Reasoning About Programs

Panagiotis Manolios
Northeastern University

March 3, 2016
Version: 94

1

Introduction

These lecture notes were developed for Logic and Computation, a freshman-level class taught at the College of Computer and Information Science of Northeastern University. Starting in Spring 2008, this is a class that all students in the college are required to take.

The goals of the Logic and Computation course are to provide an introduction to formal logic and its deep connections to computing. Logic is presented from a computational perspective using the ACL2 Sedan theorem proving system. The goal of the course is to introduce fundamental, foundational methods for modeling, designing and reasoning about computation, including propositional logic, recursion, induction, equational reasoning, termination analysis, rewriting, and various proof techniques. We show how to use logic to formalize the syntax and semantics of the core ACL2s language, a simple LISP-based language with contracts. We then use the ACL2s language to formally reason about programs, to model systems at various levels of abstraction, to design and specify interfaces between systems and to reason about such composed systems. We also examine decision procedures for fragments of first-order logic and how such decision procedures can be used to analyze models of systems.

The students taking the Logic and Computation class have already taken a programming class in the previous semester, using Racket. The course starts by reviewing some basic programming concepts. The review is useful because at the freshman level students benefit from seeing multiple presentations of key concepts; this helps them to internalize these concepts. For example, in past semesters I have asked students to write very simple programs (such as a program to append two lists together) during the first week of classes and a surprisingly large number of them produce incorrect code.

We introduce the ACL2s language. This is the language we use throughout the semester. Since ACL2s is very similar to Racket, this happens simultaneously with the programming review. During lectures, I will often point out the similarities and differences between these languages.

We introduce the semantics of the ACL2s language in a mathematical way. We show the syntax and semantics of the core language. We provide enough information so that students can determine what sequence of glyphs form a well-formed expression and how to formally evaluate well-formed expressions potentially containing user-defined functions with constants as arguments (this is always in a first-order setting). This is a pretty big jump in rigor for students and is advanced material for freshmen students, but they already have great intuitions about evaluation from their previous programming class. A key to helping students understand the material is to motivate and explain it by connecting it to their strong computational intuitions.

The lecture notes are sparse. It would be great to add more exercises, but I have not done that yet. Over the course of many years, we have amassed a large collection of homework problems, so students see lots of exercises, and working through these exercises is a great

way for them to absorb the material, but the exercises are not in the notes. You can think of the lecture notes as condensed notes for the course that are appropriate for someone who knows the material as a study guide. The notes can also be used as a starting point by students, who should mark them up with clarifications as needed when they attend lectures. I advise students to read the lecture notes before class. This way, during class they can focus on the lecture instead of taking notes, they are better prepared to ask for clarifications and they can better judge what notes they should take (if any).

When I started teaching the class, I used the ACL2 book, *Computer-Aided Reasoning, An Approach* by Kaufmann, Manolios and Moore. However, over the years I became convinced that using an untyped first-order logic was not the optimal way of introducing logic and computation to students because they come in with a typed view of the world. That's not to say they have seen type theory; they have not. But, they are surprised when a programming language allows them to subtract a string from a rational number. Therefore, with the help of my Ph.D. student Harsh Chamarthi, I have focused on adding type-like capabilities to ACL2s. Most notably, we added a new data definition framework to ACL2s that supports enumeration, union, product, record, map, (mutually) recursive and custom types, as well as limited forms of parametric polymorphism. We also introduced the `defunc` macro, which allows us to formally specify input and output contract for functions. These contracts are very general, *e.g.*, we can specify that `/` is given two rationals as input, and that the second rational is not 0, we can specify that `zip` is given two lists of the same length as input and returns a list of the same length as output and so on. Contracts are also checked statically, so ACL2s will not accept a function definition unless it can prove that the function satisfies its contracts and that for every legal input and every possible computation, it is not possible during the evaluation of the function being defined to be in a state where some other function is poised to be evaluated on a value that violates its input contract. I have found that a significant fraction of erroneous programs written by students have contract violations in them, and one of the key things I emphasize is that when writing code, one needs to think carefully about the contracts of the functions used and why the arguments to every function call satisfy the function's contract. Contracts are the first step towards learning how to specify interfaces between systems. With the move to contracts, the ACL2 book became less and less appropriate, which led me to write these notes.

