Chapter 4

Mathematical Induction

Induction, basic proofs, . . .

4.1 Lists as Mathematical Objects

A sequence is an ordered list of elements. In fact, we will more often use the term shorter term "list" for this kind of mathematical object. Our notation for lists displays the elements of the list, enclosed in parentheses, separated by spaces. For example, the formula "(8 3 7)" denotes the list with first element 8, second element 3, and third element 7. The formula "(9 8 3 7)" denotes a list with the same elements, but with an additional element "9" at the beginning. We use the symbol "nil" for the empty list (that is, the list with no elements).

We will start with three basic operators for lists. One of them, the construction operator, "cons", inserts a new element at the beginning of a sequence. Formulas using "cons", like all formulas in the mathematical notation we have been using to discuss software concepts, are written in prefix form. So, the formula "(cons x xs)" denotes the list with the same elements as the list xs, but with an additional element x inserted at the beginning. If x stands for the number "9", and xs stands for the list "(8 3 7)", then "(cons x xs)" constructs the list "(9 8 3 7)".

Any list can be constructed by starting from the empty list and using the construction operator to insert the elements of the list, one by one. For example, the formula "(cons 8 (cons 3 (cons 7 nil)))" is another notation for the list "(8 3 7)". In fact, using the operator "cons" is the only way to construct non-empty lists. The empty list “nil” is given. All other lists (that is, all non-empty lists) are constructed using the “cons” operator. The formula “(8 3 7)” is shorthand for “(cons 8 (cons 3 (cons 7 nil)))”.

The operator that checks for non-empty lists is “consp”. The formula “(consp xs)” delivers true if xs is a non-empty list and false otherwise. The {*cons*} axiom of list construction is a formal statement of the fact that all non-empty lists are constructed with the “cons” operator.

Axiom {*cons*} (consp xs) $\leftrightarrow$ ($\exists y.\exists ys.$ (xs = (cons y ys)))

We will often cite the {*cons*} axiom to write a formula like (cons x xs) in place of any list we know is not empty. When we do this, we will take care to choose the symbols x and xs to avoid conflicts with other symbols that appear in the context of the discussion. Furthermore, we will often cite a less formal version of the {*cons*} axiom when we know we are dealing with a non-empty list. For example, the list (x_1 x_2 $\ldots x_{n+1}$) cannot be empty because it has $n+1$ elements, and $n+1$ is at least one when n is a natural number. (We will always assume that subscripts are natural numbers.)

Informal Axiom {*cons*} (x_1 x_2 $\ldots x_{n+1}$) = (cons x_1 (x_2 $\ldots x_{n+1}$))

The construction operator, “cons”, cannot be the whole story, of course. To compute with lists, we need to be able to construct them, but we also need to be able to take them apart. There are two basic operators for taking lists apart: “first” and “rest”. We express the relationship between these operators and the construction operator in the form of equations ({*first*} and {*rest*}), along with the informal version of the {*cons*} axiom.

Axioms: {*cons*}, {*first*}, and {*rest*}

(x_1 x_2 $\ldots x_{n+1}$) = (cons x_1 (x_2 $\ldots x_{n+1}$))	{*cons*}
(first (cons x xs)) = x	{*first*}
(rest (cons x xs)) = xs	{*rest*}

The {*first*} axiom is a formal statement of the fact that the operator “first” delivers the first element from non-empty list. The {*rest*} axiom states that the operator “rest” delivers a list like its argument, but without the first element. Note that the list in the {*first*} and {*rest*} axioms have at least one element because they are constructed by the cons operator.

We will use equations like the ones in these axioms in the same way we used the logic equations in Figure 2.2 and the arithmetic equations of Figure 2.1. That is, whenever we see a formula like “(first (cons x xs))”, no matter what formulas x and xs stand for, we will be able to cite equation {*first*} to replace “(first (cons x xs))” by the simpler formula “x”. Vice versa, we can also cite equation {*first*} to replace any formula “x” by the more complicated formula “(first (cons x xs))”. Furthermore, the formula “xs” in the replacement can be any formula we care to make up, as long as it is grammatically correct.

Similarly, we can cite the equation {*rest*} to justify replacing the formula “(rest (cons x xs))” by “xs” and vice versa, regardless of what formulas the symbols “x” and “x” stand for. In other words, these are ordinary algebraic equations. The only new factors are (1) the kind of mathematical object they denote (lists, instead of numbers or True/False propositions), and (2) the syntactic quirk of prefix notation (instead of the more familiar infix notation).

All properties of lists, as mathematical objects, derive from the {cons} axiom and equations {first} and {rest}. For example, suppose there is an operator called “len” that delivers the number of elements in a list. We can use check-expect to test len in some specific cases.

```
(check-expect (len (cons 8 (cons 3 (cons 7 nil)))) 3)
(check-expect nil 0)
```

We can use the doublecheck facility to automate tests. We expect that the number of elements in a non-empty list is one more than the number of elements remaining in the list after the first one is dropped using the “rest” operator. The following property tests this expectation.

```
(defproperty len-test
  (xs :value (random-list-of (random-natural)))
  (= (len xs)
     (if (consp xs)
         (+ 1 (len (rest xs)))
         0)))
```

This property holds under all circumstances. We can express the idea in the form of equations that serve as axioms for the len operator.

