Reasoning About Programs

Panagiotis Manolios
Northeastern University

February 25, 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!

4

Equational Reasoning

We just finished studying propositional logic, so let's start by considering the following question:

Why do we need more than propositional logic?

After all, we were able to do a lot with propositional logic, including declarative design, security and digital logic.

What we are really after, however, is reasoning about programs, and while propositional logic will play an important role, we need more powerful logics.

To see why, let's simplify things for a moment and consider conjectures involving numbers and arithmetic operations.

Consider the conjecture:

Conjecture 1 $a + b = ba$

What does it mean for this conjecture to be true or false?

Well, there is a source of ambiguity here. If a, b are constants (like 1, 2, etc.) then we can just evaluate the equality and determine if it is true or not.

However, a and b are variables. This is similar to the propositional formulas we saw, *e.g.*,

$$p \wedge q \;\Rightarrow\; p \vee q$$

Recall that p and q are atoms, and the above formula is valid. What that means is that no matter what value p and q have, the above formula is true. Another way of saying this is that the above formula evaluates to true under all assignments.

In Conjecture 1, a and b range over a different domain than the Booleans, let's say they range over the rationals.

So, what we really mean when we say that conjecture 1 is valid (or true) is that for any rational number a and any rational number b, $a + b = ba$. Another way of saying this is that the above formula evaluates to true under all assignments. Notice the similarity with the Boolean case.

Is Conjecture 1 a valid formula?

No. We can come up with a counterexample.

Is Conjecture 1 unsatisfiable?

No. It is both satisfiable and falsifiable. Again, this is exactly the kind of characterization we used to classify Boolean formulas. How many assignments falsify the conjecture? How many assignments satisfy the conjecture?

What about the following conjecture?

Conjecture 2 $a + b = b + a$

Can we come up with a counterexample?

No.

Is the formula valid?

Yes.

How do we prove that a conjecture is valid in the case of propositional logic? We use a truth table, with a row per possible assignment, to show that no counterexample exists. A counterexample is an assignment that evaluates to false.

Can we do something similar here?

Yes, but the number of assignments is unfortunately infinite. That means, we can never completely fill in a table of assignments.

We need a radically new idea here.

We want something that allows us to do a finite amount of work and from that to deduce that there are no counterexamples in the infinite table, were we even able to construct it.

Let's look at how we might do this, but in the context of programs. First we start with the definition of `alen`, a length function whose domain is `All`.

```
(defunc alen (l)
  :input-contract t
  :output-contract (natp (alen l))
  (if (atom l)
      0
    (+ 1 (alen (rest l)))))
```

Consider the following conjecture:

Conjecture 3 `(equal (alen (list x)) (alen x))`

Is this conjecture true (valid) or false (falsifiable)?

What does it mean for Conjecture 3 to be true? That no matter what object of the ACL2s universe `x` is, the above equality holds.

Conjecture 3 is false. Why?

Suppose that `x` = `1`, then the conjecture evaluates to `nil`, *i.e.*,

$$[\![\texttt{(equal (alen (list 1)) (alen 1))}]\!] = [\![\texttt{(equal 1 0)}]\!] = \texttt{nil}$$

So, finding a counterexample is "easy." All we have to do is to find an assignment under which the conjecture evaluates to `nil`. This is just like the Boolean logic case.

Is Conjecture 3 unsatisfiable? No. Again, all we have to do is find one satisfying assignment, *e.g.*, `x` = `(1)`. How many assignments falsify the conjecture? How many assignments satisfy the conjecture?

What about:

Conjecture 4

```
(equal (alen (cons x (list z)))
       (alen (cons y (list z))))
```

Here we can't find a counterexample.

How can we go about proving that Conjecture 4 is valid?

```
    (alen (cons x (list z)))
```
= { Definition of `alen`, instantiation }
```
    (if (atom (cons x (list z))) 0 (+ 1 (alen (rest (cons x (list z))))))
```
= { `first-rest` axioms }
```
    (if (atom (cons x (list z))) 0 (+ 1 (alen (list z))))
```
= { Definition of `atom` }
```
    (if (not (consp (cons x (list z)))) 0 (+ 1 (alen (list z))))
```
= { Definition of `not` }
```
    (if (if (consp (cons x (list z))) nil t) 0 (+ 1 (alen (list z))))
```
= { `consp` axioms }
```
    (if (if t nil t) 0 (+ 1 (alen (list z))))
```
= { `if` axioms }
```
    (if nil 0 (+ 1 (alen (list z))))
```
= { `if` axioms }
```
    (+ 1 (alen (list z)))
```

What we have shown so far is:

$$\texttt{(alen (cons x (list z)))} = \texttt{(+ 1 (alen (list z)))} \tag{4.1}$$

which we will feel free to write as

$$\texttt{(alen (cons x (list z)))} = 1 + \texttt{(alen (list z))} \tag{4.2}$$

because it should be clear how to go from (4.1) to (4.2) and because we have been trained to use infix for arithmetic operators since elementary school.