I have distributed these notes to the students in Logic and Computation for several years and they have found lots of typos and have made many suggestions for improvement. Thanks and keep the comments coming!

6

Induction

Terminating functions give rise to induction schemes.

Consider the following function:

```
(defunc nind (n)
  :input-contract (natp n)
  :output-contract t
  (if (equal n 0)
      0
    (nind (- n 1))))
```

This function is admissible. Given a natural number `n` it counts down to 0 and returns, therefore it is terminating.

Induction is justified by termination: every terminating function gives rise to an induction scheme. For example, suppose we want to prove φ using the induction scheme from `(nind n)`. Our proof obligations are:

1. `(not (natp n))` $\Rightarrow \varphi$
2. `(natp n)` $\wedge$ `(equal n 0)` $\Rightarrow \varphi$
3. `(natp n)` $\wedge$ `(not (equal n 0))` $\wedge \varphi|_{\texttt{((n n-1))}} \Rightarrow \varphi$

A bit of terminology. Cases 1 and 2 are *base cases*. Case 3 is an *induction step* because we get to assume that φ holds on smaller values. The last hypothesis of case 3, $\varphi|_{\texttt{((n n-1))}}$, is called an *induction hypothesis*. Notice that the induction hypothesis is what distinguishes induction from case analysis, *i.e.*, we could try to prove φ using case analysis as follows:

1. `(not (natp n))` $\Rightarrow \varphi$
2. `(natp n)` $\wedge$ `(equal n 0)` $\Rightarrow \varphi$
3. `(natp n)` $\wedge$ `(not (equal n 0))` $\Rightarrow \varphi$

The three cases above are exhaustive. Induction is more powerful than case analysis because it also allows us to assume the induction hypothesis.

Back to induction.

Why does induction work?

First, let me describe a proof technique that I'm going to use (and that is widely used): Proof by contradiction.

What does a proof by contradiction look like? It's just a "cheap" propositional trick. Let's say that we are trying to prove:

$$\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi$$

We do this by assuming the negation of the consequent and deriving a contradiction, *i.e.*, we instead prove:

$$\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \neg\varphi \Rightarrow \mathit{false}$$

Note that these two statements are equivalent by propositional logic. To see this, note

$$A \wedge B \Rightarrow C \equiv A \wedge \neg C \Rightarrow \neg B$$

Then apply the above to

$$\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \mathit{true} \Rightarrow \varphi$$

Proof by contradiction is often referred to *reductio ad absurdum*, Latin for "reduction to the absurd."

Here is a great quote about proof by contradiction from Godfrey Harold Hardy's *A Mathematician's Apology* (1940).

> Reductio ad absurdum, which Euclid loved so much, is one of a mathematician's finest weapons. It is a far finer gambit than any chess gambit: a chess player may offer the sacrifice of a pawn or even a piece, but a mathematician offers the game.

Back to our question.

Suppose that we prove the above three cases, but φ is not valid.

Then, the set, S, of objects in the ACL2 universe for which φ does not hold is non-empty. The set can only contain positive natural numbers, as case 1 rules out there being any non-natural numbers in S and case 2 rules out 0 being in S. Consider the smallest (natural) number $s \in S$. Now instantiate the induction step (3), replacing `n` by s:

$$\texttt{(natp } s\texttt{)} \wedge \texttt{(not (equal } s\ \texttt{0))} \wedge \varphi|_{\texttt{((n } s-1\texttt{))}} \Rightarrow \varphi|_s$$

Notice that $(\varphi|_{\texttt{((n n-1))}})|_{\texttt{((n } s\texttt{))}} = \varphi|_{\texttt{((n } s-1\texttt{))}}$. But, we have that `(natp` s`)` holds and $s \neq 0$. By the minimality of s, $\varphi|_{\texttt{((n } s-1\texttt{))}}$ also holds. By MP and the above, so does $\varphi|_s$. So, $s \notin S$! That is our contradiction, so our assumption that φ is false led to a contradiction, *i.e.*, φ is in fact valid.

