saw in Section 2.7, we can define natural numbers in Athena as follows:

```
datatype N := zero | (S N)
```

This definition says that the values of sort `N` are

```
zero
(S zero)
(S (S zero))
(S (S (S zero)))
...
```

In general, 0 is represented by `zero` and to obtain the natural number $n+1$ we apply the "successor function" `S` to the natural number n. Put another way, the natural number n is obtained by an n-fold application of `S` to `zero`. Furthermore—and this is a crucial point—we define the values so obtained to be *the only values* that are natural numbers.[1]

Recall from page 30 that terms without any variables, like those in the sequence above, are called *ground terms*. Terms containing variables, such as `(S ?n)`, are called *nonground* (or sometimes *general*) terms.

Suppose now we want to define and reason about the addition function `Plus` that takes two natural numbers as inputs and returns their sum. In Athena, we first **declare** the function, specifying the number and sorts of its inputs (also called *arguments*) and the sort of its output (or "return value"), as follows:

1 Roughly speaking, we are saying that any expression that purports to denote a natural number must have an equivalent expression just in terms of `S` and `zero`. Later in the chapter we shall see how this fundamental property is formalized, in conventional mathematical terms as well as in Athena.

```
declare Plus: [N N] -> N [+]
```

This declaration says that `Plus` takes two natural numbers as inputs and produces a natural number as output. The expression `[+]` at the end of the declaration overloads the built-in symbol `+` so that it can be used as an alias for `Plus` whenever the context allows it.

We can now write terms such as `(Plus (S zero) zero)`, or

`((S zero) Plus zero)` (3.1)

if we prefer infix notation. In fact, by making sure that `S` binds tighter (has greater precedence) than `Plus`, with a directive like:

```
set-precedence S 350
```

we can write term (3.1) even more simply as:

`(S zero Plus zero)`.

And since the declaration above has also overloaded the predefined operator `+` to designate `Plus` when applied to terms of sort `N`, we can also write (3.1) as

`(S zero + zero)`.

(Another way to permit the use of `+` for arguments of sort `N` is to declare our function using the identifier `+` in the first place, placing the declaration inside a module to avoid conflict with the predefined `+`. That's how it is done in the Athena library, and how it will be done in this book after the discussion of modules in Chapter 7.)

So far these are strictly matters of syntax; they do not say anything about the *meaning* of a term such as `(S zero + zero)` other than that its sort is `N` (and is thus one of the values `zero`, `(S zero)`, `(S S zero)`, ...[2]). At this point, for all we know, it could be any natural number. To give `Plus` the intended meaning, we will *define* it by asserting some appropriate universally quantified equations. But before we do that, it will be convenient to give names to a few variables of sort `N` so that we don't have to keep typing question marks:

```
define [x y z n m k] := [?x:N ?y:N ?z:N ?n:N ?m:N ?k:N]
```

As we saw in the previous chapter, variables are regular denotable values, so we can now use these names to refer to the corresponding variables:

```
> (x Plus S y)

Term: (Plus ?x:N
            (S ?y:N))
```

2 Recall that consecutive applications of a unary function symbol such as `S` need not be separated by parentheses, so we can write `(S S S zero)` rather than `(S (S (S zero)))`. We use this convention frequently.

We can even use the defined names as arguments to quantifiers:

```
> (forall n m . n Plus m = m Plus n)

Sentence: (forall ?n:N
            (forall ?m:N
              (= (Plus ?n:N ?m:N)
                 (Plus ?m:N ?n:N))))
```

Defining variables like that is a common practice that we will follow often in this book. (In fact, `x`, `y`, and `z` are already predefined at the Athena top level as the polymorphic variables `?x`, `?y`, and `?z`, respectively.)

We now introduce the following universally quantified equations:

```
assert right-zero := (forall n . n + zero = n)

assert right-nonzero := (forall n m . n + S m = S (n + m))
```

Since we introduced them with **`assert`**, these equations are also entered into the global assumption base.

Now, of course, the meaning of `(S zero + zero)` is determined by the equation that is just the special case of `right-zero` with the ground term `(S zero)` substituted for `n`. One way to say this in Athena, and thereby get the special-case equation into the assumption base, is

```
(!instance right-zero [(S zero)])
```

to which Athena responds:

```
Theorem: (= (Plus (S zero)
                  zero)
            (S zero))
```

That is, $1 + 0 = 1$. Likewise:

```
> (!instance right-nonzero [zero (S zero)])

Theorem: (= (Plus zero
                  (S (S zero)))
            (S (Plus zero
                     (S zero))))
```

In general, the first argument to `instance` is a universally quantified sentence p in the assumption base, and the second is a list L of terms.[3] If

$$p = (\texttt{forall}\ v_1 \cdots v_n\ .\ q)$$

3 For convenience, we can also give a term t by itself as the second argument to `instance`. That has the same effect as passing the one-element list `[`t`]` as the second argument.

and $L = [t_1 \cdots t_k]$, where $k \leq n$, then instance produces the sentence

$$(\texttt{forall } v_{k+1} \cdots v_n \texttt{ . } q')$$

where q' results from substituting t_i for v_i in q, $i = 1, \ldots, k$.[4] In the first case above, $n = k = 1$, and in the second, $n = k = 2$. In the following case $n = 2$ and $k = 1$, so the result still has one quantifier:

```
> (!instance right-nonzero [zero])

Theorem: (forall ?v303:N
           (= (Plus zero
                    (S ?v303:N))
              (S (Plus zero ?v303:N))))
```

3.2 Equality chaining preview

What about the meaning of Plus for larger ground term inputs, like

$$(\texttt{S S zero + S S zero})?$$

In other words, can we now deduce that $2 + 2 = 4$? Yes, and here is one way to do it:

```
(!chain [(S S zero + S S zero)
       = (S (S S zero + S zero))    [right-nonzero]
       = (S S (S S zero + zero))    [right-nonzero]
       = (S S S S zero)             [right-zero]
        ])
```

Here we have used chain, an Athena method[5] for proving equations by chaining together a sequence of terms connected by equalities. In general,

$$(\texttt{!chain [}t_0 \texttt{ = } t_1 \texttt{ [}p_1\texttt{] = } t_2 \texttt{ [}p_2\texttt{] = } \cdots \texttt{ = } t_n \texttt{ [}p_n\texttt{]])}$$

attempts to derive the identity $(t_0 = t_n)$, provided that each p_i is in the assumption base and each equation $(t_{i-1} = t_i)$ follows from p_i,[6] typically by virtue of one of the five fundamental axioms of equality listed in Section 3.4, for $i = 1, \ldots, n$. Here $n = 3$, the working

4 The substitutions are only for the *free* occurrences of the variables, namely, those not bound by quantifiers within q'. Moreover, the substitutions are carried out in a safe manner, so as to avoid variable capture. These points are explained more fully in Section 5.1.

5 This is not a primitive method; chain is defined in Athena's library.

6 Section 5.6 will define more precisely what we mean by "follows from" in the case of first-order logic with equality, and will also explicate the notion of an interpretation for first-order logic, but an intuitive understanding of these concepts will suffice for now.

assumptions used at each step are $p_1 = p_2 =$ `right-nonzero` and $p_3 =$ `right-zero`, and Athena responds with the theorem proved:

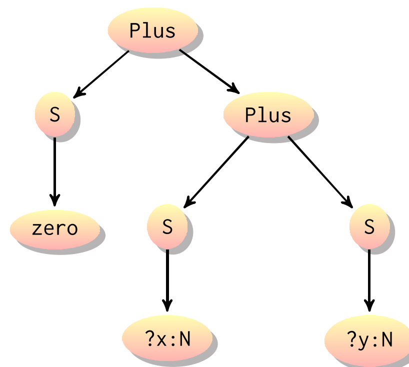
```
Theorem: (= (Plus (S (S zero))
                  (S (S zero)))
            (S (S (S (S zero)))))
```

We refer to each list `[`p_i`]` as the *justification list* for the corresponding step (from t_{i-1} to t_i), and to each p_i as a *justifier* for the step.

The interface of `chain` is actually a bit more flexible than the above description suggests. For example, multiple sentences may appear inside the square brackets (rather than a single p_i), and the structure of these sentences can be fairly complex (e.g., each sentence may be a *conditional* equation or, say, a conjunction, rather than a simple equation). But before examining how equality chaining works in general, we need to understand terms and sentences as tree structures.

3.3 Terms and sentences as trees

Terms and sentences are tree-structured objects, and in some cases it makes sense to treat them as trees explicitly. Let us start with terms. A variable or a constant symbol can be viewed as a simple one-node tree (a *leaf* node), while an application of the form $(f\ t_1 \cdots t_n)$ for $n > 0$ can be viewed as a tree with the symbol f at the root and with the trees corresponding to $t_1, \ldots, t_n$ as its immediate subtrees, arranged from left to right in that order. For instance, the tree corresponding to the term `(Plus (S zero) (Plus (S x) (S y)))` can be depicted as follows:



Thus, leaves stand for simple terms (variables and constants), while internal nodes represent compound terms—applications.

A two-dimensional tree representation of a compound term t depicts the essential syntactic structure of t, telling us exactly what function symbols are applied to what arguments and in what order, but without specifying how to write down t as a linear string. It does not tell us whether to use prefix, infix, or postfix notation; how to separate the arguments from one another (with commas, periods, spaces, indentation, etc.); what characters to use for grouping the arguments of a single application together (parentheses, square brackets, etc.); and so on. Such notational details are decided by choosing a particular *concrete syntax* for terms. The concrete syntax that Athena uses to output terms is the prefix notation common in Lisp dialects. The same prefix notation is available for input as well, but one can also use infix notation for binary function symbols, which often reduces notational clutter, especially in combination with precedence and associativity conventions. But tree representations, by dispensing with such details, are said to depict the *abstract syntax* of terms. We will have more to say about abstract syntax in Chapter 18.