You should be able to continue the proof to show that

$$\texttt{(alen (cons x (list z)))} = 2 \tag{4.3}$$

Finish the proof.

Once the proof is done, we have shown that (4.3) is valid. Any validity that we establish via proof is called a *theorem*, so (4.3) is a theorem. To reason about built-in functions such as `consp`, `if`, and `equal` we use *axioms*, which you can think of as built-in theorems providing the semantics of the built-in functions. Every time we define a function that ACL2s admits, we also get a *definitional axiom*, which, for now, you can think of as an axiom stating that the function is equal to its body (but more on this soon). We can then reason from these basic axioms (which are also theorems) using what is called *first order logic*. First order logic includes propositional reasoning, but extends it significantly. We will introduce as much of first order reasoning as needed.

Back to our proof. We are not done with the proof of Conjecture 4, but there are at least two reasonable ways to proceed.

First, we might say:

If we simplify the RHS (Right Hand Side), we get

```
(alen (cons y (list z)))
```

= { Definition of `len`, instantiation }

...

= { `if` axioms }

`(+ 1 (alen (list z)))`

= { Definition of `len`, instantiation }

...

= { Arithmetic }

2

So, the LHS (Left Hand Side) and RHS are equal.

What we realized is that the same steps that we used to simplify the LHS can be used in a symmetric way to simplify the RHS. In this class we will avoid proofs involving "...". Here's a better way to make the argument:

First, note that (4.3) is a theorem. By instantiating (4.3) with the substitution `((x y))`, we get:

$$\texttt{(alen (cons y (list z)))} = 2 \tag{4.4}$$

Putting (4.3) and (4.4) together, we have

$$\texttt{(alen (cons x (list z)))} = \texttt{(alen (cons y (list z)))}$$

So, Conjecture 4 is a theorem.

We already saw that instantiation can be used in propositional logic. Its use is indispensable when reasoning about ACL2s programs!

This example highlights the new tool we have that allows us to reason about programs: proof. The game we will be playing is to construct proofs of conjectures involving some of the basic functions we have already defined (*e.g.*, `len`, `app`, and `rev`). We will focus on these simple functions because their simplicity allows us to focus exclusively on how to prove theorems without the added complexity of having to understand what conjectures mean.

Once we prove that a conjecture is valid, we say that the conjecture is a *theorem*. We are then free to use that theorem in proving other theorems. This is similar to what happens when we program: we define functions and then we use them to define other functions (*e.g.*, we define `rev` using `app`).

What's new here?

Well, we are beyond the realm of the propositional. We have variables ranging over the ACL2s universe, equality, and functions.

Let's look at equality. To simplify notation, we tend to write expressions involving `equal` using = instead. This is similar to what we did with arithmetic. For example, instead of writing the more technically correct

$$\texttt{(equal (alen (cons x z)) (alen (cons y z)))}$$

we usually write the more familiar

$$\texttt{(alen (cons x z))} = \texttt{(alen (cons y z))}$$

We feel free to go back and forth without justification.

If we want to be pedantic, here is how `equal` and = are related.

- $x = y \;\Rightarrow\; \texttt{(equal x y)} = \texttt{t}$
- $x \neq y \;\Rightarrow\; \texttt{(equal x y)} = \texttt{nil}$

When we use $=$ or $\neq$ in expressions, they bind more tightly than any of the propositional connectives.

How can we reason about equality? We will use just two properties of equality. First, equality is what is called an *equivalence relation*, *i.e.*, it satisfies the following properties.

- *Reflexivity*: $x = x$
- *Symmetry*: $x = y \;\Rightarrow\; y = x$
- *Transitivity*: $x = y \wedge y = z \;\Rightarrow\; x = z$

That $=$ is an equivalence relation is what allows us to chain together the sequence of equalities in the proof of Conjecture 4 above to conclude that `(alen (cons x (list z)))` $= 2$.

The second property of equality we will use is the *Equality Axiom Schema for Functions*: For every function symbol `f` of arity n we have the axiom

$$(x_1 = y_1 \wedge \cdots \wedge x_n = y_n) \;\Rightarrow\; \texttt{(f } x_1 \cdots x_n\texttt{)} = \texttt{(f } y_1 \cdots y_n\texttt{)}$$

To reason about constants, we can use evaluation, *e.g.*, all of the following are theorems.

$$\texttt{t} \neq \texttt{nil}$$

$$\texttt{()} = \texttt{nil}$$

$$1 \neq 2$$

$$2/4 = 4/8$$

$$\texttt{(cons 1 ())} = \texttt{(list 1)}$$

To reason about built-in functions, such as `cons`, `first`, and `rest`, we have axioms for each of these functions that are derived from their semantics.

$$\texttt{(first (cons x y))} = \texttt{x}$$

$$\texttt{(rest (cons x y))} = \texttt{y}$$

$$\texttt{(consp (cons x y))} = \texttt{t}$$

$$x = \texttt{nil} \Rightarrow \texttt{(if x y z)} = \texttt{z}$$

$$x \neq \texttt{nil} \Rightarrow \texttt{(if x y z)} = \texttt{y}$$

What about instantiation? It is a rule of inference:
Instantiation: Derive $\varphi|_\sigma$ from φ. That is, if φ is a theorem and σ is a substitution, then by instantiation, $\varphi|_\sigma$ is a theorem.