Two observations.

1. We used a nice property of the natural numbers: they are *terminating*: every decreasing sequence is finite. This is equivalent to saying that every non-empty subset has a minimal element. Induction works as long as we have termination. We can prove termination for any kind of ACL2s function, using measure functions. For example, measure functions allow us to prove termination of functions that operate on lists. Notice that they do this by relating what happens on lists to numbers.

2. We used a proof by counterexample. Why do people use proofs by counterexample? It seems like an elaborate way of proving φ. In some sense it is, but it is a nice technique to have in your arsenal because it often helps you focus on the goal: prove false.

Here is yet another way to see why induction works. Suppose, as before, that we prove the three proof obligations above. Now, as before, the first two proof obligations directly show that φ holds for all non-natural numbers and for 0. If we instantiate the third proof obligation, the induction step, with the substitution `((n 1))`, then the hypotheses hold (as `(natp 1)`, $1 \neq 0$, and $\varphi|_{\texttt{((n 0))}}$ all hold), so by MP $\varphi|_{\texttt{((n 1))}}$ holds. Notice that we use the

induction step to go from $\varphi|_{((\texttt{n}\ 0))}$ to $\varphi|_{((\texttt{n}\ 1))}$. Similarly, we can use $\varphi|_{((\texttt{n}\ 1))}$ to get $\varphi|_{((\texttt{n}\ 2))}$ and so on for all the natural numbers. We have just shown that for any natural number i, $\varphi|_{((\texttt{n}\ i))}$ holds, so φ holds for all objects in the ACL2s universe. This is a direct proof that proof by induction is sound.

Now that we have motivated induction, here it is in its full glory.

Induction Scheme: Suppose we are given a function definition of the form:

```
(defunc foo (x1 ... xn)
  :input-contract ic
  :output-contract oc
  (cond (t1 c1)
        (t2 c2)
        ...
        (tm cm)
        (t cm+1)))
```

where none of the $\texttt{c}_i$'s have any `if`s in them.

Notice that any function definition can be written in this form.

If $\texttt{c}_i$ contains a call to `foo`, we say it is a *recursive* case; otherwise it is a *base* case. If $\texttt{c}_i$ is a recursive case, then it includes at least one call to `foo`. Say there are R_i calls to `foo` and they are $\texttt{foo}_i^j$ where $1 \leq j \leq R_i$. Let s_i^j be the substitution such that $(\texttt{foo}\ \texttt{x}_1\ \ldots\ \texttt{x}_n)|_{s_i^j} = \texttt{foo}_i^j$.

Let $\texttt{t}_{m+1}$ be `t`.

Let Case_i be $\texttt{t}_i \wedge \neg\texttt{t}_j$ for all $j < i$, *e.g.*,

- Case_1 is $\texttt{t}_1$
- Case_2 is $\texttt{t}_2 \wedge\ \neg\texttt{t}_1$
- Case_3 is $\texttt{t}_3 \wedge\ \neg\texttt{t}_1 \wedge\ \neg\texttt{t}_2$
- Case_{m+1} is $\texttt{t} \wedge \neg\texttt{t}_1 \wedge\ \neg\texttt{t}_2 \wedge \cdots \wedge\ \neg\texttt{t}_m$

The function `foo` gives rise to the following induction scheme:

To prove φ, you can instead prove

1. $\neg\texttt{ic} \Rightarrow \varphi$
2. For all $\texttt{c}_i$ that are base cases: $[\texttt{ic} \wedge \text{Case}_i] \Rightarrow \varphi$
3. For all $\texttt{c}_i$ that are recursive cases: $[\texttt{ic} \wedge \text{Case}_i \wedge \bigwedge_{1 \leq j \leq R_i} \varphi|_{s_i^j}] \Rightarrow \varphi$