Every node in the tree representation of a term can be assigned a unique list of positive integers $[i_1 \cdots i_m]$ indicating the path that must be traversed in order to get from the root of the tree to the node in question. That list represents the *position* of the node in the tree. As an example, Figure 3.1 shows the positions of all nodes in the tree representation of `(Plus (S zero) (Plus x y))`. As you can see, the position of the `S` node is [1], because we get to it by traveling down the first (leftmost) edge attached to the root; the position of `x` is [2 1], because we get to it by first visiting the second child of the root, and then moving to the first child of that node; and so on. The position of the root node is always the empty list [ ]. These integer sequences are sometimes called *Dewey paths* (or Dewey positions) because they are somewhat similar in their structure to the sequences of the Dewey decimal classification system used by libraries to organize book collections.

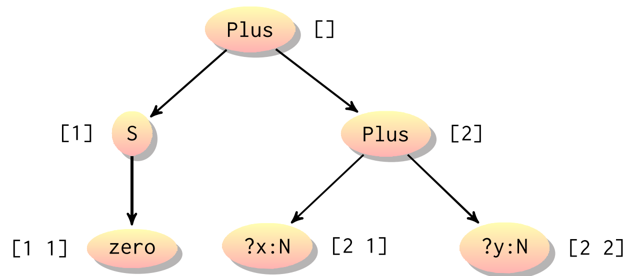


Figure 3.1
The term `(Plus (S zero) (Plus ?x ?y))` depicted as a tree structure. Nodes are annotated with their Dewey positions.

The procedure `positions-and-subterms` takes a term and produces a list of all positions in the term, each paired in a sublist with the subterm at that position. For example, for the term in Figure 3.1 we have:

```
> (positions-and-subterms (Plus (S zero) (Plus x y)))

List: [[[] (Plus (S zero) (Plus ?x ?y))]
       [[1] (S zero)]
       [[1 1] zero]
       [[2] (Plus ?x ?y)]
       [[2 1] ?x]
       [[2 2] ?y]]
```

We use this procedure in an exercise in the next section, and to help implement a basic form of equality chaining.

The following is a useful procedure that takes a term t and a position I (as a list of positive integers) and returns the subterm of t that is located at position I in the tree representation of t. An error will occur if there is no such subterm, that is, if I is not a valid position for t.

```
define (subterm t I) :=
  match I {
    [] => t
  | (list-of i rest) => (subterm (ith (children t) i)
                                 rest)
  }
```

The primitive binary procedure `ith` takes a list of $n > 0$ values `[`$V_1 \cdots V_n$`]` and an integer $i \in \{1, \ldots, n\}$ and returns V_i.

If we only want the node at a given position, we can use the following procedure:

```
define (subterm-node t I) := (root (subterm t I))

> (subterm-node (x + S S zero) [2 1])

Symbol: S
```

Another useful procedure is `replace-subterm`, where (`replace-subterm` t I t') returns the term obtained from t by replacing the subterm at position I by t', provided that the result is well-sorted. We leave the definition of this procedure as an exercise.

Similar ideas apply to sentences. An atomic sentence is just a term t, so its tree representation is that of t. A sentence of the form ($\circ$ $p_1 \cdots p_n$), for $\circ \in$ {`not`,`and`,`or`,`if`,`iff`}, can be viewed as a tree with the sentential constructor $\circ$ at the root and the trees corresponding to $p_1, \ldots, p_n$ as its immediate subtrees, listed from left to right in that order. Finally, a quantified sentence of the form (Q x p) can be viewed as a tree with the quantifier Q at the root, the sole leaf x as its left subtree, and the tree corresponding to p as its right subtree.

fpmics 2016/9/20 10:19 Page 120 #144

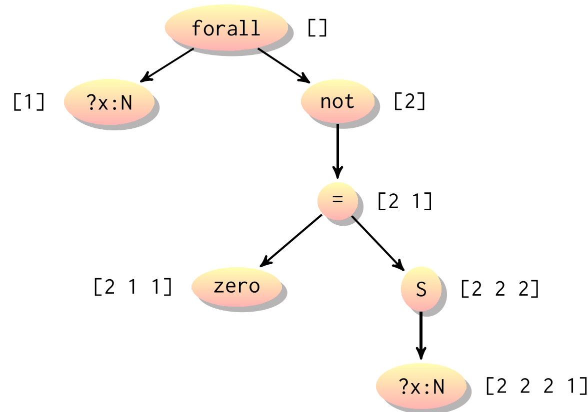


Figure 3.2
The sentence `(forall ?x (not (= zero (S ?x))))` depicted as a tree. Nodes are annotated with their respective positions.

Node positions are defined as they were for term trees. Thus, for example, the sentence `(forall x . zero =/= S x)`, which in prefix notation is written as

```
(forall x (not (= zero (S x)))),
```

is represented by the tree shown in Figure 3.2, whose nodes have been annotated with their corresponding positions. Procedures `subsentence`, `subsentence-node`, and `replace-subsentence` can be implemented analogously to `subterm`, `subterm-node`, and `replace-subterm`; see Exercise 3.37.

3.4 The logic behind equality chaining

A firm foundation for reasoning about equalities is provided by the basic *equality axioms*, which can be expressed in conventional notation as follows:

1. **Reflexivity:** $\forall\, x\,.\; x = x$.
2. **Symmetry:** $\forall\, x\; y\,.\; x = y \Rightarrow y = x$.
3. **Transitivity:** $\forall\, x\; y\; z\,.\; x = y \wedge y = z \Rightarrow x = z$.
4. **Functional Substitution:** For any function symbol f of n arguments,

$$\forall\, x_1 \cdots x_n\; y_1 \cdots y_n\,.\; x_1 = y_1 \wedge \ldots \wedge x_n = y_n \Rightarrow f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n).$$

5. **Relational Substitution:** For any relation symbol R of n arguments,

$$\forall\, x_1 \cdots x_n\, y_1 \cdots y_n\, .\; x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge R(x_1,\ldots,x_n) \Rightarrow R(y_1,\ldots,y_n).$$

Strictly speaking, the last two are *axiom schemas*, because they are indexed by f and R. Each schema has multiple axioms as instances, one for each function and relation symbol. Axiom schemas are captured in Athena with methods; we will present the relevant methods shortly. Because these are captured by methods, they are not sentences to be entered into the assumption base. Rather, any one of the first three axioms and any instance of the last two axiom schemas can be readily derived at any time by applying the corresponding method to appropriate arguments.

For simplicity, we will refer to all five of the above simply as "axioms" (rather than "axioms and axiom schemas"). Not all five are necessary, by the way. In fact, we can get away with reflexivity and a variation of the relational substitution axiom known as Leibniz's law. Symmetry, transitivity, and functional substitution would then become derivable as consequences. For practical purposes, however, it is more convenient to start out with all five.

Equational proofs in Athena are ultimately based on primitive methods corresponding to each of the five equality axioms, namely `reflex`, `sym`, `trans`, `fcong` ("functional congruence"), and `rcong` ("relational congruence"), as illustrated here:

```
domain D
declare a, b, c: D

> conclude (a = a)
    (!reflex a)

Theorem: (= a a)

> conclude (a = b ==> b = a)
    assume h := (a = b)
       (!sym h)

Theorem: (if (= a b)
             (= b a))

> conclude (a = b & b = c ==> a = c)
    assume (a = b & b = c)
      (!tran (a = b) (b = c))

Theorem: (if (and (= a b)
                  (= b c))
             (= a c))

declare f:[D D] -> D
declare R:[D D] -> Boolean
declare a1, a2, b1, b2: D
```

```
> conclude (a1 = b1 & a2 = b2 ==> (f a1 a2) = (f b1 b2))
    assume (a1 = b1 & a2 = b2)
      (!fcong ((f a1 a2) = (f b1 b2)))

Theorem: (if (and (= a1 b1)
                  (= a2 b2))
             (= (f a1 a2)
                (f b1 b2)))

> conclude (a1 = b1 & a2 = b2 & a1 R a2 ==> b1 R b2)
    assume (a1 = b1 & a2 = b2 & a1 R a2)
      (!rcong (a1 R a2) (b1 R b2))

Theorem: (if (and (= a1 b1)
                  (and (= a2 b2)
                       (R a1 a2)))
             (R b1 b2))
```

The following is a more precise specification of these methods:

- `reflex`: A unary method that takes an arbitrary term t and produces the sentence $(t = t)$, in any assumption base.
- `sym`: A unary method that takes an equality $(s = t)$ as an argument. If the sentence $(s = t)$ is in the assumption base, then the conclusion $(t = s)$ is produced. Otherwise an error occurs.
- `tran`: A binary method that takes two equalities of the form $(t_1 = t_2)$ and $(t_2 = t_3)$ as arguments. If both of these are in the assumption base, the conclusion $(t_1 = t_3)$ is produced. Otherwise an error occurs.
- `fcong`: A unary method that takes an equality p of the form

$$((f\ s_1 \cdots s_n) = (f\ t_1 \cdots t_n))$$

  as an argument. If the assumption base contains the n equalities $(s_i = t_i)$, $i = 1, \ldots, n$, then the input argument p (which represents the desired conclusion) is returned as the result. An error will occur if some $(s_i = t_i)$ is not in the assumption base and $s_i \neq t_i$.
- `rcong`: A binary method that takes two atoms of the form $(R\ s_1 \cdots s_n)$ and $(R\ t_1 \cdots t_n)$ as arguments, where R is a relation symbol of arity n. If the assumption base contains the first sentence, $(R\ s_1 \cdots s_n)$, along with the n equalities $(s_1 = t_1), \ldots, (s_n = t_n)$, then the second sentence $(R\ t_1 \cdots t_n)$ is produced, otherwise an error is generated.