For example, since this is a theorem `(equal (first (cons x y)) x)` we can derive `(equal (first (cons (foo x) (bar z))) (foo x))`.

More carefully, a substitution is just a list of the form:

$$\texttt{((}var_1\ term_1\texttt{)}\ \ldots\ \texttt{(}var_n\ term_n\texttt{))}$$

where the var_i are *target variables* and the $term_i$ are their *images*. We require that the var_i are distinct. The application of this substitution to a formula uniformly replaces every free occurrence of a target variable by its image.

What does it mean to say that the following is a theorem?

```
(alen (cons x (list z))) = (alen (cons y (list z)))
```

That no matter what you replace `x`, `y`, and `z` with from the ACL2s universe, the LHS and RHS evaluate to the same thing.

Let's try to prove another conjecture.

Conjecture 5 `(app (cons x y) z) = (cons x (app y z))`

```
    (app (cons x y) z)
= { Definition of app, instantiation }
    (if (endp (cons x y)) z (cons (first (cons x y)) (app (rest (cons x y)) z)))
= { Definition of endp, axioms for consp }
    (if nil z (cons (first (cons x y)) (app (rest (cons x y)) z)))
= { Axioms for if }
    (cons (first (cons x y)) (app (rest (cons x y)) z))
= { Axioms for first, rest }
    (cons x (app y z))
```

Unfortunately, the above "proof" has a problem. Unlike `alen`, which is defined for the whole ACL2s universe, `app` is only defined for lists.

Recall the definitions:

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

(defunc endp (l)
  :input-contract (listp l)
  :output-contract (booleanp  (endp l))
  (not (consp l)))

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

The definition of functions such as `app` give rise to *definitional axioms*. Here is the definitional axiom that `app` gives rise to:

`(listp a)` $\wedge$ `(listp b)`

```
⇒
(app a b)
=
(if (endp a)
    b
  (cons (first a) (app (rest a) b)))
```

In general, every time we successfully admit a function, we get two axioms of the form

$$ic \ \Rightarrow \ (\texttt{f} \ x_1 ... x_n) = body$$

$$ic \ \Rightarrow \ oc$$

where *ic* is the input contract for `f`, and where *oc* is the output contract for `f`. We will be very precise about what "successfully admit" means, but, for now, take this to mean that ACL2s accepts your function definition. Recall that this involves proving termination, proving the function contracts, and proving the body contracts.

So, we can't expand the definition of `app` in the proof of Conjecture 5, unless we know:

$$\texttt{(listp (cons x y))} \ \wedge \ \texttt{(listp z)}$$

which is equivalent to:

$$\texttt{(listp y)} \ \wedge \ \texttt{(listp z)}$$

So, what we really proved was:

Theorem 4.1 $\texttt{(listp y)} \wedge \texttt{(listp z)} \Rightarrow \texttt{(app (cons x y) z)} = \texttt{(cons x (app y z))}$

When we write out proofs, we will not explicitly mention input contracts when using a function definition because the understanding is that every time we use a definitional axiom to expand a function, we have to check that we satisfy the input contract, so we don't need to remind the reader of our proof that we did something we all understand always needs doing.

It is often the case that when we think about conjectures that we expect to be valid, we often forget to carefully specify the hypotheses under which they are valid. These hypotheses depend on the input contracts of the functions mentioned in the conjectures, so get into the habit of looking at conjectures and making sure that they have the needed hypotheses. *Contract checking* is the process of checking that a conjecture has all the hypotheses required by the contracts of the functions appearing in the conjecture. *Contract completion* is the process of adding the missing hypotheses (if any) identified during contract checking. Contract checking and completion is similar to what you do when you write functions: you check the body contracts of the functions you define and if you are calling the functions on arguments of the wrong type, then you modify your code appropriately. In the case of function definitions, as we have seen, it is often the case that if the function definition is wrong, there is also a contract violation. Similarly, if a conjecture is not valid, it is often the case that there is a contract violation.

Let's look at another example:

Conjecture 6 $\texttt{(endp x)} \ \Rightarrow \ \texttt{(app (app x y) z)} = \texttt{(app x (app y z))}$

Can I prove this? Check the contracts of the conjecture.

Contract checking and completion gives rise to:

Conjecture 7 `(listp x)` $\wedge$ `(listp y)` $\wedge$ `(listp z)` $\wedge$ `(endp x)` $\Rightarrow$ `(app (app x y) z)` = `(app x (app y z))`

By the way, notice all of the hypotheses. Notice the Boolean structure. This is why we studied Boolean logic first! Almost everything we will prove will include an implication.