Axioms: {*len*} (len nil) = 0 {*len0*}
(len (cons x xs)) = (+ (len xs) 1) {*len1*}

We expect the “len” operator to deliver a natural number, regardless of what its argument is, and we can derive this property of len from its axioms. Instead of plodding through this derivation at this point, we are going to proceed to some more interesting issues. For the record, we state it as a theorem. Later, you will have a chance to derive this theorem from the {*len*} axioms. The theorem refers to the natp operator, which you have seen before (see page **??**). It delivers true if its argument is a natural number and false otherwise.

Theorem {*len-nat*} $\forall xs.$(natp (len xs))

A related fact is that the formula (consp xs) is logically equivalent to the formula (> (len xs) 0). In the notation from Chapter 2: (consp xs)$\leftrightarrow$(> (len xs) 0). This theorem, too, can be derived from the{*len*} axioms, but we will take a pass on proving the theorem, for the moment, and state it without proof.

Theorem {*consp*$\leftrightarrow$len>0} $\forall xs.$(consp xs)$\leftrightarrow$(> (len xs) 0)

4.2 Mathematical Induction

The cons, first, and rest operators form the basis for computing with lists, but there are lots of others, too. For example, consider an operator "append" that concatenates two lists. We describe this operator using an informal schematic for lists that labels the elements of the list as variables with natural numbers as subscripts. The number of integers in the subscript sequence implicitly reveals the number of elements in the list.

In the following list schematics, the "x" list has m elements, the "y" list has n elements, and the concatenated list has $m + n$ elements.

$$\text{(append } (x_1\ x_2 \ldots x_m)\ (y_1\ y_2 \ldots y_n)) = (x_1\ x_2 \ldots x_m\ y_1\ y_2 \ldots y_n)$$

Some simple tests might bolster our understanding of the operator.

```
(check-expect (append '(1 2 3 4) '(5 6 7))
              '(1 2 3 4 5 6 7))
(check-expect (append '(1 2 3 4 5) nil)
              '(1 2 3 4 5))
```

We can use doublecheck for more extensive testing. If we concatenate the empty list nil with a list ys, we expect to get ys as a result: (append nil ys) = nil. If we concatenate a non-empty list xs with a list ys, we expect the first element of the result to be the same as the first element of xs. Furthermore, we expect the rest of the elements to be the elements of the list we would get if we concatenated a list made up of the other elements of

What is the single-quote mark doing in the formula '(1 2 3 4)? It is there to avoid confusing lists overlaps with computational formulas. By default, the ACL2 system interprets a formula like (f x y z) as an invocation of the operator "f" with operands x, y, and z. ACL2 interprets the first symbol it encounters after a left parenthesis as the name of an operator, and it interprets the other formulas, up to the matching right parenthesis, as operands. So, ACL2 interprets the "1" in the formula (1 2 3 4) as the name of an operator. Because there is no operator with the name "1", the interpretation fails.
If we want to specify the list "(1 2 3 4)" in a formula, we can, of course, use the cons operator to construct it: (cons 1 (cons 2 (cons 3 (cons 4 nil)))). But, that's too bulky for regular use. The single-quote trick provides a shorthand: '(1 2 3 4) has the same meaning as the bulky version. The single-quote mark suppresses the default interpretation of the first symbol after the left-parenthesis and delivers the list whose elements are in the parentheses. Without the single-quote mark, the "1" in "(1 2 3 4)" would be interpreted as an operator, and because there is no operator named "1", the formula would make no sense.

Aside 9: Single-quote Shorthand for Lists

xs, that is (rest xs), with ys. The following property definition expresses this idea formally.

```
(defproperty append-test
  (xs :value (random-list-of (random-natural))
   ys :value (random-list-of (random-natural)))
  (equal (append xs ys)
         (if (consp xs)
             (cons (first xs)
                   (append (rest xs) ys))
             ys)))
```

This might not be the first test you would think of, but if the test failed to pass, you would for sure know something was wrong with the append operator. In fact the property is so plainly correct, we are going to state it in the form of equations that we accept as axioms.

Like the {*len*} theorem, there are two {*append*} equations, and they specify the meaning of an append operation in different situations. One of

Why does the property say "(equal (append xs ys) ...)" instead of "(= (append xs ys) ...)"? the "=" operator is restricted to numbers. The "equal" operator can check for equality between other kinds of objects. You can always use "equal", but you can only use "=" when both operands are numbers. Why bother with "=", when its use is so limited? We might say it makes the formula look more like an equation, but that's not really much of an excuse, since we have already had to conform to prefix notation instead of the more familiar infix notation. So, feel free to use the "equal" operator all the time if you want to. We will be using "=" when we can and hope it doesn't put too much of an extra burden on you.