It is not difficult to see that the given axioms are *true* under the conventional interpretation of the equality symbol as the identity relation, regardless of how we interpret other symbols. Therefore, these five methods are *sound*, meaning that they can never take us

from true premises to false conclusions. How about completeness? Are these methods sufficient for deriving *all* identities that follow from a given set of equations? In conjunction with `instance`, the answer is affirmative. More precisely, if we have a finite set E of universally quantified equations and are presented with a new equation that follows from E, and we are asked to derive that equation from E, a proof could always be carried out using a combination of applications of `instance` to derive substitution instances of the various equations in E, along with applications of `reflex`, `sym`, `trans`, `fcong`, and `rcong`. Although we will not prove it here (it was first proved by Birkhoff in 1935 [11]), this is an important result that ensures that the six methods in question (the five equational methods along with `instance`) constitute a complete inference system for equational logic. Any equation that follows from E can be derived from it with a—potentially very long—sequence of applications of these few methods. However, such proofs would be operating at a very low level of abstraction, not unlike programs written in machine language, and nontrivial cases would require long and tedious proofs. Instead, from these basic ingredients we can derive results that justify *term rewriting*, which is the kind of larger step in reasoning about equalities that is routinely used by the `chain` method. We begin with:

Theorem 3.1: First Substitution Theorem

If $s = s'$ then for any function symbol f of n arguments and terms t_i of appropriate sorts,

$$f(t_1, \ldots, t_{k-1}, s, t_{k+1}, \ldots, t_n) = f(t_1, \ldots, t_{k-1}, s', t_{k+1}, \ldots, t_n).$$

PROOF: By Reflexivity, $t_i = t_i$ for i ranging from 1 to n except k. The desired equation now follows from $s = s'$ and Functional Substitution. ■

Exercise 3.1: Athena's primitive method `fcong` actually implements the First Substitution Theorem as well as the Substitution Axiom, implicitly invoking Reflexivity as needed. Thus, it accepts the desired equation (of the form that appears in the conclusion of the First Substitution Theorem) and proves it, provided $s = s'$ is in the assumption base.

We can then apply `fcong` again, with the new equation now in the assumption base, to prove an equation between larger terms. Continue the following development:

```
declare g:[N N] -> N
declare h:[N] -> N
declare i:[N N] -> N

assert premise :=  (= (i (S zero) zero)
                      (S (i zero zero)))
```

using repeated applications of `fcong` to prove the following `goal`:

```
define goal :=
  (= (h (g zero
           (i (S zero)
                (i (S zero) zero))))
     (h (g zero
           (i (S zero)
                (S (i zero zero))))))
```

Note: We are using prefix notation here to make it easier to see the terms as trees. □

We now generalize this idea of repeated applications of the First Substitution Theorem:

Theorem 3.2: Second Substitution Theorem

Let s and t be terms, let $I = [i_1 \cdots i_m]$ be a position that is in both terms, and suppose s and t are identical except at I. Let s' be the subterm of s at position I, and t' the subterm of t at position I. If $s' = t'$ then $s = t$.

Remark: Understanding the following proof requires some familiarity with proof by induction, one of the important proof methods to be studied in detail in this book (beginning in Section 3.8). You may wish to skip it and come back to it later, accepting this theorem for now without proof.

PROOF: By induction on the length m of I. For the basis case, $m = 0$, we have $I = [\,]$, hence $s = s'$ and $t = t'$, so we already have $s = t$ by the assumption $s' = t'$. Otherwise, assume the result for positions of length $m-1$ and let $s = f(s_1, \ldots, s_{i_1}, \ldots, s_n)$ and $t = g(t_1, \ldots, t_{i_1}, \ldots, t_n)$, as shown in Figure 3.3. By assumption, $f = g$ and $s_i = t_i$ for all i from 1 to n except i_1. But we also have $s_{i_1} = t_{i_1}$, by applying the induction hypothesis to these terms and position $I' = [i_2 \cdots i_m]$. Then $s = t$ follows by the First Substitution Theorem. ■

The First and Second Substitution Theorems are based on an equation $s = t$ between two terms, but we can derive an even more useful proof principle that starts with a *universally quantified* equation, implicitly using any instance of it as the basis of an application of the Second Theorem. It is best explained and justified using the notions of *rewrite rules* and *term rewriting*, which are in turn based on the notions of substitutions and matching that were described in Section 2.14.8. (It might help to review that section before continuing.)

Rewrite Rules Suppose that p is a universally quantified equation of the following form:[7]

$$p \equiv \forall\, v_1 \cdots v_n \,.\, L = R, \tag{3.2}$$

7 The case of $n = 0$, where p is an unquantified equation, is allowed. Also, as we mentioned earlier, instead of (L = R) we could have a *conditional equation* (q ==> L = R) as the body of the rule. Conditional equations are discussed in Section 3.14, and the full generality of the forms of the justifying sentences allowed in the `chain` method is discussed in Chapter 6.

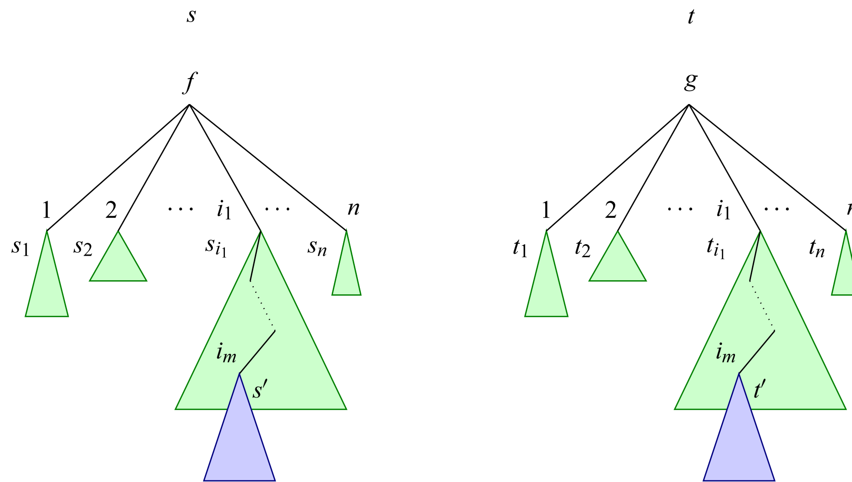


Figure 3.3
Illustration of the Second Substitution Theorem.

where the variables in the term L are $\{v_1, \ldots, v_n\}$ and those in R are a subset of $\{v_1, \ldots, v_n\}$. We call such a sentence p a *rewrite rule* (or, less often, a *rewriting rule*).
Both of the axioms for `Plus`, namely `right-zero` and `right-nonzero`, are in the form of rewrite rules. If we were dealing with addition on integers rather than on natural numbers, then we might have an axiom like this:

```
(forall ?i . ?i + (- ?i) = 0).
```

But if this sentence were written instead as

```
(forall ?i . 0 = ?i + (- ?i)),
```

it would *not* be a rewrite rule, since the variable `?i` appears on the right-hand side of the equation but not on the left. Using such an equation for term rewriting, as defined next, would result in spurious variables being introduced into terms.
Many of the theorems that we derive, such as `left-zero` below, will also be in the form of rewrite rules and will thus be suitable for use by proof techniques that are based on rewriting, such as chaining.

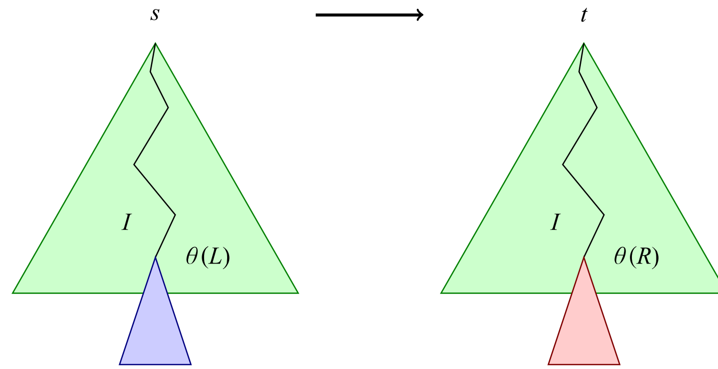


Figure 3.4
Illustration of the Rewriting Theorem.

In fact, most of our rewrite rules will be in a more structured form than what has been described above: They will usually be *constructor-based* rewrite rules. A constructor-based rewrite rule is of the same general form as (3.2) and adheres to the same variable restriction, but, in addition, the left-hand side L is of the form $(f\ \ s_1 \cdots s_n)$, where:

a. each s_i contains only variables and constructors;

b. f is not a constructor (but rather a function symbol that we are defining, such as `Plus`).

Term Rewriting Let p be a rewrite rule whose body is $L = R$. Further, suppose s and t are terms, I is a valid position for both s and t, the subterm s' of s at I matches L with substitution θ, and t is identical to s except at I, where $\theta(R)$ occurs instead. Then we say that s *rewrites* or *reduces* to t using p, and we call $\theta(L)$ and $\theta(R)$ the *redex* and the *contractum*, respectively. We write this relation as $s \longrightarrow_\theta t\ [p]$, or simply $s \longrightarrow t\ [p]$ when θ is clear from the context or immaterial. The relation is depicted in Figure 3.4. From this definition of rewriting we obtain:

Theorem 3.3: Rewriting Theorem

Let p be a rewrite rule, and $t \longrightarrow u\ [p]$. If p then $t = u$.

