Reasoning About Programs

Panagiotis Manolios
Northeastern University

February 22, 2019
Version: 107

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, 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!

7

Steering the ACL2 Sedan

7.1 Interacting with ACL2s

Most of the material in this chapter comes from the Computer-Aided Reasoning book. As depicted in Figure 7.1, the theorem prover takes input from both you and a database, called the *logical world* or simply *world*. The world embodies a theorem proving strategy, developed by you and codified into *rules* that direct the theorem prover's behavior. When trying to prove a theorem, the theorem prover applies your strategy and prints its proof attempt. You have no interactive control over the system's behavior once it starts a proof attempt, except that you can interrupt it and abort the attempt. When the system succeeds, new rules, derived from the just-proved theorem, are added to the world according to directions supplied by you. When the system fails, you must inspect the proof attempt to see what went wrong.

Your main activity when using the theorem prover is designing your theorem proving strategy and expressing it as rules derived from theorems. There are over a dozen kinds of rules, each identified by a *rule class* name. The most common are rewrite rules, but other classes include type-prescription, linear, elim, and generalize rules. The basic command for telling the system to (try to) prove a theorem and, if successful, add rules to the database is the `defthm` command.

`(defthm` *name formula*
  `:rule-classes (`$class_1 \ldots class_n$`))`

The command directs the system to try to prove the given formula and, if successful, remember it under the name *name* and build it into the database in each of the ways specified by the $class_i$. To find out details of the various rule classes, see the documentation topic `rule-classes`.

You have lots of control over what the theorem prover can do. For example, every rule has a *status* of either *enabled* or *disabled*. The theorem prover only uses enabled rules. So by changing the status of a rule or by specifying its status during a particular step of a particular proof with a "hint" (see the documentation topic `hints`), you can change the strategy embodied in the world.

7.2 The Waterfall

So, how does ACL2 work? Let's look at the classic example that shows the ACL2 waterfall. This is in `ACL2s` mode.

```
(defun rev (x)
```

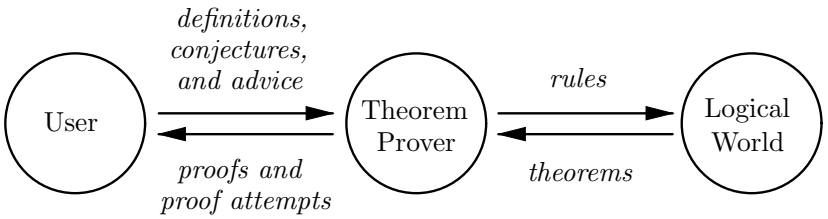

Figure 7.1: Data Flow in the Theorem Prover

```
  (if (endp x)
      nil
    (append (rev (rest x)) (list (first x)))))

(defthm rev-rev
  (implies (tlp x)
           (equal (rev (rev x)) x)))
```

Think of `defun` as `defunc` without contracts. See section 8.2 of the Computer-Aided Reasoning book for an in-depth discussion.

The `rev-rev` example nicely highlights the organization of ACL2, which is shown in Figure 7.2. At the center is a *pool* of formulas to be proved. The pool is initialized with the conjecture you are trying to prove. Formulas are removed from the pool and processed using six proof techniques. If a technique can reduce the formula, say to a set of $n \geq 0$ other formulas, then it deposits these formulas into the pool. In the case where n is 0, the formula is proved by the technique. If a technique can't reduce the formula, it just passes it to the next technique. The original conjecture is proved when there are no formulas left in the pool and the proof techniques have all halted. This organization is called "the waterfall."

Go over the proof output for the above theorem in Theorem Proving Beginner Mode in ACL2s.

7.3 Term Rewriting

It is easy to be impressed with what ACL2 can do automatically, and you might think that it does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you have to use your knowledge of the mechanization to figure out what the system is missing. And, an understanding of how simplification, and in particular rewriting, works is a requirement.

We are going to focus on rewriting, as the successful use of the theorem prover requires successful control of the rewriter.

You have to understand how the rewriter works and how the theorems you prove affect the rewriter in order to develop a successful proof strategy that can be used to prove the theorems you are interested in formally verifying.

Here is a user-level description of how the rewriter works. The following description is not altogether accurate but is relatively simple and predicts the behavior of the rewriter in