Notice that in ACL2s, we would technically write:

```
(implies (and (listp x)
              (listp y)
              (listp z)
              (endp x))
         (equal (app (app x y) z)
                (app x (app y z))))
```

The first thing to do when proving theorems is to take the Boolean structure into account by writing the conjecture in the form:

$$hyp_1 \wedge hyp_2 \wedge \cdots \wedge hyp_n \;\Rightarrow\; conc$$

where we have as many *hyp*s as possible. We will call the set of top-level hypotheses (*i.e.*, $\{hyp_1, hyp_2, \ldots, hyp_n\}$) our *context*.

Our context for Conjecture 7 is:

C1. `(listp x)`

C2. `(listp y)`

C3. `(listp z)`

C4. `(endp x)`

We then look at our context and see what obvious things our context implies. The obvious thing here is that C1 and C4 imply that `x` must be `nil`, so we add to our context the following:

C5. `x` = `nil` { C1, C4 }

Notice that any new facts we add must come with a justification. We will use the convention that all elements of our context will be given a label of the form Ci, where i is a positive integer.

The next thing we do is to start with the LHS of the conclusion and to try and reduce it to the RHS, using our proof format. If we need to refer to the context in one of proof step justifications, say Context 5, we write C5.

```
  (app (app x y) z)
= { Def of app, C5, Def of endp, if axioms }
  (app y z)
= { Def of app, C5, Def of endp, if axioms }
  (app x (app y z))
```

Notice that we took bigger steps than before. Before we might have written:

```
    (app (app x y) z)
= {  Def of app  }
    (app (if (endp x) y (cons (first x) (app (rest x) y))) z)
= {  C5  }
    (app (if (endp nil) y (cons (first nil) (app (rest nil) y))) z)
= {  Def of endp  }
    (app (if t y (cons (first nil) (app (rest nil) y))) z)
= {  If axioms  }
    (app y z)
...
```

So, the above four steps were compressed into one step. Why? Because many of the steps we take involve expanding the definition of a function. Function definitions tend to have a top-level `if` or `cond` and as a general rule we will not expand the definition of such a function unless we can determine which case of the top-level `if`-structure will be true. If we just blindly expand function definitions, we'll wind up with a sequence of increasingly complicated terms that don't get us anywhere. So, if we know which case of the top-level `if` is true, then why go to the trouble of writing out the whole body of the function? Why not just write out that one case? Well, that's why we allow ourselves to expand definitions as in the first proof of Conjecture 7.

One other comment about the first proof of Conjecture 7. Students often have no difficulty with the first step, but have difficulty with the second step. The second step requires one to see that the simple term:

```
(app y z)
```

can be transformed into the RHS

```
(app x (app y z))
```

This may seem like a strange thing to do because students are used to thinking about computation as unfolding over time. So, if `x` is `nil` then of course the following holds.

```
    (app (app x y) z)
= {  Def of app, ...  }
    (app y z)
```

Because when we compute `(app x y)` we get `y`.

What students initially have difficulty with is seeing that you can reverse the flow of time and everything still works. For example the following is true,

```
    (app y z)
= {  Def of app, ...  }
    (app (app x y) z)
```

Because starting with `(app y z)` we can run time in reverse to get `(app x (app y z))` (recall `x` is `nil`). In fact, this is "obvious" from the equality (=) axioms that tell us that equality is an equivalence relation (reflexive, symmetric, and transitive). The symmetry

axiom tells us that we can view computation as moving forward in time or backward. It just doesn't make a difference.

As an aside, it turns out that in physics, we can't reverse time and so this symmetry we have with computation is not a symmetry we have in our universe. One reason why we can't reverse time in physics is that the second law of thermodynamics precludes it. The second law of thermodynamics implies that entropy increases over time. There is an even more fundamental reason why time is not reversible. This second reason has to do with the fundamental laws of physics at the quantum level, whereas the second law of thermodynamics is thought to be a result of the initial conditions of our universe. The second reason is that in our current understanding of the universe, there are very small violations of time reversibility exhibited by subatomic particles. The extent of the violations is not fully understood and probably has something to do with the imbalance of matter and antimatter in the visible universe. There is almost no antimatter in the visible universe and one of the big open problems in physics is trying to understand why that is the case.

4.1 Testing Conjectures

Recall that since Conjecture 7 is a theorem, whatever we replace the free variables with, the conjecture will evaluate to `t`. A convenient way of checking the conjecture using ACL2s is to use `let`, as follows:

```
(let ((x nil)
      (y nil)
      (z nil))
  (implies (and (endp x)
                (listp x)
                (listp y)
                (listp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

An even more convenient method is to use ACL2s to test the conjecture. Here is how:

```
(test?
  (implies (and (endp x)
                (listp x)
                (listp y)
                (listp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

There are three possible outcomes.

1. ACL2s proves that the conjecture is a theorem.
2. ACL2s finds a counterexample, *i.e.*, the conjecture is falsifiable.
3. None of the above hold, *i.e.*, the conjecture satisfies all of the tests ACL2s tries.