PROOF: The sentence p implies $\theta(L) = \theta(R)$, and therefore $t\ =\ u$ follows by the Second Substitution Theorem. ■

All of this machinery is brought to bear in the following finite proof principle:

Principle 3.1: Single Rewriting Step

To prove an equation $s = t$ using p, where

$$p \equiv \forall\, v_1 \cdots v_n \,.\, L = R$$

is a rewrite rule, search s for a position I with subterm s' such that s' matches L; i.e., there is a substitution θ such that s' is $\theta(L)$. If t is identical to s except at position I, where $\theta(R)$ occurs instead of s', then $s \to t$ using p, from which $s = t$ follows by the Rewriting Theorem. Otherwise (if any of these conditions fail for position I), continue to the next position. If the search of all subterm positions of s fails, then perform the same search but with the roles of s and t reversed. If successful, $t = s$ is proved, from which $s = t$ follows by Symmetry.

The basic equality chaining that the `chain` method performs is based on the above principle, although `chain` supports multiple rewrite rules in one step (and multiple redex-contractum pairs), as well as conditional rewrite rules and indeed more complex justifying sentences. If all of the individual steps of a `chain` method application

```
(!chain [t0 = t1 [p1] = t2 [p2] = ··· = tn [pn]])
```

are successful, it deduces the equation $t_0 = t_n$ by combining the individually deduced equations using $n - 1$ applications of Transitivity.

We close this section with a notational convention: for a list of sentences $[p_1 \ldots p_k]$, we write

$$s \to_\theta t\, [p_1 \ldots p_k]$$

(or simply $s \to t\, [p_1 \ldots p_k]$ when θ is obvious or immaterial) iff $s \to_\theta t\, [p_i]$ for *some* p_i, $i \in \{1, \ldots, k\}$. This is a natural extension of the notation $s \to_\theta t\, [p]$. We will use it whenever we want to indicate that a term rewrites to another term by virtue of *a collection* of rewrite rules (say, a set of axioms defining a function), without having to single out specifically which axiom is used for the rewrite step in question.

Exercise 3.2: This exercise develops a simple example of the Single Rewriting Step principle, with

```
s = (Plus (S zero) (Plus (S zero) zero))
t = (Plus (S zero) (S (Plus zero zero)))
```

and the rule `(forall x (= (Plus (S x) y) (S (Plus x y))))`.

(a) Show the list of all position and subterm pairs that would be produced if the procedure `positions-and-subterms` (see page 119) were applied to s . (Check your solution by actually running the procedure or with the answer given in the solutions.)

(b) Find two distinct positions I_1 and I_2 in s such that for both $k = 1, 2$, the subterm s' at I_k matches the left-hand side `(Plus (S x) y)`.

(c) For each of the two distinct positions I_1 and I_2 in s found in the previous exercise, show the substitution θ_i by which `(Plus (S x) y)` matches the corresponding subterm.

(d) For which of I_1 and I_2 in s does the Single Rewriting Step principle succeed in proving $s = t$? In that case, what term corresponds to $\theta(R)$? □

* ***Exercise 3.3:***[8] Define a method `basic-rewrite` for proving an equation between terms based on the Second Substitution Theorem. Let s and t be terms, I be a valid position in both s and t, let s' be `(subterm` s I`)`, let t' be `(subterm` t I`)`, and assume `(`s' `=` t'`)` is in the assumption base. Then `(!basic-rewrite` s I t`)` should derive `(`s `=` t`)`. *Hint:* The solution can be expressed as a recursive method, corresponding directly to the inductive proof given for the Second Substitution Theorem. □

* ***Exercise 3.4:*** Define a method `ltr-rewrite` ("left-to-right rewrite") for proving an equation based on the Rewriting Theorem. `(!ltr-rewrite` s p t`)` should prove $(s = t)$ if $s \rightarrow t\,[p]$. *Hint:* Base the search for a redex on procedure `positions-and-subterms`. Check whether a candidate redex matches the left-hand side of the equation in p using procedure `match-terms` (see page 97); when it does, derive the corresponding substitution instance of p using the `instance` method. Then the `basic-rewrite` procedure of the preceding exercise can be used to complete the proof, if one exists for this redex. □

Exercise 3.5: Define a method `rewrite` for proving an equation based on the Single Rewriting Step principle. `(!rewrite` s p t`)` should prove `(`s `=` t`)` if either $s \rightarrow t\,[p]$ or $t \rightarrow s\,[p]$. □

Exercise 3.6: Define a method `chain` that implements equality chaining as described at the beginning of Section 3.2. While Athena's `chain` method is quite a bit more powerful—it also supports implication chaining, multistep-rewriting, and other extensions described later—the basic version that you can now implement in terms of `rewrite` should be able to handle all of the examples in this chapter (except that examples of directional rewriting, introduced on page 133, would need to be modified to use `=`, with `define --> := =; define <-- := =`). □

8 Starred exercises are generally more difficult than unstarred ones. Two stars indicate even greater difficulty.

3.5 More examples of equality chaining

Chaining equalities together to prove a new equality is one of the most useful proof methods in mathematics and computer science, as we shall see throughout this book. However, proving equations like $2+2=4$ provides underwhelming evidence for this method's importance, so let's move on to more interesting examples.

In preparation, we note a couple of more general properties of `Plus`. We begin with the following property:

```
define left-zero := (forall n . zero + n = n)
```

which differs from `right-zero` in that `zero` appears as the first input to `Plus` rather than the second.

Exercise 3.7: Although the `chain` method alone is inadequate to prove `left-zero` (we will see later exactly why this is so), `chain` can prove instances of it, with specific ground terms substituted for `n`. Prove each of the following equations using `chain`:

```
(zero + zero = zero)

(zero + S zero = S zero)

(zero + S S zero = S S zero)
```

Your solution may also use one or both of `right-zero` and `right-nonzero`. □

Later in the chapter we will come back and prove `left-zero`, but for now, we assert it into the assumption base (treating it like an axiom):

```
assert left-zero
```

Similarly:

```
assert left-nonzero := (forall m n . (S n) + m = S (n + m))
```

Although there's quite a bit more to say and prove about `Plus`, let's continue by introducing a multiplication function, `Times`, that takes two natural numbers as inputs and returns their product. First, the syntax:

```
declare Times: [N N] -> N [*]
```

Here, we have overloaded the symbol `*` to mean `Times` when applied to `N` arguments. Next, the semantics:

```
assert Times-zero := (forall x . x * zero = zero)
assert Times-nonzero := (forall x y . x * S y = x * y + x)
```

If we read `(S n)` as $n+1$, the second axiom just says

$$x \cdot (y+1) = x \cdot y + x.$$

Note that `*` has a built-in precedence greater than that of `+`, so that, for example, (x `*` y `+` z) is parsed as ((x `*` y) `+` z).

Let's also introduce a name one and give its meaning with an equation:

```
declare one: N

assert one-definition := (one = S zero)
```

The proof of the following property:

```
define Times-right-one := (forall x . x * one = x)
```

provides another simple illustration of equality chaining:

```
conclude Times-right-one
  pick-any x:N
    (!chain [(x * one)
           = (x * S zero)      [one-definition]
           = (x * zero + x)    [Times-nonzero]
           = (zero + x)        [Times-zero]
           = x                 [left-zero]])
```

The main difference from the previous examples is that the equalities involved are not just between ground terms; the terms include the (fresh) variable denoted by `x`, which, loosely speaking, represents an arbitrary value of sort `N`, as indicated by the way it is introduced in the **`pick-any`** step.

Exercise 3.8: How is `Times-right-one` related to `right-zero`? □

Here is one more property of `Times`, which for the moment we will treat as an axiom by asserting it into the assumption base:

```
assert Times-associative := (forall x y z . (x * y) * z = x * (y * z))
```

As a final bit of preparation for more substantial examples of equational proof, let us define an exponentiation function, `**`. We set the precedence of `**` higher than that of `*` (which is predefined to be 300).

```
declare **: [N N] -> N [310]
```

For semantics, we write:

```
assert Power-right-zero := (forall x . x ** zero = one)

assert Power-right-nonzero := (forall x n . x ** S n = x * x ** n)
```

3.6 A more substantial proof example

Recall the following result from elementary algebra:

```
define power-square-theorem := (forall n x . (x * x) ** n = x ** (n + n))
```

(In more conventional notation, $(x^2)^n = x^{2n}$.) If we define the following procedure:

```
define (power-square-property n) :=
  (forall x . (x * x) ** n = x ** (n + n))
```

we can then express `power-square-theorem` more succinctly as the proposition that every natural number has `power-square-property`:

```
(forall n . power-square-property n).
```

Athena will verify that the two formulations are identical:

```
> (power-square-theorem equals? (forall n . power-square-property n))

Term: true
```

How do we go about proving `power-square-theorem`? In this case it might be a good idea to start by writing down a few specific instances of the result, to convince ourselves that it is valid at least in those cases. For this theorem, instantiating only the variable `n` for a few small values yields:

```
(forall x . (x * x) ** zero = x ** (zero + zero))
(forall x . (x * x) ** S zero = x ** (S zero + S zero))
(forall x . (x * x) ** S S zero = x ** (S S zero + S S zero))
(forall x . (x * x) ** S S S zero =
               x ** (S S S zero + S S S zero))
...
```

These sentences can be automatically obtained by applying `power-square-property` to `zero`, `(S zero)`, and so on. While we could also try instantiating `x` so that we could check the resulting equations with pure calculation (and indeed in Section 3.13 we discuss a technique that automates that approach), let us instead bring proofs into play. Thus, before