We can play this game in reverse. For example, if I were to ask you to write a function that gives rise to the following induction scheme:

1. `(not (natp n))` $\Rightarrow \varphi$
2. `(natp n)` $\wedge$ `(equal n 0)` $\Rightarrow \varphi$
3. `(natp n)` $\wedge$ `(not (equal n 0))` $\wedge\ \varphi|_{((n\ n-1))} \Rightarrow \varphi$

Exercise 6.1 *One correct answer is* `nind`*, but there are infinitely many correct answers. Show this.*

6.1 Induction Examples

Recall the definition of `sumn`.

```
(defunc sumn (n)
  :input-contract (natp n)
  :output-contract (natp (sumn n))
  (if (equal n 0)
      0
    (+ n (sumn (- n 1)))))
```

Let's prove

$$\texttt{(sumn n)} = \texttt{n}(\texttt{n}+1)/2$$

Recall that the first thing we do is to contract check conjectures. After fixing the above conjecture, we get:

$$\texttt{(natp n)} \Rightarrow \texttt{(sumn n)} = \texttt{n}(\texttt{n}+1)/2 \tag{6.1}$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `sumn`. When you find yourself in such a situation, induct!

What induction scheme?

The one that the data definition gives rise to. In this case, we are reasoning about natural numbers, so `nind`'s induction scheme (which is the same as `sumn`'s induction scheme).

Our proof obligations are then:

1. `(not (natp n))` $\Rightarrow$ (6.1)
2. `(natp n)` $\wedge$ `n` $= 0 \Rightarrow$ (6.1)
3. `(natp n)` $\wedge$ `n` $\neq 0 \wedge$ (6.1)$|_{\texttt{((n (- n 1)))}} \Rightarrow$ (6.1)

Notice that the proof now goes through with just equational reasoning.

So, since we know how to do equational reasoning, we will skip the equational proofs, but you can and should fill them in.

Let's now reason about the following function definition.

```
(defunc app (a b)
  :input-contract (and (listp a) (listp b))
  :output-contract (and (listp (app a b))
                        (equal (len (app a b))
                               (+ (len a) (len b))))
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))
```

We want to prove that `app` is associative.

$$\texttt{(app (app a b) c)} = \texttt{(app a (app b c))}$$

Contract checking gives:

$$\texttt{(listp a)} \wedge \texttt{(listp b)} \wedge \texttt{(listp c)} \Rightarrow \texttt{(app (app a b) c)} = \texttt{(app a (app b c))} \quad (6.2)$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `app`. When you find yourself in such a situation, induct!

What induction scheme?

The one that the data definition gives rise to. In this case, we are reasoning about lists, so `listp`'s induction scheme. Recall

```
(defunc listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (if (consp l)
      (listp (rest l))
    (equal l ())))
```

So the induction scheme is:

1. $\neg\texttt{t} \Rightarrow (6.2)$
2. $\texttt{t} \wedge \texttt{(consp a)} \wedge (6.2)|_{\texttt{((a (rest a)))}} \Rightarrow (6.2)$
3. $\texttt{t} \wedge \neg\texttt{(consp a)} \Rightarrow (6.2)$

This is equivalent to:

1. $\neg\texttt{(consp a)} \Rightarrow (6.2)$
2. $\texttt{(consp a)} \wedge (6.2)|_{\texttt{((a (rest a)))}} \Rightarrow (6.2)$

Exercise 6.2 *Notice that the variables we use in proofs are irrelevant,* e.g.*, even though* `listp` *was defined over* `l`*, we can apply induction using* `a` *instead. Explain why this is the case.*

So, here we go. If you expand out the proof obligations, we have a problem we've seen before! Do the induction step.