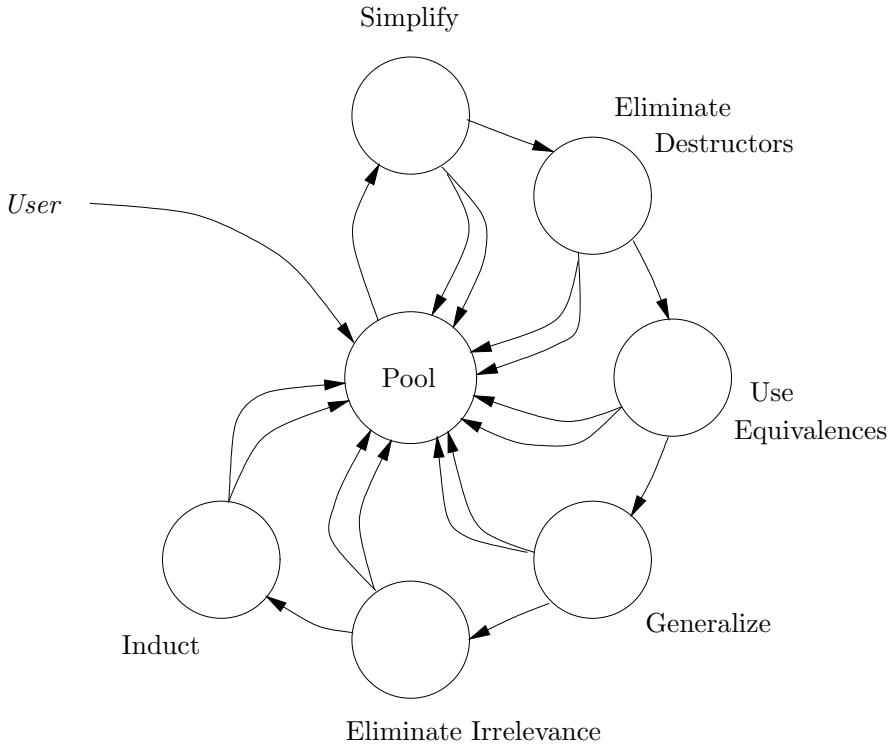

Figure 7.2: Organization of the Theorem Prover

nearly all cases you will encounter.

If given a variable or a constant to rewrite, the rewriter returns it. Otherwise, it is dealing with a function application, `(`f a_1 ... a_n`)`. In most cases it simply rewrites each argument, a_i, to get some a_i' and then "applies rewrite rules" to `(`f a_1' ... a_n'`)`, as described below.

But a few functions are handled specially. For example if f is `if`, the test, a_1, is rewritten to a_1' and then a_2 and/or a_3 are rewritten, depending on whether we can establish if a_1' is `nil`.

Now we explain how rewrite rules are applied to `(`f a_1' ... a_n'`)`. We call this the *target term* and are actually interested in a given occurrence of that term in the formula being rewritten.

Associated with each function symbol f is a list of rewrite rules. The rules are all derived from axioms, definitions, and theorems, as described below, and are stored in reverse chronological order – the rule derived from the most recently proved theorem is the first one in the list. The rules are tried in turn and the first one that "fires" produces the result.

A rewrite rule for f may be derived from a theorem of the form

```
(implies (and hyp_1 ... hyp_k)
         (equal (f b_1 ... b_n)
                rhs))
```

Note that the definition of f is of this form, where $k = 0$.

Aside: A theorem concluding with a term of the form `(not (`p ...`))` is considered, for these purposes, to conclude with `(iff (`p ...`) nil)`. A theorem concluding with `(`p ...`)`, where p is not a known equivalence relation and not `not`, is considered to conclude with `(iff (`p ...`) t)`.

Such a rule causes the rewriter to replace instances of the *pattern*, `(`f b_1 ... b_n`)`, with the corresponding instance of rhs under certain conditions as discussed below.

Suppose that it is possible to instantiate variables in the pattern so that the pattern matches the target. We will depict the instantiated rule as follows.

```
(implies (and hyp_1' ... hyp_k')
         (equal (f a_1' ... a_n')
                rhs'))
```

To apply the instantiated rule the rewriter must establish its hypotheses. To do so, rewriting is used recursively to establish each hypothesis in turn, in the order in which they appear in the rule. This is called *backchaining*. If all the hypotheses are established, the rewriter then recursively rewrites rhs' to get rhs''. Certain heuristic checks are done during the rewriting to prevent some loops. Finally, if certain heuristics approve of rhs'', we say the rule *fires* and the result is rhs''. This result replaces the target term.

7.3.1 Example

Suppose you just completed a session with ACL2s where you proved theorems leading to the following rewrite rules.

These rewrite rules were admitted in the give order, *i.e.*, 1 was admitted first, then 2, then 3, then 4.

1. `(f (h a) b) = (g a b)`

2. `(g a b) = (h b)`
3. `(g c d) = (h c)`
4. `(f (h a) (h b)) = (f (h b) a)`

- Some of the rewrite rules above can *never* be applied to *any* expression. Which rules are they?
- Show what ACL2s rewrites the following expression into. Show all steps, in the order that ACL2s will perform them.

```
(equal (f (f (h b) (h a)) b) (h b))
```

Answer:
1. Rule 2 cannot be applied because rule 3 will always match first.
2. Here are all the steps.

```
  (equal (f (f (h b) (h a)) b) (h b))
```
= { By Rule 4 }
```
  (equal (f (f (h a) b) b) (h b))
```
= { By Rule 1 }
```
  (equal (f (g a b) b) (h b))
```
= { By Rule 3 }
```
  (equal (f (h a) b) (h b))
```
= { By Rule 1 }
```
  (equal (g a b) (h b))
```
= { By Rule 3 }
```
  (equal (h a) (h b))
```

So, ACL2s will not prove the above conjecture. But, does there exist a proof of the above conjecture?