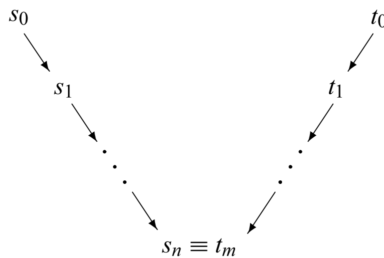
trying to prove `power-square-theorem` in its most general form, let's see if we can prove some of its above instances. The proof of `(power-square-property zero)` is simple:

```
conclude power-zero-case := (power-square-property zero)
  pick-any x:N
    (!chain [((x * x) ** zero)
           = one                           [Power-right-zero]
           = (x ** zero)                   [Power-right-zero]
           = (x ** (zero + zero))          [right-zero]])
```

For `n` ≡ `(S zero)`, first consider the following proof:

```
conclude power-one-case := (power-square-property (S zero))
  pick-any x:N
    (!combine-equations
      (!chain [((x * x) ** S zero)
             = ((x * x) * (x * x) ** zero)  [Power-right-nonzero]
             = ((x * x) * one)              [Power-right-zero]
             = (x * x)                      [Times-right-one]])
      (!chain [(x ** (S zero + S zero))
             = (x ** (S (S zero + zero)))   [right-nonzero]
             = (x ** (S S zero))            [right-zero]
             = (x * (x ** S zero))          [Power-right-nonzero]
             = (x * (x * (x ** zero)))      [Power-right-nonzero]
             = (x * x * one)                [Power-right-zero]
             = (x * x)                      [Times-right-one]]))
```

The structure of this proof, combining *two* applications of `chain`, illustrates a frequently useful variation of equality chaining. Instead of a single chain of equations, we first write a chain in which we start with the left-hand side of the equation we want to prove, at each step using some axiom or other equation in a *left-to-right* direction. That is, we match the equation's left-hand side to some subterm of the current term, replacing it with the right-hand side. Then we work in the same manner starting with the right-hand side, again using axioms and other equations in a left-to-right direction. Diagrammatically:



If we can rewrite each side of the equation we are trying to prove down to the very same term, then we can combine the two chain conclusions to obtain the proof. The method `combine-equations` does just that:

$$\texttt{(!combine-equations } (s_0 \texttt{ = } s_n) \; (t_0 \texttt{ = } t_m)\texttt{)}$$

proves (s_0 = t_0) when both (s_0 = s_n) and (t_0 = t_m) are in the assumption base and s_n and t_m are identical.

In addition, `chain` allows the direction of rewriting to be indicated on each step: Instead of an equality symbol between two terms, either `-->` or `<--` can be used. If `-->` is used, `chain` only attempts to rewrite left-to-right: $t_i \rightarrow t_{i+1}$; and if `<--` is used, it only attempts to rewrite right-to-left: $(t_{i+1} \rightarrow t_i)$. If `=` is specified, `chain` first tries left-to-right, and if that fails, it tries right-to-left.

Note that in right-to-left rewriting, as defined here, it is *not* the equation within p_i, say $L_i = R_i$, that is reversed. That must be avoided, as there might be variables in L_i that do not occur in R_i. (Recall the requirement that the variables of the right-hand side of the equation must be a subset of those of the left.)

For example, every equality symbol in the preceding proof of `power-one-case` could be replaced by `-->`, and the proof of `power-zero-case` could be written as follows:

```
conclude (forall x . (x * x) ** zero = x ** (zero + zero))
  pick-any x:N
    (!chain [((x * x) ** zero)
         --> one                    [Power-right-zero]
         <-- (x ** zero)            [Power-right-zero]
         <-- (x ** (zero + zero))   [right-zero]])
```

Thus, in the original `power-zero-case` proof, where `=` was used, `chain` tries to do the second step using `Power-right-zero` left-to-right and fails, but it succeeds in using it right-to-left, and likewise for the third step.

A fair question at this point is: Do we really need `combine-equations`? Why not write a single application of `chain`? For example, instead of

```
(!combine-equations
  (!chain [s0 --> s1 [p1] --> s2 [p2] --> ··· --> sn [pn]])
  (!chain [t0 --> t1 [q1] --> t2 [q2] --> ··· --> tm [qm]]))
```

we could write a single chain in which we reverse the order of the second sequence of chain links:

```
(!chain [s0 --> s1 [p1] --> s2 [p2] --> ··· --> sn [pn]
           <-- tm-1 [qm] <-- ··· <-- t1 [q2] <-- t0 [q1]])
```

Though possible, it is often more difficult to find the correct sequence of right-to-left proof steps to get from the term s_n (which must be identical to t_m) to the desired term t_0 than it is to find the separate chains working down to a common term using only (or primarily) left-to-right rewrites. In this book, we have a goal more important than succinctness: We want to illustrate and promote good strategies for *finding* proofs and writing them down in

a clear, easy-to-understand style. The strategy of rewriting both sides to a common term often leads to a successful proof more easily than by trying to construct one as a single chain of rewrites, and the proof is usually easier for someone else to read and understand.

Thus, we will usually avoid writing equality proofs as single chains, restricting that practice to cases where (1) the proof can be done just with left-to-right rewriting anyway, or (2) only a few right-to-left rewriting steps are used, and they are only applied to fairly simple terms, as in our proof above of the `n = zero` case. Otherwise, we will tend to write two chains and combine them, as in the `n = (S zero)` case.

Exercise 3.9: Modify the given proof of `power-one-case` to use a single application of `chain`. Indicate the direction of each rewrite with `-->` or `<--` instead of just using `=` signs. Verify that the resulting proof works. □

3.7 A better proof

The proof given for `power-one-case`, whether written with one chain application or two, is longer than it needs to be. It can be shortened by taking advantage of the `power-zero-case` theorem, as follows:

```
conclude (forall x . (x * x) ** S zero = x ** (S zero + S zero))
  pick-any x:N
    (!combine-equations
      (!chain [((x * x) ** S zero)
           --> ((x * x) * ((x * x) ** zero))     [Power-right-nonzero]
           --> ((x * x) * (x ** (zero + zero)))  [power-zero-case]
           --> (x * x * x ** (zero + zero))      [Times-associative]])
      (!chain [(x ** (S zero + S zero))
           --> (x ** (S (S zero + zero)))        [right-nonzero]
           --> (x ** (S (S (zero + zero))))      [left-nonzero]
           --> (x * (x ** (S (zero + zero))))    [Power-right-nonzero]
           --> (x * x * x ** (zero + zero))      [Power-right-nonzero]]))
```

With this change, the proof is not just shorter than the previous one; it has a structure that can be generalized to prove the theorem for *any* value of `n`. To see this, note that the `zero` that appears in the `(S zero)` term in the theorem itself and throughout the proof *never plays any role in the proof.* (For example, neither `right-zero` nor `Times-zero-axiom` or `Power-right-zero` is ever used.) We can take advantage of this observation most easily if we encapsulate the proof in a *method*, as follows:

```
define power-square-step :=
  method (n)
    let {previous-result := (power-square-property n)}
      conclude (power-square-property (S n))
        pick-any x:N
          (!combine-equations
```

```
          (!chain [((x * x) ** S n)
                   --> ((x * x) * ((x * x) ** n))   [Power-right-nonzero]
                   --> ((x * x) * (x ** (n + n)))   [previous-result]
                   --> (x * x * (x ** (n + n)))     [Times-associative]])
          (!chain [(x ** (S n + S n))
                   --> (x ** (S (S n + n)))         [right-nonzero]
                   --> (x ** (S S (n + n)))         [left-nonzero]
                   --> (x * (x ** (S (n + n))))     [Power-right-nonzero]
                   --> (x * x * (x ** (n + n)))     [Power-right-nonzero]]))
```

This method definition does not itself prove a theorem, but a successful call of it does. In particular, if we apply this method to any specific natural number *n*, we will obtain the theorem

(power-square-property (S *n*)),

provided that (power-square-property *n*) *is already in the assumption base*, which is required for the appeal to previous-result in the body of the method to succeed. Hence the name, power-square-step, indicating the "stepping" of power-square-property from one natural number to the next. Let's also encapsulate the previously given proof of the n ≡ zero case in a separate method, power-square-base, which needs no argument:

```
define power-square-base :=
  method ()
    conclude (power-square-property zero)
      pick-any x:N
        (!chain [((x * x) ** zero)
                 = one                      [Power-right-zero]
                 = (x ** zero)              [Power-right-zero]
                 = (x ** (zero + zero))     [right-zero]])
```

Then the following sequence of calls could be extended to obtain the proof of (power-square-property *n*) for *any* natural number *n*:

```
(!power-square-base)
(!power-square-step zero)
(!power-square-step (S zero))
        ⋮
```

Exercise 3.10: Verify in an Athena session that the above sequence of calls does prove (power-square-property *n*) for *n* ≡ zero, (S zero), (S S zero), and (S S S zero). □

3.8 The principle of mathematical induction

At this point we have proved a few special cases of `power-square-theorem`, and we can even imagine repeatedly invoking `power-square-step` to eventually obtain the proof for *any* given n. In Athena we could even program an iterative or recursive proof method that carries out all of the proofs up to the desired natural number, which in fact we'll do later, mainly as an illustration of Athena's proof-programming facilities. While that is probably enough to convince us of the validity of `power-square-theorem`,[9] we still don't have a formal proof of it in its general form, with `n` universally quantified. But we do now have the main ingredients of a proof, namely our `power-square-base` and `power-square-step` methods. To complete the job, we now invoke a fundamental proof method known as *the principle of mathematical induction*. As we will see, this principle does not apply only to sentences about natural numbers, but the form it takes for the natural numbers is one of the most important, and is exactly what we need here:

Principle 3.2: Mathematical Induction for Natural Numbers

To prove $\forall\, n \,.\, P(n)$ where n ranges over the natural numbers, it suffices to prove:

1. *Basis case*: $P(0)$.
2. *Induction step*: $\forall\, n \,.\, P(n) \Rightarrow P(n+1)$.

