Reasoning About Programs

Panagiotis Manolios
Northeastern University

March 1, 2019
Version: 107

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, specifying and reasoning about computation. The topics covered include propositional logic, recursion, contracts, testing, induction, equational reasoning, termination analysis, term 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, in 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 students produce incorrect code.

During the programming review, we introduce the ACL2s language. This is the language we use throughout the semester and it is similar to Racket. The syntax and semantics of the core ACL2s language are presented in a mathematical way. 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. This intuition helps them understand the rigorous presentation of the syntax and semantics, which in turns helps strengthen their programming abilities.

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 and they are better prepared to ask for clarifications.

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 contracts 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:

```
(definec nind (n :nat) :nat
  (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 of `(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. When reasoning about programs, case analysis is a very useful proof technique, which we now define. Notice that it is the natural generalization of the synonymous proof technique we defined in the context of propositional logic.

Case Analysis: If ψ is a formula and $\varphi_1, \ldots, \varphi_n$ are formulas such that `(or` $\varphi_1\ \ldots\ \varphi_n$`)` is valid, then ψ is valid iff all of $(\varphi_1 \Rightarrow \psi), \ldots, (\varphi_n \Rightarrow \psi)$ are valid.

The intuition is the same as before. We are proving that ψ holds by considering the cases $\varphi_1, \ldots, \varphi_n$ and since the cases are exhaustive ($\varphi_1 \vee \cdots \vee \varphi_n \equiv t$), ψ always holds.

A commonly occurring example of where case analysis is useful is when we are proving a theorem of the following form.

$$\varphi_1 \vee \cdots \vee \varphi_n \ \Rightarrow\ \psi$$

It is often a good idea to use case analysis to instead prove the following set of (individually) simpler theorems.

$$\varphi_1 \;\Rightarrow\; \psi,\; \ldots,\; \varphi_n \;\Rightarrow\; \psi$$

Induction is more powerful than case analysis because it also allows us to assume the induction hypothesis. This makes all the difference in the world when reasoning about programs.

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: This is just a "cheap" propositional trick. Let's say that we are trying to prove the validity of the formula:

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

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: why does induction work?

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:

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

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 $\varphi|_{\texttt{((n }s\texttt{))}}$. 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 contradiction. Why do people use proofs by contradiction? 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 Schemes: 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)))
```

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

We will mostly restrict ourselves to functions where none of the $\texttt{t}_i$'s have recursive calls and none of $\texttt{c}_i$'s have any `if`s (including any macros that expand to `if`s).

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 $(\texttt{foo}\ \texttt{x}_1\ \ldots\ \texttt{x}_n)|_{\sigma_i^j}$, for $1 \leq j \leq R_i$ (the σ_i^j's are substitutions).

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 an *induction scheme* that is parameterized by a formula φ. The induction scheme of $(\texttt{foo}\ \texttt{x}_1\ \ldots\ \texttt{x}_n)$ for φ consists of the following formulas:

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

Proof by Induction: Let φ be any formula and let `f` be an n-ary function. If all of the formulas in the induction scheme of (`f` $\texttt{x}_1$ ... $\texttt{x}_n$) for φ are valid, then so is φ.

We allow one minor generalization that we did not formalize to avoid notational clutter. We allow you to use induction schemes for functions with any distinct variables as arguments. For example, the induction scheme of `(nind k)` is generated as described above, but we make believe that `nind` was defined using `k` instead of `n`, so we wind up with the induction scheme show previously, after applying the substitution `((n k))` to everything but φ. This is useful when we have formulas with many variables or with variables that differ from those used to define the function whose induction schemes we are using.

We saw how to go from functions to induction schemes, but we can also play this game in reverse. Consider the following exercise.

Exercise 6.1 *Provide 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 6.2 *One correct answer to the previous exercise is* `nind`*, but there are infinitely many correct answers. Show this.*

Exercise 6.3 *Prove that no function 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))` $\wedge$ $\varphi|_{((n\ n+1))}$ $\Rightarrow\ \varphi$

Why do we not allow "induction schemes" like the one in the above exercise? Because they are not sound! Remember that induction works because of termination. The only "functions" that give rise to the above "induction scheme" are nonterminating functions.