Consider another example. Consider the claim that app2, below, is equivalent to app.

```
(defunc app2 (x y)
  :input-contract (and (listp x) (listp y))
  :output-contract (listp (app2 x y))
  (if (endp y)
      x
    (cons (first y) (app2 x (rest y)))))
```

The claim is false, but app2 works fine on many tests, *e.g.*,

```
(check= (app2 '(1 2) '(1 2)) '(1 2 1 2))
(check= (app2 nil nil) nil)
(check= (app2 nil '(1 2 3)) '(1 2 3))
(check= (app2 '(1 2 3) nil) '(1 2 3))
```

Here is how we can use ACL2s to test the conjecture that app2 is equivalent to our definition.

```
(test?
  (implies (and (listp x) (listp y))
           (equal (app2 x y)
                  (app x y))))
```

ACL2s gives us counterexamples. It also shows us cases in which the conjecture is true.

Now, suppose that the specification for app2 only stated that (app2 x y) must return a list that contains all of the elements in x and all of the elements in y, where order doesn't matter, but repetitions do. The definition of app2 above satisfies the specification. In addition, there are different functions that satisfy the specification. How can we write tests that are independent of the implementation? We cannot write simple check=’s because there are exponentially many correct answers that app2 could return. We can't test that app2 is equal to our solution for the same reason. But, we can write conjectures that capture the specification and ACL2s can be used to test these conjectures.

Here is one way of doing this. We test that every element in (app2 x y) is also an element of (app x y) and conversely.

```
; check that if a is in app2, it is in app
(test?
  (implies (and (listp x)
                (listp y)
                (in a (app2 x y)))
           (in a (app x y))))

; check that if a is in app, it is in app2
(test?
  (implies (and (listp x)
                (listp y)
                (in a (app x y)))
           (in a (app2 x y))))
```

Exercise 4.1 *Unfortunately, we can define* `app2` *in a way that does not satisfy the specification, but does satisfy the above* `test?`*'s. Exhibit such a definition and check that it passes the above tests. A better solution is to test that* `app2` *is a permutation of* `app`*. Define a function that checks if its arguments are permutations of one another and use this to test both your faulty definition of* `app2` *and the definition given above.*

You can control how much testing ACL2s does. The default number of tests depends on the mode, but you can set it to whatever number you want, *e.g.*, here is how to instruct ACL2s to run 1,000 tests.

```
(acl2s-defaults :set num-trials 1000)
```

To summarize, ACL2s provides `test?`, a powerful facility for automatically testing programs. Instead of having to manually write tests, ACL2s generates as many tests as requested automatically. The other major advantage is that we do not have to specify exactly what functions have to do. In the `app2` example above, we did not have to say what `app2` returns; instead, we specified the properties we expect `app2` to satisfy. The advantage is that we *decouple* the testing of `app2` from the development of `app2`. In fact, even if we change the implementation of `app2`, the tests can remain the same.

4.2 Equational Reasoning with Complex Propositional Structure

Many of the conjectures we will examine have rich propositional structure. We now examine how to reason about such conjectures.

Conjecture 8

```
(consp x)
⇒
 [[(listp (rest x)) ∧ (listp y) ∧ (listp z)
   ⇒
   (app (app (rest x) y) z) = (app (rest x) (app y z))]
  ⇒
  [(listp x) ∧ (listp y) ∧ (listp z)
   ⇒
   (app (app x y) z) = (app x (app y z))]]
```

The above conjecture has the form

$$A \Rightarrow [B \Rightarrow C]$$

where

A is `(consp x)`

B is `[(listp (rest x)) ∧ (listp y) ∧ (listp z)`
`⇒ (app (app (rest x) y) z) = (app (rest x) (app y z))]`

C is `[(listp x) ∧ (listp y) ∧ (listp z)`
`⇒ (app (app x y) z) = (app x (app y z))]`

What we are doing here is identifying some of the propositional structure of Conjecture 8. Here's why. It turns out that

$$A \Rightarrow [B \Rightarrow C] \equiv [A \wedge B] \Rightarrow C \tag{4.5}$$

This propositional equality is one we will use over and over. We will use it to rewrite Conjecture 8 so that the context has as many conjunctions as possible. After applying (4.5) to Conjecture 8, we get:

```
[(consp x) ∧
 [(listp (rest x)) ∧ (listp y) ∧ (listp z)
  ⇒
  (app (app (rest x) y) z) = (app (rest x) (app y z))]
⇒
[(listp x) ∧ (listp y) ∧ (listp z)
   ⇒
   (app (app x y) z) = (app x (app y z))]]
```

Applying (4.5) again and rearranging conjuncts gives us:

Conjecture 9

```
[(consp x) ∧
 (listp x) ∧
 (listp y) ∧
 (listp z) ∧
 [(listp (rest x)) ∧ (listp y) ∧ (listp z)
  ⇒
  (app (app (rest x) y) z) = (app (rest x) (app y z))]]
⇒
  (app (app x y) z) = (app x (app y z))]]
```

Now, we can extract the context. Doing so gives us:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp y)`

C4. `(listp z)`

C5. `[(listp (rest x)) ∧ (listp y) ∧ (listp z)]` ⇒
`[(app (app (rest x) y) z) = (app (rest x) (app y z))]`

Notice that we *cannot* use (4.5) on C5 to add the hypotheses of C5 to our context. Why?