Aside 10: "equal" vs "="

them specifies the meaning when the first argument in the invocation is a list with at least one element (that is, when (consp xs) is true), the other when it has no elements (that is, when it is nil).

Axioms: {*append*}
(append nil ys) = ys {*app0*}
(append (cons x xs) ys) = (cons x (append xs ys)) {*app1*}

These equations about the append operation simple enough, but it turns out that lots of other properties of the append operation can be derived from them. For example, we can prove that the length of the concatenation of two lists is the sum of the lengths of the lists. We call this theorem the "additive law of concatenation". Let's see how a proof of this law could be carried out.

First, let's break it down into a sequence of special cases. We will use L(n) as shorthand for the proposition that (len (append (x_1 x_2 ... x_n) ys)) is the sum of (len (x_1 x_2 ... x_n)) and (len ys):

$$\text{L}(n) \equiv (= (\text{len (append } (x_1\ x_2 \ldots x_n)\ ys))$$
$$(+\ (\text{len } (x_1\ x_2 \ldots x_n))\ (\text{len } ys)))$$

For the first few values of n, L(n) would stand for the following equations.

$$
\begin{array}{lcll}
\mathrm{L}(0) & \equiv & (= & (\text{len (append nil } ys)) \\
 & & & (+\ (\text{len nil})\ (\text{len } ys))) \\
\mathrm{L}(1) & \equiv & (= & (\text{len (append } (x_1)\ ys)) \\
 & & & (+\ (\text{len } (x_1))\ (\text{len } ys))) \\
\mathrm{L}(2) & \equiv & (= & (\text{len (append } (x_1\ x_2)\ ys) \\
 & & & (+\ (\text{len } (x_1\ x_2))\ (\text{len } ys))) \\
\mathrm{L}(3) & \equiv & (= & (\text{len (append } (x_1\ x_2\ x_3)\ ys)) \\
 & & & (+\ (\text{len } (x_1\ x_2\ x_3))\ (\text{len } ys))) \\
\mathrm{L}(4) & \equiv & (= & (\text{len (append } (x_1\ x_2\ x_3\ x_4)\ ys)) \\
 & & & (+\ (\text{len } (x_1\ x_2\ x_3\ x_4))\ (\text{len } ys)))
\end{array}
$$

We can derive L(0) from the {*append*} and {*len*} axioms as follows, starting from the first operand in the equation that L(0) stands for (the left-hand side, if the equation were written in the conventional way rather than prefix form), and ending with the second operand (right-hand side).

$$
\begin{array}{lll}
 & (\text{len (append nil } ys)) & \\
= & (\text{len } ys) & \{\textit{app0}\} \\
= & (+\ (\text{len } ys)\ 0) & \{+ \text{ identity}\}\ (\text{page } 14) \\
= & (+\ 0\ (\text{len } ys)) & \{+ \text{ commutative}\}\ (\text{page } 14) \\
= & (+\ (\text{len nil})\ (\text{len } ys)) & \{\textit{len0}\}
\end{array}
$$

That was easy. How about L(1)?

$$
\begin{array}{lll}
 & (\text{len (append } (x_1)\ ys)) & \\
= & (\text{len (append (cons } x_1 \text{ nil) } ys) & \{\textit{cons}\} \\
= & (\text{len (cons } x_1 \text{ (append nil } ys))) & \{\textit{app1}\} \\
= & (+\ 1\ (\text{len (append nil } ys))) & \{\textit{len1}\} \\
= & (+\ 1\ (+\ (\text{len nil})\ (\text{len } ys))) & \{\mathrm{L}(0)\} \\
= & (+\ (+\ 1\ (\text{len nil}))\ (\text{len } ys)) & \{+ \text{ associative}\}\ (\text{page } 14) \\
= & (+\ (\text{len (cons } x_1 \text{ nil)})\ (\text{len } ys)) & \{\textit{len1}\} \\
= & (+\ (\text{len } (x_1)\ (\text{len } ys)) & \{\textit{cons}\}
\end{array}
$$

That was a little harder. Will proving L(2) be still harder? Let's try it.

$$\begin{array}{lll}
 & (\text{len } (\text{append } (x_1\ x_2)\ ys)) & \\
= & (\text{len } (\text{append } (\text{cons } x_1\ (x_2))\ ys)) & \{\textit{cons}\} \\
= & (\text{len } (\text{cons } x_1\ (\text{append } (x_2)\ ys))) & \{\textit{app1}\} \\
= & (+\ 1\ (\text{len } (\text{append } (x_2)\ ys))) & \{\textit{len1}\} \\
= & (+\ 1\ (+\ (\text{len } (x_2))\ (\text{len } ys))) & \{\text{L}(1)\} \\
= & (+\ (+\ 1\ (\text{len } (x_2)))\ (\text{len } ys)) & \{+\ \text{associative}\} \\
= & (+\ (\text{len } (\text{cons } x_1\ (x_2)))\ (\text{len } ys)) & \{\textit{len1}\} \\
= & (+\ (\text{len } (x_1\ x_2))\ (\text{len } ys)) & \{\textit{cons}\}
\end{array}$$