Exercise 6.3 *Assume that the output contract for* `app` *is* `t`*. This version of* `app` *is admissible. Using this version of* `app`*, use induction to prove the first conjunct in the output contract of the definition given earlier (*`(listp (app x y))`*).*

Exercise 6.4 *Assume that the output contract for* `app` *is* `t`*. This version of* `app` *is admissible. Using this version of* `app`*, use induction to prove the second conjunct in the output contract of the definition given earlier.*

Exercise 6.5 *Play the same game of proving the output contracts of the functions we have defined. Some will require induction, but some can be proved using just equational reasoning.*

Exercise 6.6 *Formalize (using ACL2s) and prove the following theorem (n is a natural number):*

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Next, we will play around with `rev`. Here is the definition.

```
(defunc rev (x)
  :input-contract (listp x)
  :output-contract (and (listp (rev x))
                        (equal (len (rev x))
                               (len x)))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))
```

Now we want to prove:

$$\texttt{(rev (rev x))} = \texttt{x}$$

Contract checking gives:

$$\texttt{(listp x)} \Rightarrow \texttt{(rev (rev x))} = \texttt{x} \tag{6.3}$$

We can't prove this theorem using equational reasoning. Why?

Because we don't know how many times to expand `rev`. When you find yourself in such a situation, induct!

What induction scheme?

The one that the data definition gives rise to. In this case, we are reasoning about lists, so `listp`'s induction scheme, which is:

1. $\neg$`(consp x)` $\Rightarrow$ (6.3)
2. `(consp x)` $\wedge$ (6.3)$|_{\texttt{((x (rest x)))}}$ $\Rightarrow$ (6.3)

Try the proof. You will get stuck.

Now what? Well, we need a lemma. If we had the following:

$$\texttt{(listp x)} \wedge \texttt{(listp y)} \Rightarrow \texttt{(rev (app x y))} = \texttt{(app (rev y) (rev x))} \tag{6.4}$$

we could finish the proof.

Let's assume we have it and then finish the proof.

So, notice that sometimes in the process of proving a theorem by induction, we have to prove lemmas in order for the proof to go through.

Exercise 6.7 *Prove* (6.4). *Use the induction scheme* `(listp x)` *gives rise to.*

In the proof of (6.4), we needed to prove

$$\texttt{(listp x)} \Rightarrow \texttt{(app x nil)} = \texttt{x}$$

So, even the proof of the lemma requires a lemma. How far can this go? Far! It's recursive.

Exercise 6.8 *What induction scheme does this give rise to?*

```
(defunc fib (n)
  :input-contract (posp n)
  :output-contract (posp (fib n))
  (cond ((equal n 1)
         1)
        ((equal n 2)
         2)
        (t (+ (fib (- n 1)) (fib (- n 2))))))
```

Let's prove the following:

$$\texttt{(fib n)} \geq n$$

Contract checking gives us:

$$\texttt{(posp n)} \Rightarrow \texttt{(fib n)} \geq n$$

Now what?

Can we prove this using induction on the natural numbers? Try it. The base case is trivial, but the induction step winds up requiring case analysis.

A better idea is to use the induction scheme `fib` gives rise to.

Why should you not be surprised? Well, because we didn't define `fib` using the data definition for `Nat`. `fib` is an example of generative recursion.

The point is that the induction scheme one should use to prove a conjecture is often related to the recursion schemes of the functions appearing in the conjecture. If all of the functions in the conjecture are based on a common recursion scheme (or design recipe), then the induction schemes will also be based on the same recursion scheme.

Exercise 6.9 *Prove the previous conjecture using the induction scheme* `fib` *gives rise to.*

6.2 Data-Function-Induction Trinity

Every admissible recursive definition leads to a valid induction scheme. What underlies both recursion and induction is *termination*. So, terminating functions give us both recursion schemes and induction schemes.

Notice a wonderful connection.

The data-function-induction (DFI) trinity:

1. Data definitions give rise to predicates recognizing such definitions. These predicates must be shown to terminate. (Otherwise they are inadmissible by the Definitional Principle.) Their bodies give rise to a *recursion scheme*, *e.g.*, `listp` gives rise to the common recursion scheme for iterating over a list.