Exercise 6.4 *Prove that the "induction scheme" from Exercise 6.3 is not sound,* i.e.*, use the "induction scheme" to prove* `nil`*.*

We end by noting that ACL2s generates induction schemes from function definitions in a way that is very similar to what was explained here, but there are some differences. If you use the theorem prover and ask it to perform a particular induction and you wind up with proof obligations that seem strange, check what induction scheme was generated and if it is not what you expected, then read the documentation on `induction`, `rulers` and related topics.

6.1 Induction Examples

Recall the definition of `sumn`.

```
(definec sumn (n :nat) :nat
  (if (equal n 0)
      0
    (+ n (sumn (- n 1)))))
```

Let's prove

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

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

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

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). You have to identify both the function and the variables when using induction schemes! We are using the induction scheme of `(nind n)`.

Our proof obligations are then:

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

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.

To see one reason why we need to identify variables, suppose we were asked to prove the following equivalent conjecture:

$$\texttt{(natp x)} \Rightarrow \texttt{(sumn x)} = \mathtt{x}(\mathtt{x}+1)/2$$

The induction scheme to use is now `(nind x)`, which is the induction scheme we would get if `nind` was defined using `x` instead of `n`.

Let's now reason about the following function definition.

```
(definec aapp (a :tl b :tl) :tl
  (if (endp a)
      b
    (cons (first a) (aapp (rest a) b))))
```

We want to prove that `aapp` is associative.

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

Contract checking gives:

$$\texttt{(tlp a)} \wedge \texttt{(tlp b)} \wedge \texttt{(tlp c)} \Rightarrow \texttt{(aapp (aapp a b) c)} = \texttt{(aapp a (aapp b c))} \quad (6.2)$$

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

Because we don't know how many times to expand `aapp`. 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 `tlp`'s induction scheme. Recall the definition of `tlp`.

```
(definec tlp (l :all) :bool
  (if (consp l)
      (tlp (cdr l))
    (equal l ())))
```

This brings up another reason why we need to identify variables in induction schemes. In (6.2), we have multiple variables that are true-lists, so which one are we using to generate an induction scheme? Let's say we use `a`, then the induction scheme of `(tlp a)` is:

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

This is equivalent to:

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

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

Exercise 6.5 *We saw that the variables we use in proofs are irrelevant,* e.g.*, even though* `tlp` *was defined over* `l`*, we can apply induction using* `a` *instead. Explain why this is the case.*

Exercise 6.6 *Assume that the output type for* `aapp` *was defined to be* `all`*. This version of* `aapp` *is admissible. Using this version of* `aapp`*, use induction to prove* `(tlp (aapp x y))`*. (You have to perform contract completion first.)*

Exercise 6.7 *Prove the following conjecture.*

```
(implies (and (tlp x)
              (tlp y))
         (equal (llen (aapp x y))
                (+ (llen x) (llen y))))
```

Exercise 6.8 *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 6.9 *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 `rrev`. Here is the definition.

```
(definec rrev (x :tl) :tl
  (if (endp x)
      nil
    (aapp (rrev (rest x)) (list (first x)))))
```

Now we want to prove:

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

Contract checking gives:

$$\texttt{(tlp x)} \Rightarrow \texttt{(rrev (rrev x))} = \texttt{x} \tag{6.3}$$

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

Because we don't know how many times to expand `rrev`. 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 `tlp`'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:

$$\texttt{(tlp x)} \wedge \texttt{(tlp y)} \Rightarrow \texttt{(rrev (aapp x y))} = \texttt{(aapp (rrev y) (rrev x))} \tag{6.4}$$

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 6.10 *Prove* (6.4)*. Use the induction scheme of* `(tlp x)`*.*