Fortunately, proving L(2) was no harder than proving L(1). In fact the two proofs cite exactly the same equations all the way through, except in one place. Where the proof of L(1) cited the equation L(0), the proof of L(2) cited the equation L(1). Maybe the proof of L(3) will work the same way.

$$\begin{array}{lll}
 & (\text{len } (\text{append } (x_1\ x_2\ x_3)\ ys)) & \\
= & (\text{len } (\text{append } (\text{cons } x_1\ (x_2\ x_3))\ ys)) & \{\textit{cons}\} \\
= & (\text{len } (\text{cons } x_1\ (\text{append } (x_2\ x_3)\ ys))) & \{\textit{app1}\} \\
= & (+\ 1\ (\text{len } (\text{append } (x_2\ x_3)\ ys))) & \{\textit{len1}\} \\
= & (+\ 1\ (+\ (\text{len } (x_2\ x_3))\ (\text{len } ys))) & \{\text{L}(2)\} \\
= & (+\ (+\ 1\ (\text{len } (x_2\ x_3)))\ (\text{len } ys)) & \{+\ \text{associative}\} \\
= & (+\ (\text{len } (\text{cons } x_1\ (x_2\ x_3)))\ (\text{len } ys)) & \{\textit{len1}\} \\
= & (+\ (\text{len } (x_1\ x_2\ x_3))\ (\text{len } ys)) & \{\textit{cons}\}
\end{array}$$

By now, it's easy to see how to derive L(4) from L(3), then L(5) from L(4), and so on. If you had the time and patience, you could surely prove L(100), L(1000), or even L(1000000) by deriving the next one from the one you just finished proving, following the established pattern. We could even write a program to print out the proof of L(n), given any natural number n.

Since we know how to prove L(n) for any natural number n, it seems fair to say that we know all those equations are true. However, to complete proof of the formula ($\forall n$.L(n)), we need a rule of inference that allows us to make conclusions from patterns like those we observed in proving L(1), L(2), and so on. That rule of inference is known as "mathematical induction".

Mathematical induction provides a way to prove that formulas like ($\forall n$.P(n)) are true when P is a predicate whose universe of discourse is the natural numbers. If for each natural number n, P(n) stands for a

$$\begin{array}{l}\text{Prove P}(0)\\ \text{Prove } (\forall n.(\text{P}(n) \rightarrow \text{P}(n+1)))\\ \hline \text{Infer } (\forall n.\text{P}(n))\end{array}$$

Figure 4.1: Mathematical Induction–a rule of inference

proposition, then mathematical induction is an applicable inference rule in a proof that is $(\forall n.\text{P}(n))$ true. That is not to say that such a proof can be constructed. It's just that mathematical induction might provide some help in the process.

The rule goes as follows: one can infer the truth of $(\forall n.\text{P}(n))$ from proofs of two other propositions. Those two propositions are P(0) and $(\forall n.(\text{P}(n) \rightarrow \text{P}(n+1)))$. It's a very good deal if you think about it. A direct proof of $(\forall n.\text{P}(n))$ would require a proof of proposition P(n) for each value of n (0, 1, 2, . . .). But, in a proof by induction, the only proposition that needs to be proved on its own is P(0). In the proof any of the other propositions, you are allowed to cite the previous one in the sequence as a justification for any step in the proof.

The reason you can assume that P(n) is true in the proof of P($n+1$) is because the goal is to prove that the formula $\text{P}(n) \rightarrow \text{P}(n+1)$ has the value "true". We know from the truth table of the implication operator (see page 20) that the formula $\text{P}(n) \rightarrow \text{P}(n+1)$ is true when P(n) is false. So, we only need to verify that the formula is true when P(n) is true. The formula will be true in this case when P($n+1$) is true. So, all we need to prove is that P($n+1$) under the assumption that P(n) is true.

That is, in the proof of P($n+1$), you can cite P(n) to justify any step in the proof. P(n) gives you a leg up in the proof of P($n+1$) and is known as the "induction hypothesis".

Now, let's apply mathematical induction to prove the additive law of concatenation. Here, the predicate that we will apply the method to is L:

$$\begin{array}{ll}\text{L}(n) \equiv (= & (\text{len } (\text{append } (x_1\ x_2 \ldots x_n)\ ys))\\ & (+\ (\text{len } (x_1\ x_2 \ldots x_n))\ (\text{len } ys)))\end{array}$$

We have already proved L(0). All that is left is to prove $(\forall n.(\text{L}(n) \rightarrow \text{L}(n+1)))$. That is, we have to derive L($n+1$) from L(n) for an arbitrary natural number n. Fortunately, we know how to do this. Just copy the derivation of, say L(3) from L(2), but start with an append formula in which the first operand is a list with $n+1$ elements, and cite L(n) where we would have cited L(3).