In the induction step, the antecedent assumption $P(n)$ is called the *induction hypothesis*.

This principle is embodied in Athena's **`by-induction`** proof construct. We can use it to prove `power-square-theorem` as follows:

```
by-induction power-square-theorem {
  zero   => (!power-square-base)
| (S n) => (!power-square-step n)
}
```

The keyword **`by-induction`** is followed by the sentence to be derived, which is a goal of the form

$$\forall\, n : N \,.\, P(n),$$

followed by a number of *clauses*, enclosed in curly braces and separated by `|`, expressing the cases that together are sufficient to complete the proof. There are usually two clauses (there can be more): one that expresses the basis case, corresponding to $P(0)$, and the other

9 In Section 3.13 we will see another technique that can help to convince us that a conjecture p is true without having a proof for it: the `falsify` procedure.

the induction step (or "inductive step"), corresponding to

$$\forall\, n\, .\, P(n) \Rightarrow P(n+1).$$

Each clause is essentially a pair consisting of a constructor pattern π_i that expresses one of the cases of the inductive argument, and a corresponding subproof D_i. The arrow keyword => separates π_i from D_i. The subproof D_i will be evaluated in the original assumption base *augmented with all appropriate inductive hypotheses*. There may be zero inductive hypotheses if the pattern π_i corresponds to a basis case.

The induction-step sentence for our example can be written in Athena as follows:

```
(forall n . power-square-property n ==> power-square-property (S n))
```

If we were trying to prove this sentence from scratch, without the benefit of **by-induction**, we could do it with a proof along the following lines:

```
pick-any n:N
  assume induction-hypothesis := (power-square-property n)
    conclude (power-square-property (S n))
      (!power-square-step n)
```

But with **by-induction** it is not necessary to write this much detail. Essentially, Athena automatically takes care of the first three steps: the **pick-any**, **assume**, and **conclude**. The **pick-any** identifiers will be all and only those identifiers that occur as pattern variables in π_i. These identifiers will be bound to freshly generated variables of the appropriate sorts throughout the evaluation of D_i, obviating the need for an explicit **pick-any**. And the inductive hypotheses will be automatically constructed and inserted into the assumption base for you (temporarily, only while evaluating the subproof D_i), thus obviating the need for an explicit **assume**. So all D_i needs to do is to derive the result $P(n+1)$. Athena will then check that the produced result is of the right form, which also avoids the need for an explicit **conclude**.

Exercise 3.11: Verify in Athena that the above proof derives the induction step, and that the **by-induction** proof before that derives power-square-theorem. □

We expressed proof principle 3.2 in informal notation, writing, for example, $\forall\, n\, .\, P(n)$ instead of using Athena notation. But what exactly is property P? Perhaps the best way to think of it is as a unary procedure that takes a natural number term t and produces a sentence. We can define such a procedure directly in Athena and use it to drive the entire workflow of an inductive proof, as outlined in the following schema:

```
# Start by defining a unary ''property procedure'':

define (P t) := ...

# Then use it to define a goal which says that
```

```
# every object has this property:

define goal := (forall n:N . P n)

# Finally, prove the goal by induction:

by-induction goal {
  zero  => conclude (P zero)
             (!basis-case ...)
| (n as (S m)) =>
    conclude (P n)           # Here the assumption base contains
      (!induction-step ...)  # the inductive hypothesis (P m).
}
```

where `basis-case` and `induction-step` are methods that may take any number of arguments, depending on the situation.

In this particular example, the procedure encoding the property of interest was `power-square-property`:

```
define (power-square-property n) :=
  (forall x . (x * x) ** n = x ** (n + n))
```

This simple procedure can be applied to an arbitrary term t of sort `N` and will produce a sentence essentially stating that `power-square-theorem` holds for t:

```
> (power-square-property zero)

Sentence: (forall ?x:N
            (= (** (Times ?x:N ?x:N)
                   zero)
               (** ?x:N
                   (Plus zero zero))))

> (power-square-property (S zero))

Sentence: (forall ?x:N
            (= (** (Times ?x:N ?x:N)
                   (S zero))
               (** ?x:N
                   (Plus (S zero)
                         (S zero)))))

> (power-square-property ?k)

Sentence: (forall ?x:N
            (= (** (Times ?x:N ?x:N)
                   ?k:N)
               (** ?x:N
                   (Plus ?k:N ?k:N))))
```

But it is not necessary to adhere to this style of defining property procedures for each inductive proof. Neither is it necessary to define proof methods like `power-square-base` and `power-square-step` in order to use **by-induction**. We can simply write out the proofs of the basis case and induction step *inline*. For a simpler example of this approach, let us take a property that we defined earlier in the chapter:

```
define left-zero := (forall n . zero + n = n)
```

There we asserted it into the assumption base, but now let us prove it, using **by-induction** and inline proofs:

```
by-induction left-zero {
  zero => conclude (zero + zero = zero)
            (!chain [(zero + zero) --> zero [right-zero]])
| (n as (S m)) =>
    conclude (zero + n = n)
      let {induction-hypothesis := (zero + m = m)}
        (!chain [(zero + S m)
              --> (S (zero + m))      [right-nonzero]
              --> (S m)               [induction-hypothesis]])
}
```

The principal advantage of writing the subproofs inline is that it requires less preparation. As with ordinary programming, however, when "proof code" gets larger it may be worth the extra trouble to encapsulate it in methods. Then the proof of the basis case can be tested even before attempting to write down the proof of the induction step. Similarly, the proof of the induction step can be separately tested, either with successive applications to `zero`, `(S zero)`, `(S S zero)`, etc., or in the manner shown above for the induction step for `power-square-theorem`.[10] Similar remarks apply to defining procedures like `power-square-property`; while not necessary, they are often useful in making our proofs more structured.

3.8.1 Different ways of understanding mathematical induction

With this simple proof of `left-zero` at hand, let's return to a point we alluded to at the beginning of the chapter, namely that the *only* natural number values are those that can be obtained by starting with `zero` and applying `S` some finite number of times. *One way to understand the principle of mathematical induction for natural numbers is that it says precisely the same thing.* Why? Because:

1. For any value n that is obtained by starting with `zero` and applying `S` some finite number of times, we can prove $P(n)$ by starting with the basis case and applying the induction step the same number of times; and

10 Another, more general approach to testing partial proofs will be introduced in Section 4.7.

2. the principle says that just proving the basis case and the induction step is sufficient to prove P for all natural numbers.

As another aid to intuition about mathematical induction, consider what would happen if we posited a natural number called, say, `unnatural`, that we assume is *not* equal to any of the values generated by `zero` and `S`. Consider then, once again, `left-zero`. Is it still valid? We proved it, but our proof, using the principle of mathematical induction, did not consider the possibility that there could be natural numbers other than those that can be generated from `zero` and `S`. And, without further information, we *cannot* conclude

`(zero + unnatural = unnatural),`

so the property isn't necessarily valid. We might in fact have `(zero + unnatural = zero)` or `(zero + unnatural = one)`, or any other value besides `unnatural`, and that would invalidate `left-zero`.

Exercise 3.12: Joe Certain claims that he can prove

`(zero + unnatural = unnatural)`

using the fact that `Plus` satisfies a commutative law:

`(forall m n . m + n = n + m).`

Thus:

```
(!chain [(zero + unnatural)
     --> (unnatural + zero)      [Plus-commutative]
     --> unnatural               [right-zero]])
```

What's wrong with Joe's "proof"? After all, one *can* prove `Plus-commutative` using mathematical induction; see Exercise 3.20. In answering, assume that `unnatural` has somehow been declared, and in such a way that we do not get a sort-checking error with any of the terms in the above proof (i.e., your answer shouldn't simply be that `unnatural` is an undefined identifier or that there is a sort error). □

Here is still another angle on mathematical induction, this time from the point of view of defining datatypes like `N`. Recall that in Athena, `N` is introduced with the following definition:

```
datatype N := zero | (S N)
```

which can be read as "Sort `N` consists of all values, and only those values, recursively generated by the *constructors* `zero` and `S`." The values in question are called *canonical terms*. We will have more to say about these in Section 3.10, but in general, a canonical term is defined as a term that contains only constructors and/or numerals of sort `Int` or `Real` and/or meta-identifier constants of the form `'`I.

It is this **datatype** definition that determines **by-induction**'s requirements for proof clauses: when the first universally quantified variable of the stated goal is of sort N, **by-induction** requires subsequent clauses that "cover" all possible values that can be generated using constructors zero and S.

Exercise 3.13: In Athena, experiment with a declaration of a datatype N' that differs from N in having an additional value, unnatural:

```
datatype N' := zero' |  unnatural | (S' N)
```

Which of the proofs in this chapter (to this point) would still work if N is replaced by N', zero by zero', and S by S'? □

3.9 List equations

We now turn to a different class of equations, those involving *lists* rather than natural numbers. We will introduce lists as a datatype with two constructors, nil and ::, which are analogous to the natural number constructors zero and S, respectively. (The :: constructor is adopted from similar usage in ML and is pronounced "cons" in the tradition of lists in Lisp and other functional programming languages.) We will see how useful functions on lists, such as join and reverse, can be precisely defined with equational axioms and how further properties can be derived from these axioms. Since the axioms and additional properties are all stated as universally quantified equations, like those concerning natural numbers, we already have at hand the necessary proof methods—equality chaining and induction. We will see that equational chaining works for lists exactly as it does for natural numbers, and that induction requires only a minor adjustment to account for the difference in constructors.