2. Functions over these data types are defined by using the *recursion scheme* as a template. Templates allow us to define correct functions by assuming that the function we are defining works correctly in the recursive case. For example, in the definition of `app`, we get to assume that `app` works correctly in the recursive case, even if its

first input has 1,000,000 elements, *i.e.*, we get to assume that `app` applied to 999,999 elements works and all we have to do is to figure out what to do with the first element. This is about as simple an extension to straight-line code as we can imagine. Recursion provides us with a *significant* increase in expressive power, allowing us to define many functions that are not expressible using only straight-line code.

3. The *Induction Principle*: Proofs by induction involving such functions and data definitions should use the same *recursion scheme* to generate proof obligations. Non-recursive cases are proven directly. For each recursive case, we assume the theorem under *any* substitutions that map the formals to arguments in that recursive call. Induction provides us with a *significant* increase in theorem proving power over equational reasoning, analogous to the increase in definitional power we get when we move from straight-line code to recursive code. Notice also that induction and recursion are tightly related, *e.g.*, when defining recursive functions, we get to assume that the function works on smaller inputs; when proving theorems with induction, we get to assume that the theorem holds on smaller inputs (the induction hypothesis).

6.3 The Importance of Termination

Notice how important termination turns out to be.

1. Termination is the non-trivial proof obligation for admitting function definitions.
2. Termination is what justifies common recursion schemes and the design recipe.
3. Termination is what justifies generative recursive function definitions.
4. Complexity analysis is just a refinement of termination.
5. Termination is what justifies mathematical induction.
6. Terminating functions give rise to induction schemes. In fact, the only induction schemes we will use are the ones we can extract from terminating functions.

Exercise 6.10 *Consider the following non-terminating function definition.*

```
(defunc f (x)
  :input-contract t
  :output-contract t
  (f x))
```

Were we to admit `f` *(which we really can't because it is non-terminating), we would get the definitional axiom* `(f x) = (f x)`.

We have seen that this axiom does not lead to unsoundness. However, the "induction scheme" this function gives rise to does lead to unsoundness. Prove `nil` *using the induction scheme* `f` *gives rise to. Notice that there is a stronger relationship between induction and termination than there is between admissibility and termination. This relationship is an important reason why ACL2s does not admit non-terminating function definitions.*

6.4 Generalization

Consider the following definitions.

```
(defunc in (a X)
  :input-contract (listp x)
  :output-contract (booleanp (in a X))
  (cond ((endp x) nil)
        ((equal a (first X)) t)
        (t (in a (rest X)))))

(defunc subsetp (x y)
  ; checks if every element in x is in y
  :input-contract (and (listp x) (listp y))
  :output-contract (booleanp (subsetp x y))
  (cond ((endp x) t)
        ((in (first x) y)
         (subsetp (rest x) y))
        (t nil)))
```

Try to prove the following theorem:

$$\texttt{(listp x)} \Rightarrow \texttt{(subsetp x x)} \tag{6.5}$$

Notice that we can't prove this by induction. Why? Because whatever we do, we have to substitute for `x` and we want to distinguish the two occurrences of `x`. Unfortunately, we can't do that.

The solution?

Generalize: prove a theorem that we can prove by induction and that can then be used to prove the theorem we really want.

Here is the generalization:

$$\texttt{(listp x)} \wedge \texttt{(listp y)} \wedge \texttt{(subsetp x y)} \Rightarrow \texttt{(subsetp x (cons a y))} \tag{6.6}$$

Now, we can prove (6.6) using induction.

Exercise 6.11 *Prove* (6.6)

Exercise 6.12 *Prove* (6.5)

Exercise 6.13 *Prove* $\texttt{(listp x)} \wedge \texttt{(listp y)} \wedge \texttt{(listp z)} \wedge \texttt{(subsetp x y)} \wedge \texttt{(subsetp y z)} \Rightarrow \texttt{(subsetp x z)}$

6.5 Reasoning About Accumulator-Based Functions

Let's start with a simple definition we have already seen.

```
(defunc rev (x)
  :input-contract (listp x)
```

```
  :output-contract (listp (rev x))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))
```

The problem with this definition is that it requires $O(n^2)$ conses. Why?