$$\begin{array}{cll}
 & \text{(len (append (}x_1\ x_2 \ldots x_{n+1}\text{) }ys\text{))} & \\
= & \text{(len (append (cons }x_1\text{ (}x_2 \ldots x_{n+1}\text{)) }ys\text{))} & \{\textit{cons}\} \\
= & \text{(len (cons }x_1\text{ (append (}x_2 \ldots x_{n+1}\text{) }ys\text{)))} & \{\textit{app1}\} \\
= & \text{(+ 1 (len (append (}x_2 \ldots x_{n+1}\text{) }ys\text{)))} & \{\textit{len1}\} \\
= & \text{(+ 1 (+ (len (}x_2 \ldots x_{n+1}\text{)) (len }ys\text{)))} & \{\text{L}(n)\} \\
= & \text{(+ (+ 1 (len (}x_2 \ldots x_{n+1}\text{))) (len }ys\text{))} & \{\text{+ associative}\} \\
= & \text{(+ (len (cons }x_1\text{ (}x_2 \ldots x_{n+1}\text{))) (len }ys\text{))} & \{\textit{len1}\} \\
= & \text{(+ (len (}x_1\ x_2 \ldots x_{n+1}\text{)) (len }ys\text{))} & \{\textit{cons}\}
\end{array}$$

An important point to notice in this proof is that we could not cite the {*cons*} equation to replace ($x_2 \ldots x_{n+1}$) with (cons x_2 ($x_3 \ldots x_{n+1}$)). The reason we could not do this is that we are trying to derived L($n+1$) from L(n) without making any assumptions about n other than the fact that it is a natural number. Since zero is a natural number, the list ($x_2 \ldots x_{n+1}$) could be empty, and the cons operation cannot deliver an empty list as its value.

In the next section, we will prove some properties of append that confirm its correctness with respect to a specification in terms of other operators. These properties, and in fact all properties of the append operator, can be derived from the append theorem. That theorem states properties of the append operation in two cases: (1) when the first operand is the empty list (the {*app0*} equation), and (2) when the first operand is a non-empty list (the {*app1*} equation). When the first operand is the empty list, the result must be the second operand, no matter what it is. When the first operand is not empty, it must have a first element. That element must also be the first element of the result. The other elements of the result are the ones you would get if you appended the rest of the first operand with the second operand.

Both of these properties are so straightforward and easy to believe that we would probably be willing to accept them as axioms, with no proof at all. It might come as a surprise that all of the other properties of the append operation can be derived from the two simple properties {*app0*} and {*app1*}. That is the power of mathematical induction. The two equations

Foundational	There is at least one non-inductive equation (that is, an equation that only refers to the operator being defined on one side of the equation).
Complete	All possible combinations of operands are covered by at least one equation in the definition.
Consistent	Combinations of operands covered by by two or more equations imply the same value for the operation.
Computational	In inductive equations (that is, equations that refer to the operator being defined on both sides of the equation), all invocations of the operator on the right-hand side of the equation have operands that are closer to the operands on the left-hand side of a non-inductive equation than the operands on the left-hand side of the equation.

Figure 4.2: Characteristics of Inductive Definitions

of the append theorem amount to an inductive definition of the append operator.

An inductive definition is circular in the sense that some of the equations in the definition refer to the operator on both sides of the equation. Most of the time, we think circular definitions are not useful, so it may seem surprising that they can be useful in mathematics. Some aren't, but some of them are, and you will gradually learn how to recognize and create useful, circular (that is, inductive) definitions.

It turns out that all functions that can be defined in software have inductive definitions in the style of the equations of the append theorem. The keys to an inductive definition of an operator are listed in Figure 4.2. All of the software we will discuss will take the form of a collection of inductive definitions of operators. That makes it possible to use mathematical induction as the fundamental tool in verifying properties of that software to a logical certainty.

This is not the only way to write software. In fact, most software is not written in terms of inductive definitions. But, properties of the software written using conventional methods cannot be derived using conventional logic. So, in terms of understanding what computers do and how they do it, inductive definitions provide solid footing. That is why we base our presentation on software written in terms of inductive definitions rather than by conventional methods.

4.3 Contatenation, Prefixes, and Suffixes

If you concatenate two lists, *xs* and *ys*, you would expect to be able to retrieve the elements of *ys* by dropping some of the elements of the concatenated lists. How many elements would you need to drop? That depends on the number of elements in *xs*. If there are n elements in *xs*, and you drop n elements from (append *xs ys*), you expect the result to be identical to the list *ys*. We can state that expectation by using an intrinsic operation in ACL2 with the arcane name “nthcdr”. The nthcdr operation takes two arguments: a natural number and a list. The formula (nthcdr n *xs*) delivers a list like *xs*, but without its first n elements. If *xs* has fewer than n elements, then the formula delivers the empty list.

The following equations state some simple properties of the nthcdr operation that we take as axioms.