We will be confronted with implications in our context (like C5) over and over. Usually what we will need is the consequent of the implication, but we can only use the consequent if we can also establish the antecedent, so we will try to do that. Here's how:

C6. `(listp (rest x))` { C1, C2, Def `listp` }

C7. `(app (app (rest x) y) z)` = `(app (rest x) (app y z))` $\{C6, C3, C4, C5, MP\}$

So, notice what we did. First we added C6 to our context. How did we get C6? Well, we know `(listp x)` (C2) and `(consp x)` (C1) so if we use the definitional axiom of `listp`, we get C6: `(listp (rest x))`.

Now, we have extended our context to include the antecedent of C5, so by propositional logic (Modus Ponens, abbreviated MP), we get that the conclusion also holds, *i.e.*, C7.

Recall that Modus Ponens tells us that if the following two formulas hold

$$A \;\Rightarrow\; B$$

$$A$$

Then so does the formula

$$B$$

We are now ready to prove the theorem. We start with the LHS of the equality in the conclusion of Conjecture 9.

```
  (app (app x y) z)
```
= { Def of `app`, C1, C2, C3 }
```
  (app (cons (first x) (app (rest x) y)) z)
```
= { Theorem 4.1 }
```
  (cons (first x) (app (app (rest x) y) z))
```
= { C7 }
```
  (cons (first x) (app (rest x) (app y z)))
```
= { Def of `app`, C1, C2, C3, C4 }
```
  (app x (app y z))
```

4.3 The difference between theorems and context

It is very important to understand the difference between a formula that is a theorem and one that appears in a context. A formula that appears in a context cannot be instantiated. It can only be used as is, in the proof attempt for the conjecture from which it was extracted. This is a major difference. Our contexts will never include theorems we already know. Theorems we already know are independent of any conjecture we are trying to prove and therefore do not belong in a context. A context is always formula specific.

Here is an example that shows why instantiation of context formulas leads to unsoundness. Here is a "proof" of

$$\mathtt{x} = 1 \;\Rightarrow\; 0 = 1 \tag{4.6}$$

Context:

C1. $\mathtt{x} = 1$

Proof 0
= { Instantiate C1 with `((x 0))` }
1

So, now we have a "proof" of (4.6), but using (4.6) we can get:

$$\texttt{nil} \tag{4.7}$$

How?
Instantiate (4.6) with `((x 1))`, use Propositional logic, and Arithmetic.
Now we have a proof for any conjecture we want, *e.g.*,

$$\varphi \tag{4.8}$$

How?
Well, `nil` (false) implies anything, so this is a theorem

$$\texttt{nil} \Rightarrow \varphi$$

Now, φ follows using (4.7) and Modus Ponens.

The point is that a context is *completely* different from a theorem. The context of (4.6) does not tell us that for all `x`, `x` = 1. It just tells us that `x` = 1 in the context of conjecture (4.6). Contexts are just a mechanism for extracting propositional structure from a conjecture, which in turn allows us to focus on the important part of a proof and to minimize the writing we have to do.

4.4 How to prove theorems

When presented with a conjecture, make sure that you check contracts, as shown above.

If the contracts checking succeeds, make sure you understand what the conjecture is saying.

Once you do, see if you can find a counterexample.

If you can't find a counterexample, try to prove that the conjecture is a theorem.

One often iterates over the last two steps.

During the proof process, you have available to you all the theorems we have proved so far. This includes all of the axioms (`first-rest` axioms, `if` axioms, ...), all the definitional axioms (def of `app`, `len`, ...), all the contract theorems (contracts of `app`, `len`, ...). These theorems can be used at any time in any proof and can be instantiated using any substitution. They are a great weapon that will help you prove theorems, so make sure you understand the set of already proven theorems.

There are also local facts extracted from the conjecture under consideration. Recall that the first step is to try and rewrite the conjecture into the form:

$$[C_1 \wedge C_2 \wedge \ldots \wedge C_n] \Rightarrow \text{RHS}$$

where we try to make RHS as simple as possible. $C_1, \ldots, C_n$ are going to be the first n components of our context. Formulas in the context are specific to the conjecture under consideration. They are completely different from theorems (as per the above discussion).

A good amount of manipulation of the conjecture may be required to extract the maximal context, but it is well worth it.

The next step is to see what other facts the $C_1, \ldots, C_n$ imply. For example, if the current context is:

Context:

C1. `(endp x)`

C2. `(listp x)`

then we would add

C3. `x` = `nil` { C1, C2 }

This will happen a lot. Another case that will happen a lot is:

C1. `(consp x)`

C2. `(listp x)`

C3. `(listp (rest x))` $\Rightarrow \varphi$

then we would add

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

C5. φ { C4, C3, MP }

where MP is modus Ponens.