Because `(app x y)` requires `(len x)` conses (as we have seen previously).

In the recursive case of `rev`, we have `app` applied to `(rev (rest x))` which requires `(len (rev (rest x)))` conses plus one cons for the list, *i.e.*, `(len x)` conses. Since `rev` is called on `x`, then `(rest x)`, then `(rest (rest x))`, ..., it requires `(len x)` + `(len x)`-1 + `(len x)`$-2 + \ldots + 1$ conses, which is $O(n^2)$, for $n =$ `(len x)`.

This is a horrible function from an efficiency point of view, so we want to do better. One way of doing better is to define a tail-recursive function that uses an accumulator.

Here is such a definition.

```
(defunc revt (x acc)
  :input-contract (and (listp x) (listp acc))
  :output-contract (listp (revt x acc))
  (if (endp x)
      acc
    (revt (rest x) (cons (first x) acc))))
```

But, we want a function with the same interface as `rev` and `revt` takes 2 arguments. Hence, we define:

```
(defunc rev* (x)
  :input-contract (listp x)
  :output-contract (listp (rev* x))
  (revt x nil))
```

Exercise 6.14 *Show that* `(rev* l)` *requires only* $O(n)$ *conses, where* $n =$ `(len l)`.

We are now in a situation that computer scientists often find themselves in. We have one function definition `rev` that is simple, but inefficient. We also have another function definition that is more complex, but efficient. We want to show that these two functions are related in some way.

What relationship do we want to establish between `rev*` and `rev`?

Let's prove that they are equal.

$$\texttt{(listp x)} \Rightarrow \texttt{(rev* x)} = \texttt{(rev x)} \tag{6.7}$$

Is it true?

Can we solve this with equational reasoning? No. Why not?

Notice that proving (6.7) will require proving:

$$\texttt{(listp x)} \Rightarrow \texttt{(revt x nil)} = \texttt{(rev x)} \tag{6.8}$$

It is the recursive definitions that we have to worry about!

We will try to prove correctness using what we already know. Our proof attempt will run into several hurdles and we will have to analyze what went wrong and how to proceed. By the end of this section, we will have constructed a little recipe for reasoning about accumulator-based functions in the future.

Let's try proving (6.8) using the induction scheme `listp` gives rise to.

1. ¬`(consp x)` ⇒ (6.8)
2. `(consp x)` ∧ (6.8)$|_{\texttt{((x (rest x)))}}$ ⇒ (6.8)

Let's try to prove this.
Proof?
The base case is simple. Here is an attempt at proving the induction step.
The context is:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp (rest x))` ⇒ `(revt (rest x) nil)` = `(rev (rest x))`

C4. `(listp (rest x))` {C1, C2, Def `listp`}

C5. `(revt (rest x) nil)` = `(rev (rest x))` {C3, C4, MP}

Proof:

```
    (revt x nil)
```

= { By C1 and the definition of `revt` }

```
    (revt (rest x) (cons (first x) nil))
```

But, now what? Our induction hypothesis tells us something about

```
(revt (rest x) nil)
```

but we really need to know something about

```
(revt (rest x) (cons (first x) nil))
```

so we're stuck. The point is that when we expand

```
(revt x nil)
```

we get an expression that is of the form

```
(revt ... (cons ...))
```

The second argument is not `nil`, but any way of instantiating the theorem we want to prove (as we do to get the induction hypothesis) will give us a `nil` in the second argument.

We need a handle on that second argument, but we're not going to get it if the theorem we want to prove has a `nil` there; we need a variable. We call this a generalization step because we are now going to prove a theorem about `(revt x acc)`, whereas before we were proving a theorem about a special case of the above, namely about `(revt x nil)`. But, now we have to figure out what the new theorem we want to prove is.