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

$$\texttt{(tlp x)} \Rightarrow \texttt{(aapp x nil)} = \texttt{x}$$

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

Exercise 6.11 *What induction scheme does this give rise to?*

```
(definec fib (n :pos) :pos
  (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 6.12 *Prove the previous conjecture using the induction scheme of* `fib`*.*

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.*, `tlp` 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 `aapp`, we get to assume that `aapp` works correctly in the recursive case, even if its first input has 1,000,000 elements, *i.e.*, we get to assume that `aapp` 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. Nonrecursive 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 6.13 *Consider the following non-terminating function definition.*

```
(definec f (x :all) :all
  (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 Induction Like a Professional

In exercise 6.10, we asked you to prove Conjecture 6.4.

$$\texttt{(tlp x)} \wedge \texttt{(tlp y)} \Rightarrow \texttt{(rrev (aapp x y))} = \texttt{(aapp (rrev y) (rrev x))} \tag{6.5}$$

We told you what induction scheme to use (the one that `(tlp x)` gives rise to).

Typically, you will have to figure out what induction scheme to use. How do you go about doing that?

Look at the left-hand-side and of the equality in the conclusion. What variables control the recursive functions? On the LHS, `x`, due to `(aapp x y)` and and on the right hand side it is both `x` and `y`, so `x` is a better choice.

Here is how I think about it. I can assume that the theorem I am trying to prove holds for "smaller" values of the arguments, so let me just not worry about exactly what smaller values to use and after I do the proof, find the induction scheme that I needed! Also, I will just sketch out the proof, without worrying about full justifications, which I will add later.

This is how the professions think about it and it is a more powerful way of thinking about induction.

So, here we go.

```
(rrev (aapp x y))
= { Def aapp }
(rrev (cons (first x) (aapp (rest x) y)))
= { Def rrev }
(aapp (rrev (aapp (rest x) y)) (list (first x)))
= { Using an instance of the theorem ((x (rest x))) }
(aapp (aapp (rrev y) (rrev (rest x))) (list (first x)))
= { Lemma assoc-append }
(aapp (rrev y) (aapp (rrev (rest x)) (list (first x))))
= { Def rrev }
(aapp (rrev y) (rrev x))
```

Bingo! Induct on `(tlp x)` because we need an induction scheme that allows us to assume that the theorem holds when we replace `x` with `(rest x)`.

We still have to do the base case. That should be easy, right. If so, we often just say it is "obvious" using equational reasoning. But don't do that unless you are sure.

So, let us try us sketch out the base case.

```
C1. (tlp x)
C2. (tlp y)
C3. (not (consp x))
-------------------
D1. x=nil

(rrev (aapp x y))
= { Def aapp, D1 }
(rrev y)
= { ??? }
(aapp (rrev y) nil)
= { Def rrev, D1 }
(aapp (rrev y) (rrev x))
```

But how do I justify the `???` step?

I can't just say { `Def aapp` } because `aapp` recurs down its first argument, so we have to prove

```
(aapp x nil) = x
```

Exercise 6.14 *Try proving the following using conjecture using induction.*

```
(implies (tlp x) (equal (aapp x nil) x))
```

A. Try proving it with the induction scheme of `aapp`.
B. Try proving it like a professional.
C. Try proving it with the induction scheme of `tlp`.

6.5 Generalization

Consider the following definitions.

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))

(definec subsetp (x :tl y :tl) :tl
  ; checks if every element in x is in y
  (or (endp x)
      (and (in (first x) y)
           (subsetp (rest x) y))))
```

Try to prove the following theorem:

$$\texttt{(tlp x)} \Rightarrow \texttt{(subsetp x x)} \tag{6.6}$$

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:

$$\texttt{(tlp x)} \wedge \texttt{(tlp y)} \wedge \texttt{(subsetp x y)} \Rightarrow \texttt{(subsetp x (cons a y))} \tag{6.7}$$

Now, we can prove (6.7) using induction.

Exercise 6.15 *Prove* (6.7)

Exercise 6.16 *Prove* (6.6)

Exercise 6.17 *Prove* `(tlp x)` $\wedge$ `(tlp y)` $\wedge$ `(tlp z)` $\wedge$ `(subsetp x y)` $\wedge$ `(subsetp y z)` $\Rightarrow$ `(subsetp x z)`

6.6 Reasoning About Accumulator-Based Functions

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