As was the case when we studied propositional logic, we have "word problems," something we now consider:

Conjecture 10 $x \leq xy$ *if* $y \geq 1$

Discussion: What does the above conjecture mean, anyway?

It means that for any values of x, and y, if $y \geq 1$ then $x \leq xy$.

Really? Any values? What if x and y are functions or strings or ...? Usually the domain is implicit, *i.e.*, "clear from context."

We will be using ACL2s, and we can't appeal to "context." This is a good thing!

Notice also that we can use ACL2s, a programming language, to make mathematical statements. Of course! Programming languages are mathematical objects and you reason about programs the way you reason about the natural numbers, the reals, sets, etc.: you prove theorems.

In ACL2s, we have to be precise about the conditions under which we expect the conjecture to hold. The conjecture can be formalized in ACL2s as follows:

```
(thm
 (implies (and (rationalp x)
               (rationalp y)
               (>= y 1))
          (<= x (* x y))))
```

In standard mathematical notation it is:

$$\langle \forall x, y \in Q :: y \geq 1 \ \Rightarrow \ x \leq xy \rangle$$

Is the above conjecture true?

Well, when given a conjecture, we can try one of two things:

1. Try to falsify it.
2. Try to prove it is correct.

How do we falsify a conjecture?

Simple exhibit a counterexample.

Remember that in the design recipe, we construct examples and tests. You should do the same thing with conjectures. That is, we can test that the conjecture is true on examples. Here are some:

1. $x = 0, y = 0$
2. $x = 12, y = 1/3$
3. $x = 9, y = 3/2$

Any others?

How do we test this in ACL2s? Put the conjecture in the body of a let.

```
(let ((x 0)
      (y 0))
  (implies (and (rationalp x)
                (rationalp y)
                (>= y 1))
           (<= x (* x y))))
```

We are using a programming language, so we can do better. We can write a program to test the conjecture on a large number of cases. How many cases are there? We can use a random number generator to "randomly" sample from the domain. We'll see how to do that in ACL2s.

```
(test?
  (implies (and (rationalp x)
                (rationalp y)
                (>= y 1))
           (<= x (* x y))))
```

If all of the tests pass, then we can try to prove that the conjecture is a theorem.

What would a "proof" of the above conjecture look like?

Most proofs are informal and it takes a long time for students to understand what constitutes an informal proof. This happens by osmosis over time.

In our case, we have a simple rule: it's a proof if ACL2s says it is.

```
(thm
 (implies (and (rationalp x)
               (rationalp y)
               (>= y 1))
```

```
          (<= x (* x y))))
```

Of course, this isn't a theorem.

Let's consider another example:

Conjecture 11 *x(y+z) = xy + xz*

How do we write this in ACL2s?

```
(thm (implies (and (rationalp x)
                   (rationalp y)
                   (rationalp z))
              (equal (* x (+ y z))
                     (+ (* x y) (* x z)))))
```

Is the above conjecture true?

Well, we can try to falsify it.

```
(let ((x 0)
      (y 0)
      (z 0))
  (equal (* x (+ y z))
         (+ (* x y) (* x z))))
```

We can try many examples. We can automatically generate random examples.

```
(test?
 (implies (and (rationalp x)
               (rationalp y)
               (rationalp z))
          (equal (* x (+ y z))
                 (+ (* x y) (* x z)))))
```

When do we give up falsifying this?

Can we just try all the possibilities? If we had infinite time. Do we? Maybe (ask a physicist), but, as a practical matter, we currently don't.

Maybe we should consider a proof. Can we prove the above?

One answer might be: "of course, multiplication distributes over addition."

In ACL2s, the conjecture turns out to be true

```
(thm (implies (and (rationalp x)
                   (rationalp y)
                   (rationalp z))
              (equal (* x (+ y z))
                     (+ (* x y) (* x z)))))
```

This is pretty amazing because a proof gives us a finite way of running an infinite number of examples. That's the power of logic and mathematics.

When ACL2s proves this theorem, is it thinking?

> The question of whether Machines Can Think ... is about as relevant as the question of whether Submarines Can Swim.
>
> Edsger W. Dijkstra: EWD898, 1984

See the EWD archives at the University of Texas at Austin.
Here is another example

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

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

(defunc in (a X)
  :input-contract (listp x)
  :output-contract (booleanp (in a X))
  (if (endp x)
      nil
    (or (equal a (first X))
        (in a (rest X)))))

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

Conjecture 12

```
(listp x)
⇒
(in a x)   ⇒   (not (in a (del a x)))
```

Using induction (something we will describe later), the above conjecture leads to the following proof obligation:

```
(and (implies (endp x)
              (implies (listp x)
                       (implies (in a x)
                                (not (in a (del a x))))))
     (implies (and (consp x)
                   (equal a (first x)))
              (implies (listp x)
                       (implies (in a x)
```

```
                              (not (in a (del a x))))))
     (implies (and (consp x)
                   (not (equal a (first x)))
                   (implies (listp (rest x))
                            (implies (in a (rest x))
                                     (not (in a (del a (rest x)))))))
              (implies (listp x)
                       (implies (in a x)
                                (not (in a (del a x)))))))
```

Is this true? If so, give a proof. Is it false? If so, exhibit a counterexample.