Axioms {*nthcdr*}	
(nthcdr 0 *xs*) = *xs*	{*sfx0*}
(nthcdr n nil) = nil	{*sfx-nil*}
(nthcdr (+ n 1) (cons x *xs*)) = (nthcdr n *xs*)	{*sfx1*}

TODO: Rex: sfx1 isn't true, right? I'm not sure we want to introduce it as an axiom, if later we'll have to explain it isn't really true. Ruben: left as is for now, with (natp n) implicit, but inserted a comment about the type of n

Given this background, we state the expected relationship between the append and nthcdr operators in terms of a sequence of special cases. We will use S(n) as a shorthand for special case number n. There will be one case for each natural number.

$$\begin{array}{llll} \text{S}(n) \equiv (\text{equal} & (\text{nthcdr} & (\text{len}\ (x_1\ x_2 \ldots x_n)) \\ & & (\text{append}\ (x_1\ x_2 \ldots x_n)\ ys)) \\ & ys) & \end{array}$$

If S(n) to be true, regardless of what natural number n stands for, then the formula ($\forall n$.S(n)) is true. Since the universe of discourse of the predicate S is the natural number, mathematical induction may be useful in verifying that formula. All we need to do is to prove that (1) the formula S(0) is true and (2) the formula S($n+1$) is true under the assumption that S(n) is true, regardless of what natural number n stands for. Let's do that.

First, we prove S(0). When n is zero, the list (x_1 x_2 $\ldots x_n$) is empty, which is normally denoted by the symbol "nil". So, S(0) stands for the following equation.

$$\begin{array}{ll} \text{S}(0) \equiv (\text{equal} & (\text{nthcdr (len nil) (append nil } ys))\\ & ys) \end{array}$$

Following our usual practice when proving an equation, we start with the formula on one side and use previously known equations to gradually transform that formula to the one on the other side of the equation.

$$\begin{array}{lll} & (\text{nthcdr (len nil) (append nil } ys)) & \\ = & (\text{nthcdr (len nil) } ys) & \{\textit{app0}\} \text{ (see page 52)}\\ = & (\text{nthcdr } 0\ ys) & \{\textit{len0}\} \text{ (see page 49)}\\ = & ys & \{\textit{sfx0}\} \end{array}$$

That takes care of S(0). Next, we prove S($n+1$), assuming that S(n) is true.

$$\begin{array}{llll} \text{S}(n+1) \equiv (\text{equal} & (\text{nthcdr} & (\text{len } (x_1\ x_2\ \ldots x_{n+1}))\\ & & (\text{append } (x_1\ x_2\ \ldots x_{n+1})\ ys))\\ & ys) & \end{array}$$

$$\begin{array}{llll} & (\text{nthcdr} & (\text{len } (x_1\ x_2\ \ldots x_{n+1})) & \\ & & (\text{append } (x_1\ x_2\ \ldots x_{n+1})\ ys)) & \\ = & (\text{nthcdr} & (\text{len (cons } x_1\ (x_2\ \ldots x_{n+1}))) & \{\textit{cons}\} \text{ (page 48)}\\ & & (\text{append (cons } x_1\ (x_2\ x_2\ \ldots x_{n+1}))\ ys)) & \{\textit{cons}\}\\ = & (\text{nthcdr} & (+\ (\text{len } (x_2\ \ldots x_{n+1}))\ 1) & \{\textit{len1}\} \text{ (page 49)}\\ & & (\text{cons } x_1\ (\text{append } (x_2\ \ldots x_{n+1}))\ ys)) & \{\textit{app1}\} \text{ (page 52)}\\ = & (\text{nthcdr} & (\text{len } (x_2\ \ldots x_{n+1})) & \{\textit{sfx1}\}\\ & & (\text{append } (x_2\ \ldots x_{n+1})\ ys)) & \\ = & ys & & \{\text{S}(n)\} \end{array}$$

The last step in the proof is justified by citing S(n). This is a little tricky because the formula that S(n) stands for is not exactly the same as the formula in the next-to-last step of the proof. We interpret the formula (x_1 x_2 $\ldots x_n$) in the definition of S(n) to stand for any list with n elements. The elements in the list (x_2 $\ldots x_{n+1}$) are numbered 2 through $n+1$, which means there must be exactly n of them.

With this interpretation, the formula in the next-to-last step matches the formula in the definition of S(n), which makes it legitimate to cite S(n) to justify the transformation to *ys* in the last step of the proof. We will use this interpretation frequently in proofs. We refer to it as the "numbered-list interpretation", or {*nlst*} for short.

(x_m $\ldots x_n$) denotes a list with max($n-m+1$, 0) elements {*nlst*}

TODO: Rex: We may want to skip this paragraph for now. We can introduce this notation later, when needed. Ruben: right, done

Of course, if n is zero, or if *xs* is empty, (prefix n *xs*) must be the empty list. If n is non-zero natural number and *xs* is not empty, then the first element of (prefix n *xs*) must be the first element of *xs*, the the other elements must be the first $n-1$ elements of (rest *xs*). The following equations, which we take as axioms, put these expectations in formal terms. The formula (posp n) refers to the intrinsic ACL2 operator "posp". It is true if n is a non-zero natural number (that is, a strictly positive integer) and false otherwise.