```
(definec rrev (x :tl) :tl
  (if (endp x)
      nil
    (aapp (rrev (rest x)) (list (first x)))))
```

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

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

In the recursive case of `rrev`, we have `aapp` applied to `(rrev (rest x))` which requires `(len (rrev (rest x)))` conses plus one cons for the list, *i.e.*, `(len x)` conses. Since `rrev` 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.

```
(definec revt (x :tl acc :tl) :tl
  (if (endp x)
      acc
    (revt (rest x) (cons (first x) acc))))
```

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

```
(definec rev* (x :tl) :tl
  (revt x nil))
```

Exercise 6.18 *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 `rrev` 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 `rrev`?

Let's prove that they are equal.

$$\texttt{(tlp x)} \Rightarrow \texttt{(rev* x)} = \texttt{(rrev x)} \tag{6.8}$$

Is it true?

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

Notice that proving (6.8) will require proving:

$$\texttt{(tlp x)} \Rightarrow \texttt{(revt x nil)} = \texttt{(rrev x)} \tag{6.9}$$

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 (6.9) using the induction scheme of `(tlp x)`.

1. ¬`(consp x)` ⇒ (6.9)
2. `(consp x)` ∧ (6.9)$|_{\texttt{((x (rest x)))}}$ ⇒ (6.9)

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. `(tlp x)`

C3. `(tlp (rest x))` ⇒ `(revt (rest x) nil)` = `(rrev (rest x))`

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

C5. `(revt (rest x) nil)` = `(rrev (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.

`(tlp x)` ∧ `(tlp 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:

$$\texttt{(tlp x)} \wedge \texttt{(tlp acc)} \Rightarrow \texttt{(revt x acc)} = \texttt{(aapp (rrev x) acc)} \tag{6.10}$$

Suppose we prove (6.10). Can we use it to prove (6.8) and (6.9)?
Yes. Why?
The base case is simple. Here is a proof of the induction step. First, our context is:

C1. `(consp x)`

C2. `(tlp x)`

C3. `(tlp acc)`

C4. `(tlp (rest x))` ∧ `(tlp acc)` ⇒
`(revt (rest x) acc)` = `(aapp (rrev (rest x)) acc)`

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

C6. `(revt (rest x) acc)` =
`(aapp (rrev (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

$$\texttt{(revt (rest x) acc)} = \texttt{(aapp (rrev (rest x)) acc)}$$

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 of `(revt x acc)`. That gives us the following context:

C1. `(consp x)`

C2. `(tlp x)`

C3. `(tlp acc)`

C4. `(tlp (rest x))` ∧ `(tlp (cons (first x) acc))` ⇒
`(revt (rest x) (cons (first x) acc))` =
`(aapp (rrev (rest x)) (cons (first x) acc))`

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

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

C7. `(revt (rest x) (cons (first x) acc))` =
`(aapp (rrev (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 }
```
    (aapp (rrev (rest x)) (cons (first x) acc))
```
= { Def of `aapp` (working on pulling `acc` out to match RHS) }
```
    (aapp (rrev (rest x)) (aapp (list (first x)) acc))
```
= { Associativity of `aapp` (now we pull `acc` out) }
```
    (aapp (aapp (rrev (rest x)) (list (first x))) acc)
```
= { Def of `rrev`, C1 }
```
    (aapp (rrev 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 ⇒ `(f* ...)` = `(f ...)`

Part 2: Proving theorems

4. Identify a lemma that relates `ft` to `f`. It should have the following form:

   hyps ⇒ `(ft ... acc)` = ... `(f ...)` ...

   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 ⇒ `(f* ...)` = `(f ...)`

   If you have to prove lemmas, prove them later.