To begin, we introduce lists in Athena with a **datatype** definition, first in a form in which the list elements are natural numbers:

```
datatype N-List := nil | (:: N N-List)
```

Thus, the constructor nil takes no arguments and the constructor :: takes two: a natural number, and, recursively, an N-List. Here are a few ground terms of this sort:

```
nil
(one :: nil)
(zero :: nil)
(zero :: S zero :: S S zero :: S S S zero :: nil)
(one :: zero :: nil)
(one :: zero :: S one :: nil)
```

We interpret `nil` as the empty list (that contains no elements) and $(x :: L)$ as the list whose first element is x and whose remaining elements, if any, are those of list L. (Borrowing from Lisp terminology, we often say that $(x :: L)$ is the result of "consing" x onto L.) So the last example has `one` as its first element, `zero` as its second element, and `(S one)` as its third and final element.

This interpretation is reflected in the definition of the first list function we will study, `join`, for concatenating two lists. By concatenation we mean that the result of `(L1 join L2)` is a list that begins with the elements of list `L1`, in the same order as in `L1`, followed by the elements of `L2`, in the same order as in `L2`.

```
define [L L' L1 L2 L3] :=
       [?L:N-List ?L':N-List ?L1:N-List ?L2:N-List ?L3:N-List]

declare join: [N-List N-List] -> N-List [++]
```

We defined some handy variable names to avoid having to type variables in their fully explicit form (preceded by question marks) in what follows, and we also introduced `++` as an alias for the function symbol `join`.

We want `::` to bind tighter than `++` (to make sure, e.g., that `(x::L1 ++ L2)` is understood as the join of `x::L1` and `L2` rather than the result of consing `x` onto the join of `L1` and `L2`), so we give a higher precedence to `::` with the following directive:

```
set-precedence :: 150
```

We now introduce two axioms that define concatenation:

```
assert left-empty := (forall L . nil ++ L = L)
assert left-nonempty := (forall x L1 L2 . x::L1 ++ L2 = x::(L1 ++ L2))
```

With the previous definitions for natural-number functions such as `Plus` and `Times`, we were able to rely on a lot of experience. Lists, by contrast, are likely not as familiar, so let's make sure we understand how these axioms define `join`. The first axiom states that the list produced by joining `nil` and `L` is identical to `L`.

The second axiom can be read as follows: By consing `x` to a list `L1` and then joining the result with a list `L2`, we get the same list that we would get by first joining `L1` and `L2` and then consing `x` to the result. Figure 3.5 illustrates this relation. The left part of the diagram depicts the computation of $(x{::}L_1 \texttt{ ++ } L_2)$ and the right part that of $(x{::}(L_1 \texttt{ ++ } L_2))$. We see that the results are identical.

Besides holding up defining axioms to scrutiny and drawing diagrams, it is also helpful to use the axioms to compute the function for a few ground term inputs, such as

`(one::zero::nil ++ one::zero::S one::nil).`

We can reduce such a ground term to one that only involves `::` and `nil` by applying the `join` axioms in a chain of equalities:

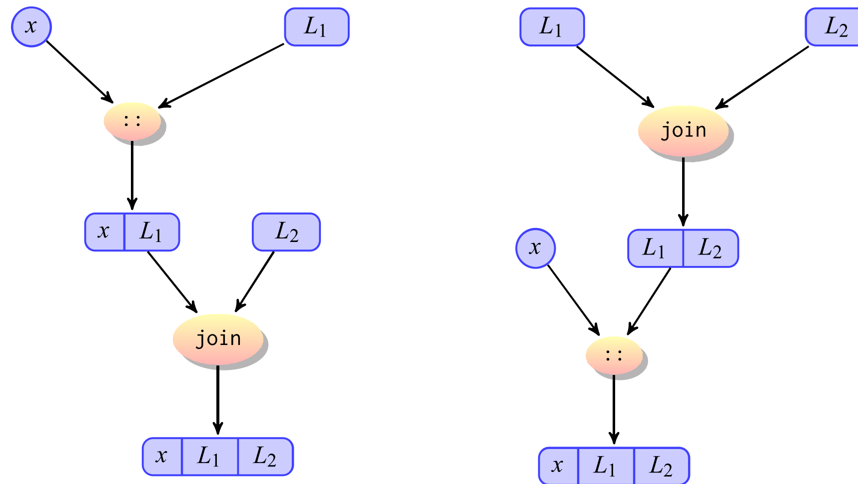


Figure 3.5
Illustration of the second axiom for list concatenation.

```
> (!chain [(one::zero::nil ++ one::zero::S one::nil)
         = (one::(zero::nil ++ one::zero::S one::nil)) [left-nonempty]
         = (one::zero::(nil ++ one::zero::S one::nil)) [left-nonempty]
         = (one::zero::one::zero::S one::nil)          [left-empty]])

Theorem: (= (join (:: one
                      (:: zero nil))
                  (:: one
                      (:: zero
                          (:: (S one)
                              nil))))
            (:: one
                (:: zero
                    (:: one
                        (:: zero
                            (:: (S one)
                                nil))))))
```

Carrying out such equality proofs and examining the results provides additional evidence that the axioms for `join` give it the meaning we informally prescribed for it. (A more direct computational way to obtain such evidence—without constructing proofs—will be introduced in Section 3.10.) But still another way to bolster our confidence is to state, as conjectures, more properties that we intuitively expect a list concatenation function to have. The simplest such property is one like the first axiom, but with `nil` as the second argument:

```
define right-empty := (forall L . L ++ nil = L)
```

Note the analogy with the natural number function `Plus`, with `nil` now playing the role of `zero`: We had an axiom

```
assert right-zero := (forall n . n + zero = n),
```

and we later stated and proved

```
define left-zero := (forall n . zero + n = n).
```

If we can prove `right-empty` we will have completed the analogy (except that our list axiom and property have `nil` in the first and second arguments of `join`, respectively, the reverse of what we had with `zero` for `Plus`).

We cannot prove `right-empty` just by equality chaining, since neither of our axioms can be used to rewrite one side of its equation to the other. But as we did with `Plus`, let's see what happens with a few ground term inputs to `join` when its second argument is `nil`. For example:

```
(!chain [(one::nil ++ nil)
       = (one::(nil ++ nil))    [left-nonempty]
       = (one::nil)             [left-empty]])

(!chain [(one::zero::nil ++ nil)
       = (one::(zero::nil ++ nil))     [left-nonempty]
       = (one::zero::(nil ++ nil))     [left-nonempty]
       = (one::zero::nil)              [left-empty]])

(!chain [(S one::one::zero::nil ++  nil)
       = (S one::(one::zero::nil ++ nil))   [left-nonempty]
       = (S one::one::(zero::nil ++ nil))   [left-nonempty]
       = (S one::one::zero::(nil ++ nil))   [left-nonempty]
       = (S one::one::zero::nil)            [left-empty]])
```

In each case, we verify that the result computed by `join` is the same as its first argument, and we strongly suspect this must be true for lists of any length. But if we continued this pattern, then to verify `right-empty` for a list of length n we would need a calculation of length n. We can do better by noting that we can use the result for length n in the proof for length $n+1$. The following method definitions and evaluations are analogous to those of `power-sqare-base` and `power-square-step` in Section 3.7, but they are a little more complicated due to the extra argument for the list element in the `::` constructor.

```
define (join-nil-base) :=
  conclude (nil ++ nil = nil)
    (!chain [(nil ++ nil) = nil [left-empty]])

define (join-nil-step IH x L) :=
  conclude (x::L ++ nil = x::L)
```

```
    (!chain [(x::L ++ nil)
           = (x::(L ++ nil))      [left-nonempty]
           = (x :: L)             [IH]])
```

We now have:

```
> define p0 := (!join-nil-base)

Theorem: (= (join nil nil)
            nil)

Sentence p0 defined.

> define p1 :=
    pick-any x
      (!join-nil-step p0 x nil)

Theorem: (forall ?x:N
           (= (join (:: ?x:N nil)
                    nil)
              (:: ?x:N nil)))

Sentence p1 defined.

> define p2 :=
    pick-any x:N y:N
      (!join-nil-step p1 x (y::nil))

Theorem: (forall ?x:N
           (forall ?y:N
             (= (join (:: ?x:N
                          (:: ?y:N nil))
                      nil)
                (:: ?x:N
                    (:: ?y:N nil)))))

Sentence p2 defined.

> define p3 :=
    pick-any x:N y:N z:N
      (!join-nil-step p2 x (y::z::nil))

Theorem: (forall ?x:N
           (forall ?y:N
             (forall ?z:N
               (= (join (:: ?x:N
                            (:: ?y:N
                                (:: ?z:N nil)))
                        nil)
                  (:: ?x:N
                      (:: ?y:N
```

```
                              (:: ?z:N nil)))))))

Sentence p3 defined.
```

The pattern of these proofs suggests that we could prove `right-empty` in full generality by mathematical induction, perhaps by reformulating the property in terms of the length of the first list, which is a natural number. In fact, however, we need no such reformulation; we can prove it directly. In Athena, the following proof works:

```
> by-induction right-empty {
    nil => (!join-nil-base)
  | (h::t) => let {IH := (t ++ nil = t)}
                (!join-nil-step IH h t)
  }

Theorem: (forall ?L:N-List
           (= (join ?L:N-List nil)
              ?L:N-List))
```

or, writing the proofs of the `nil` and `::` cases inline:

```
> by-induction right-empty {
    nil => (!chain [(nil ++ nil) = nil [left-empty]])
  | (h::t) =>
      let {IH := (t ++ nil = t)}
        conclude (h::t ++ nil = h::t)
          (!chain [(h::t ++ nil)
                 = (h :: (t ++ nil))   [left-nonempty]
                 = (h::t)              [IH]])
  }

Theorem: (forall ?L:N-List
           (= (join ?L:N-List nil)
              ?L:N-List))
```

The way **`by-induction`** works for lists is based on the following:

Principle 3.3: Mathematical Induction for Lists of Natural Numbers

To prove $\forall\ L\ .\ P(L)$ where L ranges over lists of natural numbers, it suffices to prove:

1. *Basis case*: $P(\texttt{nil})$.
2. *Induction step*: $\forall\ L\ .\ P(L) \Rightarrow \forall\ x\ .\ P(x\texttt{::}L)$.