Axioms {*prefix*}	
(prefix 0 xs) = nil	{*pfx0*}
(prefix n nil) = nil	{*pfx-nil*}
(prefix (+ n 1) (cons x xs)) = (prefix n xs)	{*pfx1*}

We can derive the prefix property of the append function from the equations for the prefix and append operations. The proof will cite mathematical induction. As before, we will use a shorthand for special case number n.

TODO: Rex: Should we replace xs with (x1 x2 ... xn). Ruben: right, done

P(n) $\equiv$ (equal (prefix (len (x_1 x_2 $\ldots x_n$))

```
        (append ($x_1$ $x_2$ ... $x_n$) $ys$))
  ($x_1$ $x_2$ ... $x_n$))
```

We will prove that P(0) is true, and also that P($n+1$) is true whenever P(n) is true. Then, we will cite mathematical induction to conclude that P(n) is true, regardless of which natural number n stands for.

```
P($n$) ≡ (equal (prefix (len nil)
                        (append nil $ys$))
                nil)
```

As in the proof of the append suffix theorem, we start with the formula on one side of the equation and use known equations to gradually transform that formula to the one on the other side of the equation.

```
    (prefix (len nil) (append nil $ys$))
=   (prefix 0 (append nil $ys$))          {len0} (see page 49)
=   nil                                   {pfx0}
```

That takes care of P(0). Next, we prove P($n+1$), assuming that P(n) is true.

```
P($n+1$) ≡ (equal (prefix (len ($x_1$ $x_2$ ... $x_{n+1}$))
                          (append ($x_1$ $x_2$ ... $x_{n+1}$) $ys$))
                  ($x_1$ $x_2$ ... $x_{n+1}$))
```

TODO: Rex: This following indentation isn't perfect, but it's close. I haven't figured out how to remove the vertical space before the tabbing, though I can probably hack it....

(prefix (len (x_1 x_2 . . . x_{n+1}))
        (append (x_1 x_2 . . . x_{n+1}) ys))

= {*cons*} (page 48)

(prefix (len (cons x_1 (x_2 . . . x_{n+1})))
        (append (cons x_1 (x_2 x_2 . . . x_{n+1})) ys))

= {*len1*} (page 49) {*app1*} (page 52)

(prefix (+ (len (x_2 . . . x_{n+1})) 1)
        (cons x_1 (append (x_2 . . . x_{n+1}) ys)))

= {*pfx1*}

(cons (first (cons x_1 (x_2 . . . x_{n+1})))
      (prefix (- (+ (len (x_2 . . . x_{n+1})) 1) 1)
              (rest (cons x_1 (append (x_2 . . . x_{n+1}) ys)))))

= {*first*} (page 48) {*arithmetic*} {*rest*} (page 48)

(cons x_1
      (prefix (len (x_2 . . . x_{n+1}))
              (append (x_2 . . . x_{n+1}) ys)))

= {P(n)}

(cons x_1
      (x_2 . . . x_{n+1}))

= (x_1 x_2 . . . x_{n+1}) {*cons*} (page 48)

At this point we know three important facts about the append function:

- additive length theorem: (len (append xs ys)) = (+ (len xs) (len ys))
- append-prefix theorem: (prefix (len xs) (append xs ys)) = xs
- append-suffix theorem: (nthcdr (len xs) (append xs ys)) = ys

Together, these theorems provide a deep level of understanding of the append operation. They give us confidence that it correctly concatenates lists. We refer to these theorems as "correctness properties" for the append operation. They are, of course, an infinite variety of other facts about the append operation. Their relative importance depends on how we are using the operation.

A property that is sometimes important to know is that concatenation is “associative”. That is, if there are three lists to be concatenated, you you could concatenate the first list with the concatenation of the last two. Or, you could concatenate the first two, then append that with the third.

Theorem {*app-assoc*}
(append xs (append ys zs)) = (append (append xs ys) zs)

Addition and multiplication of numbers are associative in an analogous way (but subtraction and division aren’t associative). Another way to say this is that the formula $(\forall n.\mathrm{A}(n))$ is true, where the predicate A is defined as follows.

$$\begin{array}{lll}\mathrm{A}(n) \equiv & (\text{equal} & (\text{append } (x_1\ x_2 \ldots x_n)\ (\text{append } ys\ zs)) \\ & & (\text{append } (\text{append } (x_1\ x_2 \ldots x_n)\ ys)\ zs)\end{array}$$

Putting it this way makes the theorem amenable to a proof by mathematical induction. We leave that as a something you can use to practice your proof skills.

TODO: Ruben: The ExerciseList tag doesn’t put in any vertical space, but should, I think.

Ex. 14 — Carry out a paper-and-pencil proof by mathematical induction of the {*app-assoc*} theorem.

TODO: next section will introduce defthmd and proofs using the ACL2 mechanized logic by replaying all of the theorems of this section in ACL2 notation

4.4 Mechanized Logic

The proofs we have been doing depend on matching grammatical elements in formulas against templates in axioms and theorems. The formulas are then transformed to equivalent ones with different grammatical structures. Gradually, we move from a starting formula to a concluding one to verify an equation for a new theorem.