Try this before reading further.

Conjecture 12 is false, *e.g.*, consider

```
(let ((x '(1 1))
      (a 1))
  ...)
```

where ... in the above let is Conjecture 12.

What about the following conjecture?

Conjecture 13

```
(listp x)
⇒
(in a x)   ⇒   (in a (app x y))
```

Which by induction leads to the following proof obligation:

```
(and (implies (endp x)
              (implies (listp x)
                       (implies (in a x)
                                (in a (app x y)))))
     (implies (and (consp x)
                   (equal a (first x)))
              (implies (listp x)
                       (implies (in a x)
                                (in a (app x y)))))
     (implies (and (consp x)
                   (not (equal a (first x)))
                   (implies (listp (rest x))
                            (implies (in a (rest x))
                                     (in a (app (rest x) y)))))
              (implies (listp x)
                       (implies (in a x)
                                (in a (app x y))))))
```

Is this true? If so, give a proof. Is it false? Is so, exhibit a counterexample.

Try this before reading further.

This is true and you should be able to prove it by breaking Conjecture 13 into three parts and proving each in turn.

In the cases we have seen so far, it was easy to decide if a conjecture was true or false, and with a good amount of testing, we would have identified the false conjectures.

Is this always the case?

No.

Anyone heard of Fermat's last theorem?

Conjecture 14 *For all positive integers x, y, z, and n, where $n > 2$, $x^n + y^n \neq z^n$.*

In 1637, Fermat wrote about the above:

> "I have a truly marvelous proof of this proposition which this margin is too narrow to contain."

This is called Fermat's Last Theorem. It took 357 years for a correct proof to be found (by Andrew Wiles in 1995).

We can use Fermat's last theorem to construct a conjecture that is hard to prove in ACL2s. After defining `expt`, a function that computes exponents, we define:

```
(defunc f (x y z n)
  :input-contract (and (posp x)
                       (posp y)
                       (posp z)
                       (natp n)
                       (> n 2))
  :output-contract (booleanp (f x y z n))
  (not (equal (+ (expt x n) (expt y n))
              (expt z n))))

(thm (f x y z n))
```

So, proving theorems may be hard.

Notice also that if the output contract was

```
:output-contract (equal (f x y z n) t)
```

then ACL2s would have to prove a theorem that eluded mankind for centuries in order to even admit `f`!

But, it is easy to find a counterexample to a conjecture that is not a theorem, right?

That is not true either. There are many examples of conjectures that took a long time to resolve, and which turned out to be false.

4.5 Arithmetic

We can also reason about arithmetic functions. For example, consider the following conjecture

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

That is, summing up $0, 1, \ldots, n$ gives $\frac{n(n+1)}{2}$.

We can prove this using mathematical induction.
Here is how we do it in ACL2s. First, we have to define Σ.

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

We can prove that `(sum n)` $= \frac{n(n+1)}{2}$, which is formalized as:

```
(implies (natp n)
         (equal (sum n)
                (/ (* n (+ n 1)) 2)))
```

by mathematical induction. How?
First, we have the base case.

$$\texttt{(equal n 0)} \wedge \texttt{(natp n)} \Rightarrow \texttt{(sum n)} = \texttt{(/ (* n n+1) 2)} \tag{4.9}$$

Second, we have the induction step.

$$\texttt{n} > 0 \wedge \texttt{(natp n)} \wedge \texttt{(sum n-1)} = \texttt{(/ (* n-1 n) 2)} \Rightarrow \texttt{(sum n)} = \texttt{(/ (* n n+1) 2)} \tag{4.10}$$

Here is the proof, starting with (4.9).
Context.

C1. `(natp n)`

C2. `(equal n 0)`

Proof.
`(sum n)`
= { Def of `sum`, C2 }
`0`
= { Arithmetic, C2 }
`(/ (* n n+1) 2)`

Here is the proof of (4.10).
Context.

C1. `(natp n)`

C2. $\texttt{n} \neq 0$

C3. `(natp n-1)` $\Rightarrow$ `(sum n-1) = (/ (* n-1 n) 2)`

C4. `(natp n-1)` { C1, C2 }

C5. `(sum n-1) = (/ (* n-1 n) 2)` { C3, C4, MP }

Proof.

```
  (sum n)
```

= { Def of `sum`, C2 }

```
  n + (sum (- n 1))
```

= { C5 }

```
  n + (/ (* n-1 n) 2)
```

= { Arithmetic }

```
  (2n + n(n-1))/2
```

= { Arithmetic }

```
  (2n + n^2 -n)/2
```

= { Arithmetic }

```
  (n^2 +n)/2
```

= { Arithmetic }

```
  n(n+1)/2
```