In the induction step, the antecedent assumption $P(L)$ is called the *induction hypothesis*, and x ranges over natural numbers.

So we have seen two flavors of mathematical induction, for natural numbers and for lists of natural numbers. Both are supported by Athena's **by-induction** proof form. The way **by-induction** adapts to different datatypes is through the information supplied in **datatype** definitions about the given constructors. Thus, from

```
datatype N-List := nil | (:: N N-List),
```

by-induction deduces that it must expect clauses corresponding to at least two cases, one corresponding to `nil` and one corresponding to `::`. The `nil` case is called a *basis case* because `nil` is an irreflexive constructor that takes no argument of the datatype being defined; there would be more than one basis case if there were more such constructors. The `::` case, since `::` does take an argument of the datatype being defined, corresponds to an *induction step* (or "inductive step"), and **by-induction** temporarily assumes an appropriate inductive hypothesis for the duration of the proof given in that clause. The general evaluation semantics of **by-induction** are discussed in Appendix A.3.

Exercise 3.14: Reformulate the inductive proof of `right-empty` in the property-procedure style described in Section 3.8. □

3.9.1 Polymorphic datatypes

Before going on to further examples of proofs about list functions, it should be noted that none of the axioms or proofs so far have depended in any way on the sort of the list elements. In place of `N`, we could have used any sort, and everything other than the actual values in ground terms would be the same. So, for example, if we defined

```
datatype Boolean-List := nilb | (consb Boolean Boolean-List)
```

then we could repeat all of the development by replacing any `N` ground values with `true` or `false`; nothing else would need to change. But such repetition is something we should avoid if possible; writing proofs is hard enough without having to repeat them over and over again for different sorts! Fortunately, we can avoid it by issuing declarations and definitions with *sort parameters*, as discussed in Section 2.8:

```
datatype (List S) := nil | (:: S (List S))

declare join: (S) [(List S) (List S)] -> (List S) [++]
```

(Note that `(List S)` is predefined in Athena and in practice there would be no need to actually issue the above definition.) As a convenience, we define a few general (polymorphic) variable names, as well as some specifically for polymorphic lists; and we set the precedence level of `::` to 150:

```
define [x y z x1 x2 h h1 h2] :=  [?x ?y ?z ?x1 ?x2 ?h ?h1 ?h2]

define [L L' L0 L1 L2 t t1 t2] :=
       [?L:(List 'S1) ?L':(List 'S2) ?L0:(List 'S3) ?L1:(List 'S4)
        ?L2:(List 'S5) ?t:(List 'S6) ?t1:(List 'S7) ?t2:(List 'S8)]

set-precedence :: 150
```

We can now give a polymorphic definition of `join` as follows:

```
assert left-empty    := (forall L . nil ++ L = L)
assert left-nonempty := (forall x L1 L2 . x::L1 ++ L2 =  x::(L1 ++ L2))
```

All of the proofs we previously did with `N-List` values could now be redone using `(List N)` values instead; for example, one of the proofs on page 142:

```
let {L := (one::zero::S one::nil)}
  (!chain [(one::zero::nil ++ L)
        = (one::(zero::nil ++ L))  [left-nonempty]
        = (one::zero::(nil ++ L))  [left-nonempty]
        = (one::zero::L)           [left-empty]])
```

or the proof on page 146:

```
define right-empty := (forall L . L ++ nil = L)

> by-induction right-empty {
    nil => (!chain [(nil ++ nil) = nil [left-empty]])
  | (L as (h::t)) =>
    let {IH := (t ++ nil = t)}
      conclude (h::t ++ nil = h::t)
        (!chain [(h::t ++ nil)
               = (h :: (t ++ nil))     [left-nonempty]
               = L                     [IH]])
  }

Theorem: (forall ?L:(List 'S)
           (= (join ?L:(List 'S)
                    nil:(List 'S))
              ?L:(List 'S)))
```

The first of these proofs involves ground terms of the list element sort (`zero`, `one`, etc.), and so would have to be redone for lists of another sort (`Boolean`, say, with `true` and `false` elements). But the second proof contains no occurrence of ground terms, so it applies without change to `(List Boolean)` or indeed to `(List` *S*`)` for *any* sort *S*: Once the theorem it proves is in the assumption base, the theorem can be used with `(List S)` values for any sort `S`. The same will be true for all of the example proofs and exercises that follow.

The following more general induction principle pertains to lists of values of an arbitrary sort S:

Principle 3.4: Mathematical Induction for Lists over sort S

To prove $\forall\, L \,.\, P(L)$ where L ranges over lists of sort S, it suffices to prove:

1. *Basis case*: P(`nil`).
2. *Induction step*: $\forall\, L \,.\, P(L) \Rightarrow \forall\, x \,.\, P(x::L)$.

As before, the antecedent assumption $P(L)$ in the induction step is called the *induction hypothesis*; the variable x is of sort S.

Exercise 3.15: Our axioms for `join` define equations for terms in which `nil` and `::` terms occur in the first argument to `join`, and we proved `right-empty` for the case of `nil` in the second argument. Write an equation for the case of a `::` term in the second argument. (An answer is given in Exercise 3.30, which asks you to prove the given equation.) □

By now we have seen several cases where a binary function obeys one or more axioms, such as identity element axioms (`zero` is an identity element for `Plus`, `one` is for `Times`, and now, `nil` is for `join`), and associative and commutative axioms (with which of course we are familiar in the cases of `Plus` and `Times` for the natural numbers, but which exercises in Section 3.17 ask you to confirm with proofs). Such basic axioms are so frequently useful in reasoning about specific applications of binary functions that when we are presented with a new binary function it is a good strategy to determine whether or not they hold for it. So let's next consider whether our list `join` function is associative:

```
define join-associative :=
  (forall L0 L1 L2 . (L0 ++  L1) ++ L2 = L0 ++ (L1 ++ L2))
```

Intuitively, this relation should hold, as Figure 3.6 suggests. Note the similarity of this diagram to the one on page 143 depicting a relation between `join` and `::`, which can be regarded as a "mixed" associativity relation.

In fact, in the following proof of `join-associative` by induction, we make several uses of that same relation, `left-nonempty`, between `++` and `::`:

```
by-induction join-associative {
  nil =>
    pick-any L1 L2
      (!chain [((nil ++ L1) ++ L2)
           --> (L1 ++ L2)              [left-empty]
           <-- (nil ++ (L1 ++ L2))    [left-empty]])
| (L as (h::t)) =>
    let {IH := (forall L1 L2 . (t ++ L1) ++ L2 = t ++ (L1 ++ L2))}
      conclude (forall L1 L2 . (L ++ L1) ++ L2 = L ++ (L1 ++ L2))
```

```
        pick-any L1 L2
          (!chain
            [((h::t ++ L1) ++ L2)
         --> ((h::(t ++ L1)) ++ L2)  [left-nonempty]
         --> (h::((t ++ L1) ++ L2))  [left-nonempty]
         --> (h::(t ++ (L1 ++ L2)))  [IH]
         <-- (h::t  ++ (L1 ++ L2))   [left-nonempty]])
}
```

We give additional examples of proofs of list theorems in Section 3.11, but we first step back to consider what we can do to ensure that the axioms on which we are basing our proofs are the proper starting points. The main tool we recommend for this purpose is *evaluation of ground terms*, as discussed in the next section.

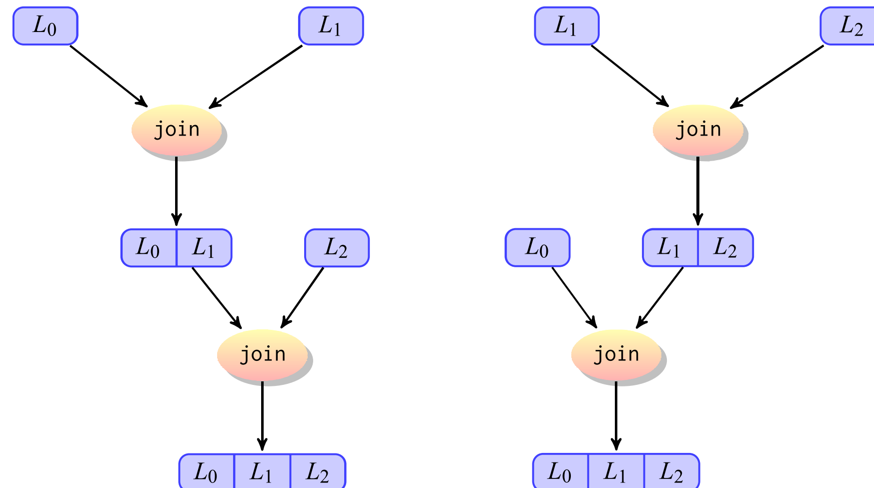


Figure 3.6
Illustration of the associativity of list concatenation.

3.10 Evaluation of ground terms

How do we know that the two axioms we have given for `Plus` succeed in capturing our intentions, namely, that they characterize `Plus` as the binary addition function on the natural numbers? Admittedly, these axioms are simple and it is arguable that nothing more than a moment's reflection is needed to grasp their meaning. Moreover, we are able to prove various results about `Plus` that we know to be true of the addition function, such