It is easy to make mistakes in this detailed, syntax-matching process, but computers carry it out flawlessly. This relieves us from an obligation to focus with monk-like devotion on the required grammatical analysis. We can leave it to the computer count on having it done right.

There are several mechanized logic systems that people use to assist with proofs of the kind we have been doing. One of them is ACL2 (A Computational Logic for Applicative Common Lisp). Theorems for the ACL2 proof engine are stated in the same form as properties for the DoubleCheck testing facility in Dracula. ACL2 has a built-in strategy for finding inductive proofs, and for some theorems it succeeds in fully automating proofs. It also permits people to guide it through proofs while it pushes through all of the grammatical details.

To illustrate how this works, we will go the theorems discussed earlier in this chapter, one by one. The notation for stating theorems in ACL2 form will be familiar, but not identical to the one we have been using for our paper-and-pencil proofs. For one thing, it employs prefix notation throughout, and we have been using a mixture of prefix and infix.

Our first proof by mathematical induction verified the additive law of concatenation (see page 52). Our statement of the theorem asserted that a proposition L(n) is true, regardless of which natural number n stands for: $(\forall n.\mathrm{L}(n))$. L($n$) is a shorthand for the following formula:

$$\begin{aligned}\mathrm{L}(n) \equiv (&= (\text{len } (\text{append } (x_1\ x_2 \ldots x_n)\ ys)) \\ &(+\ (\text{len } (x_1\ x_2 \ldots x_n))\ (\text{len } ys)))\end{aligned}$$

We could have used the DoubleCheck facility of Dracula to run tests on this property.

```
(defproperty additive-law-of-concatenation-tst
    (xs :value (random-list-of (random-natural))
     ys :value (random-list-of (random-natural)))
  (= (len (append xs ys))
     (+ (len xs) (len ys))))
```

Of course, the DoubleCheck specification of this property cannot employ the informal notation of the numbered list interpretation (see page 60). Instead, the property simply uses a symbol xs to stand for the list. The property does not state the length of xs, but we know its length will be some natural number, n, so the property, as stated, has the same meaning as the formula that L(n) stands for.

The statement of the additive law as a theorem in the form required by ACL2 cannot use the informal notation, either. In fact, the theorem takes a form that is like the property specification, except for the :value portion and the keyword “defproperty”.

Theorem statements in ACL2 start with the “defthmd” keyword. After that comes a name for the theorem and the Boolean formula that expresses the meaning of the theorem, as illustrated in the following definition.

```
(defthmd additive-law-of-concatenation-thm
  (= (len (append xs ys))
     (+ (len xs) (len ys))))
```

The mechanized logic of ACL2 fully automates the proof of this theorem. It uses a built-in, heuristic procedure to find an induction scheme and pushes the proof through on its own. To see ACL2 in action, enter the above theorem in the program pane of the Dracula window (the upper left pane), press the start button in the ACL2 pane (the right half of the ACL2 window). When the Admit button lights up, press it.

After a short time, the theorem will turn green, indicating a successful proof. Details of the proof appear in the ACL2 pane. Later we will learn how to interpret some of these details, but for now, we are just looking for success (green coloring of the theorem in the program pane) or failure (red coloring).

Probably you can follow the above example to convert all theorems from this chapter into ACL2 theorem statements. Just to make sure, we will look at another one, then leave the rest for practice exercises.

The append-suffix theorem, which states that when the first argument in an append formula is a list of length n, then you can reconstruct the second argument by dropping n elements from the front of the concatenation. We stated this this theorem in the form $(\forall n.\mathrm{S}(n))$, where $\mathrm{S}(n)$ is a shorthand for the following formula.

$$\begin{array}{lll} \mathrm{S}(n) \equiv (\text{equal} & (\text{nthcdr} & (\text{len } (x_1\ x_2 \ldots x_n)) \\ & & (\text{append } (x_1\ x_2 \ldots x_n)\ ys)) \\ & ys) & \end{array}$$

The following definition specifies this theorem in ACL2 notation.

```
(defthmd append-suffix-thm
  (equal (nthcdr (len xs) (append xs ys))
         ys))
```

ACL2 can prove this theorem, but the proof requires knowing something about the algebra of numbers, such as the associative law of addition. Fortunately, someone has worked out a basic theory of numeric algebra in ACL2 terms, and we can take advantage of that by importing it into the working environment. To do this, we use a command called “include-book”. The name of the book with the theory we need is “arithmetic/top”, and it resides in the “system” directory of ACL2.

```
(include-book "arithmetic/top" :dir :system)
```

The include-book command makes that theory accessible to the mechanized logic. When you put the command above the append-suffix theorem in the program pane and press the start button and then the Admit All button, ACL2 succeeds in the proof without further assistance. For practice, try it yourself.

Ex. 15 — Define the {*append-prefix*} theorem in ACL2 notation, and use Dracula to run it through the mechanized logic. If you state it correctly, the proof should succeed.

Ex. 16 — Define the {*append associativity*} theorem in ACL2 notation, and use Dracula to run it through the mechanized logic. If you state it correctly, the proof should succeed.