6. Prove the lemma in 4. Use the induction scheme of `ft`.
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.

6.7 Exercises

6.7.1 List Theory

The following functions are used in the exercises below.

```
(definec rem-dups (a :tl) :tl
  (cond ((endp a) nil)
        ((in (car a) (cdr a))
         (rem-dups (cdr a)))
        (t (cons (car a) (rem-dups (cdr a))))))
```

```
(definec no-dups (a :tl) :bool
  (or (endp a)
      (and (not (in (car a) (cdr a)))
           (no-dups (cdr a)))))
```

Exercise 6.19 *Prove the following.*

```
;theorem llen-rrev
(implies (and (tlp x)
              (tlp y))
         (equal (llen (aapp x y))
                (+ (llen x) (llen x))))
```

Exercise 6.20 *Prove the following.*

```
;theorem llen-rrev
(implies (tlp x)
         (equal (llen (rrev x))
                (llen x)))
```

Exercise 6.21 *Prove the following.*

```
;theorem llen-rem-dups
(implies (tlp x)
         (<= (llen (rem-dups x))
             (llen x)))
```

Exercise 6.22 *Prove the following.*

```
;theorem llen-rem-dups-no-dups
(implies (and (tlp x)
              (no-dups x))
         (equal (llen (rem-dups x))
                (llen x)))
```

Exercise 6.23 *Prove the following.*

```
;theorem in-aapp
(implies (and (tlp x)
              (tlp y))
         (equal (in a (aapp x y))
                (or (in a x) (in a y))))
```

Exercise 6.24 *Prove the following.*

```
;theorem in-rrev
(implies (tlp x)
         (equal (in a (rrev x))
                (in a x)))
```

Exercise 6.25 *Prove the following.*

```
;theorem in-rem-dups
(implies (tlp x)
         (equal (in a (rem-dups x))
                (in a x)))
```

Exercise 6.26 *Prove the following.*

```
;theorem rem-dups-no-dups
(implies (and (tlp x)
              (no-dups x))
         (equal (rem-dups x)
                x))
```

Exercise 6.27 *Prove the following.*

```
;theorem rem-dups-has-no-dups
(implies (tlp x)
         (no-dups (rem-dups x)))
```

Exercise 6.28 *Prove the following.*

```
;theorem rem-dups-idempotent
(implies (tlp x)
         (equal (rem-dups (rem-dups x))
                (rem-dups x)))
```

6.7.2 Set Theory

The following functions are used in the exercises below.

```
(definec == (x :tl y :tl) :bool
  ; checks if x and y have exactly the same set of elements
  (and (subsetp x y)
       (subsetp y x)))

(definec union (x :tl y :tl) :tl
; the union of x and y
  (aapp x y))

(definec intersect (x :tl y :tl) :tl
  ; the intersection of x and y
  (cond ((endp x)  nil)
        ((in (car x) y)
         (cons (car x) (intersect (cdr x) y)))
        (t (intersect (cdr x) y))))

(definec cardinality (x :tl) :nat
; the number of distinct elements in x
  (llen (rem-dups x)))
```

For all the exercises in this section, either prove the conjecture or exhibit a counterexample. You may find it useful to do the exercises in an order that is different than the order in which they are presented.

Exercise 6.29 *Prove the following.*

```
;theorem intersect-same
(implies (tlp x)
         (equal (intersect x x)
                x))
```

Exercise 6.30 *Prove the following.*

```
;theorem intersect-x-sub-y
(implies (and (tlp x)
              (tlp y)
              (subsetp x y))
         (equal (intersect x y)
                x))
```

Exercise 6.31 *Prove the following.*

```
;theorem intersect-llen
(implies (and (tlp x)
              (tlp y)
              (tlp z)
              (subsetp x y)
              (subsetp y z))
         (equal (llen (intersect x (intersect y z)))
                (llen x)))
```

Exercise 6.32 *Prove the following.*

```
;theorem in-intersect
(implies (and (tlp x)
              (tlp y))
         (equal (in a (intersect x y))
                (and (in a x) (in a y))))
```