Yes! For example, if we use rule 2 instead of rule 3 at the last step, then ACL2s will be left with

```
(equal (h b) (h b))
```

which it will rewrite to `t` (using the reflexivity of `equal`). The point is that ACL2s is not a decision procedure for arbitrary properties of programs. In fact, this is an undecidable problem, so there can't be a decision procedure.

7.3.2 Three Rules of Rewriting

Let's consider how we can take what we learned to make effective use of the theorem prover.

Remember the three rules.

1. We saw that ACL2 uses lemmas (theorems) as rewrite rules. Rewrite rules are oriented, *i.e.*, they are applied only from left to right. We saw that theorems of the form

```
(implies ... (equal (foo ...) ...))
```

are used to rewrite occurrences of

```
(foo ...)
```

2. Rules are tried in reverse-chronological order (last first) until one that matches is found. If the rule has hypotheses, then we backchain, trying to discharge those hypotheses. If we discharge the hypotheses, we apply the rule, and recursively rewrite the result.

3. We saw that rewriting proceeds inside-out, *i.e.*, first we rewrite the arguments to a function before rewriting the function.

7.3.3 Pitfalls

Suppose we have both of the following rules.

```
(defthm app-associative
  (implies (and (tlp x) (tlp y) (tlp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))

(defthm app-associative2
  (implies (and (tlp x) (tlp y) (tlp z))
           (equal (app x (app y z))
                  (app (app x y) z))))
```

Ignoring hypotheses for the moment, when we try to rewrite

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

We wind up getting into an infinite loop. So notice that you can have non-terminating rewrite rules. ACL2 does not check that rewrite rules are non-terminating, so if you admit the above two rules, you can easily cause ACL2 to chase its tail forever. Non-termination here does not cause unsoundness; all that happens is that ACL2 becomes unresponsive and you have to interrupt it, but non-terminating rewrite rules will *never* allow you to prove `nil`. Contrast this with non-terminating function definitions, which, as we have seen, can lead to unsoundness.

7.3.4 Examples

Suppose that we have the following rule.

```
(defthm app-associative
  (implies (and (tlp x) (tlp y) (tlp z))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

What does the following get rewritten to?

```
(implies (and (tlp a) (tlp b) (tlp c) (tlp d))
```

```
        (equal (app (app (app a b) c) d)
               (app (app a (app b c) d))))
```

Here is another separate example. Suppose that we have the following two rules.

```
(defthm +-commutative
  (implies (and (rationalp x) (rationalp y))
           (equal (+ x y)
                  (+ y x))))
```

Oops. This seems like a really bad rule. Why?

ACL2 is smart enough to identify rules like this that permute their arguments. It recognizes that they will lead to infinite loops, so it only applies when the term associated with `y` is "smaller" than the term associated with `x`. The details are not relevant. What is relevant is that this does not lead to infinite looping. Also a variable is smaller than another if it comes before it in alphabetical order, and a variable is smaller than a non-variable expression, *e.g.*, `x` is smaller than `(f y)`.

We also have

```
(defthm +-associative
  (implies (and (rationalp x) (rationalp y) (rationalp z))
           (equal (+ (+ x y) z)
                  (+ x (+ y z)))))
```

What does the following get rewritten to?

```
(implies (and (rationalp a) (rationalp b) (rationalp c))
         (equal (+ (+ b a) c)
                (+ a (+ c b))))
```

What does this get rewritten to?

```
(implies (and (rationalp a) (rationalp b) (rationalp c))
         (equal (+ a (+ b c))
                (+ (+ c b) a)))
```

Can you prove using equational reasoning that the above conjecture is true?

Yes, but rewriting does not discover this fact. Because rewriting is directed. What does one do in such situations?

Add another rule, *e.g.*, :

```
(defthm +-swap
  (implies (and (rationalp x) (rationalp y) (rationalp z))
           (equal (+ x (+ y z))
                  (+ y (+ x z)))))
```

Now what happens to the above?

7.3.5 Generalize!

The previous example shows that sometimes we have to add rewrite rules in order to prove conjectures (by rewriting).

As a general rule, we may have many options for adding rewrite rules and we want to do it in the most general way.

For example, suppose that during a proof, we are confronted with the following *stable* subgoal, where a subgoal is stable if none of our current rewrite rules can be applied to any subexpression of the subgoal.

```
(... (app (rev (app x y)) nil) ...)
```

We realize that `(app (rev (app x y)) nil)` is equal to `(rev (app x y))`, so we need a rewrite rule that allows us to simplify the above further. One possibility is:

```
(defthm lemma1
  (implies (and (tlp x) (tlp y))
           (equal (app (rev (app x y)) nil)
                  (rev (app x y)))))
```

But a better, and more general, lemma is the following.

```
(defthm lemma2
  (implies (tlp x)
           (equal (app x nil)
                  x)))
```

Why is the second lemma better? Because given the contracts for `app` and `rev`, any time we can apply lemma1, we can apply lemma2, but not the other way around. That means that lemma2 allows us to simplify more expressions than lemma1.