`(listp x)` ∧ `(listp acc)` ⇒ `(revt x acc)` = `???`

What is `???` ? Think about the role of `acc` in the definition of `revt`. The accumulator corresponds to a partial result: it should be the reverse of the elements of the original argument to `revt` that we have seen already, so we wind up with:

$$\texttt{(listp x)} \wedge \texttt{(listp acc)} \Rightarrow \texttt{(revt x acc)} = \texttt{(app (rev x) acc)} \qquad (6.9)$$

Suppose we prove (6.9). Can we use it to prove (6.7) and (6.8)?
Yes. Why?
The base case is simple. Here is a proof of the induction step. First, our context is:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp acc)`

C4. `(listp (rest x))` $\wedge$ `(listp acc)` $\Rightarrow$ `(revt (rest x) acc)` = `(app (rev (rest x)) acc)`

C5. `(listp (rest x))` {C1, C2, Def `listp`}

C6. `(revt (rest x) acc)` = `(app (rev (rest x)) acc)` {C5, C3, C4, MP}

Here is the proof. It starts the same way as before.

`(revt x acc)`

= { By C1 and the definition of `revt` }

`(revt (rest x) (cons (first x) acc))`

But, now we have yet another problem. The induction hypothesis doesn't match the above, since, as stated it is

`(revt (rest x) acc)` = `(app (rev (rest x)) acc)`

and we don't want `acc` as the second argument of `revt`; we want `(cons (first x) acc)`.

So, we can either define a function that gives rise to this induction scheme, or we can see if we have such a function available to us. In fact, we do: `revt`! So, let's use the induction scheme `revt` gives rise to. That gives us the following context:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp acc)`

C4. `(listp (rest x))` $\wedge$ `(listp (cons (first x) acc))` $\Rightarrow$ `(revt (rest x) (cons (first x) acc))` = `(app (rev (rest x)) (cons (first x) acc))`

C5. `(listp (rest x))` {C1, C2, Def `listp`}

C6. `(listp (cons (first x) acc))` {C3, Def `listp`}

C7. `(revt (rest x) (cons (first x) acc))` = `(app (rev (rest x)) (cons (first x) acc))` {C5, C6, C4, MP}

Proof:

```
    (revt x acc)
```

= { By C1 and the definition of `revt` }

```
    (revt (rest x) (cons (first x) acc))
```

= { By C7 }

```
    (app (rev (rest x)) (cons (first x) acc))
```

= { Def of `app` (working on pulling `acc` out to match RHS) }

```
    (app (rev (rest x)) (app (list (first x)) acc))
```

= { Associativity of `app` (now we pull `acc` out) }

```
    (app (app (rev (rest x)) (list (first x))) acc)
```

= { Def of `rev`, C1 }

```
    (app (rev x) acc)
```

Based on our experience above, here is a little recipe for reasoning about accumulator-based functions.

Part 1: Defining functions

1. Start with a function `f`.
2. Define `ft`, a tail-recursive version of `f` with an accumulator.
3. Define `f*`, a non-recursive function that calls `ft` and is logically equivalent to `f`, *i.e.*, this is a theorem
$$hyps \Rightarrow \texttt{(f* ...)} = \texttt{(f ...)}$$

Part 2: Proving theorems

4. Identify a lemma that relates `ft` to `f`. It should have the following form:
$$hyps \Rightarrow \texttt{(ft ... acc)} = \ldots\ \texttt{(f ...)}\ldots$$
Remember that you have to generalize, so all arguments to `ft` should be variables (no constants). The RHS should include `acc`.
5. Assuming that the lemma in 4 is true, and using only equational reasoning, prove the main theorem
$$hyps \Rightarrow \texttt{(f* ...)} = \texttt{(f ...)}$$
If you have to prove lemmas, prove them later.
6. Prove the lemma in 4. Use the induction scheme `ft` gives rise to.
7. Prove any remaining lemmas.

You might wonder why we bother to define `f*`? So that we can use `f*` as a replacement for `f`: `ft` won't do because it has a different signature than `f`.

You might want to swap steps 5 and 6. Don't because you want to first make sure that the lemma from 4 is the one you need.

If you want to swap steps 6 and 7, that's fine.