Exercise 6.33 *Prove the following.*

```
;theorem intersect-associative
(implies (and (tlp x)
              (tlp y)
              (tlp z))
         (equal (intersect (intersect x y) z)
                (intersect x (intersect y z))))
```

Exercise 6.34 *Prove the following.*

```
;theorem intersect-llen-lemma
(implies (and (tlp x)
              (tlp y)
              (tlp z)
              (tlp w))
         (equal (llen (intersect (intersect x y) (intersect w z)))
                (llen (intersect x (intersect (intersect y w) z)))))
```

Exercise 6.35 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (subsetp (rrev x) y))
         (subsetp x (rrev y)))
```

Exercise 6.36 *Prove the following.*

```
(implies (tlp x)
         (== (union x x) x))
```

Exercise 6.37 *Prove the following.*

```
(implies (tlp x)
         (== (intersect x x) x))
```

Exercise 6.38 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (== (union x y)
             (union y x)))
```

Exercise 6.39 *Prove the following.*

```
(implies (tlp x)
         (equal (intersect x x) x))
```

Exercise 6.40 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (<= (cardinality (union x y))
             (+ (cardinality x) (cardinality y))))
```

Exercise 6.41 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (tlp z))
         (equal (intersect (intersect x y) z)
                (intersect x (intersect y z))))
```

Exercise 6.42 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (tlp z))
         (equal (union (union x y) z)
                (union x (union y z))))
```

Exercise 6.43 *Prove the following.*

```
(implies (tlp x)
         (<= (cardinality x)
             (llen x)))
```

Exercise 6.44 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (equal (cardinality (union x x))
                (cardinality x)))
```

Exercise 6.45 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (equal (cardinality (intersect x x))
                (cardinality x)))
```

Exercise 6.46 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y))
         (<= (cardinality (union x y))
             (+ (cardinality x) (cardinality y))))
```

Exercise 6.47 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (equal (cardinality (union x y)) (cardinality y)))
         (equal (cardinality (union y x))
                (cardinality y)))
```

Exercise 6.48 *Prove the following.*

```
(implies (and (tlp x)
              (equal (cardinality x) (llen x)))
         (equal (rem-dups x) x))
```

Exercise 6.49 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (subsetp x y)
              (<= n (cardinality x)))
         (<= n (llen y)))
```

Exercise 6.50 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (< 0 (cardinality x))
              (== x y))
         (consp y))
```

Exercise 6.51 *Prove the following.*

```
(implies (and (tlp x)
              (tlp y)
              (subsetp (union x y) x)
              (< (cardinality x) (cardinality y)))
         (not (equal x x)))
```

6.7.3 Sorting

In this section, the exercises require you define functions and formalize claims written in (rigorous) English.

Exercise 6.52 *Use* `defdata` *to define* `lor`*, a (true) list of rationals.*

Exercise 6.53 *Define insertion sort, where the input is a* `lor`*.*

Exercise 6.54 *Define quicksort, where the input is a* `lor`*.*

Exercise 6.55 *Define merge sort, where the input is a* `lor`*.*

Exercise 6.56 *Define bubble sort, where the input is a* `lor`*.*

Exercise 6.57 *Define* `orderedp`*, a function that given an* `lor` *checks that the elements of the list are in non-decreasing order.*

Exercise 6.58 *Define* `permp`*, a function that given a two true-lists (not necessarily* `lor`*s) checks that they are permutations of each other.*

Exercise 6.59 *Show that insertion sort returns an ordered permutation.*

Exercise 6.60 *Show that quicksort returns an ordered permutation.*

Exercise 6.61 *Show that merge sort returns an ordered permutation.*

Exercise 6.62 *Show that bubble sort returns an ordered permutation.*

Exercise 6.63 *Show that if* `x` *and* `y` *are permutations of each other and they are both ordered, then they are equal.*

Exercise 6.64 *Show that all the sorting algorithm are equal to each other.*

Exercise 6.65 *Without using the definition of* `permp` *(or any theorems that depend on* `permp`*) show that insertion sort is equal to quicksort.*