CSU290, Logic and Computation — Wed, Mar 26, 2008
Pete Manolios — Lecture 25

1 Announcements

Exam tomorrow.

Friday, March 28th is the last day to drop a class with a W grade.

Reading assignment for this part of the course: Chapters 8 and 9 of the book.

2 The ACL2 Theorem Proving System

We are now in a new part of the course. We are going to learn how ACL2 works and how to use it to prove theorems.

Recall last time, the last thing we tried was:

```
(thm (equal (true-listp (append x y))
            (true-listp y)))
```

Now if we try

```
(thm (true-listp (append x nil)))
```

we may expect ACL2 to get it since it "knows" the more general theorem we just proved. However, you have to tell ACL2 to remember theorems it proves, by using `defthm`, so let's try this again:

```
(defthm true-listp-append
  (equal (true-listp (append x y))
         (true-listp y)))
```

Now if we try

```
(thm (true-listp (append x nil)))
```

we get the expected behavior.

3 Interacting with ACL2

As depicted in Figure 1, the theorem prover takes input from both you and a data base, 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,

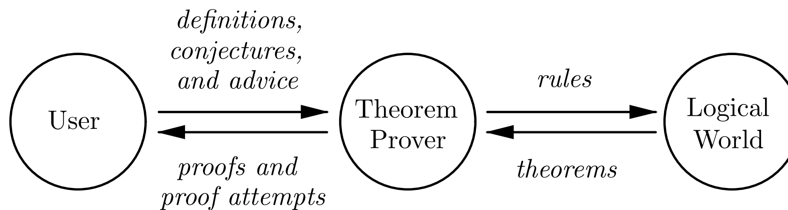

Figure 1: Data Flow in the Theorem Prover

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 data base is the `defthm` command.

```
(defthm name formula
  :rule-classes (class_1 ... 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 data base 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.

4 The Waterfall

So, how does ACL2 work? Let's look at the classic example that shows the ACL2 waterfall:

```
(defthm rev-rev
  (implies (true-listp x)
           (equal (rev (rev x)) x)))
```

See section 8.2 of your book, because we are going to cover exactly that.

The `rev-rev` example nicely highlights the organization of ACL2, which is shown in Figure 2. At the center is a *pool* of formulas to be proved. The pool

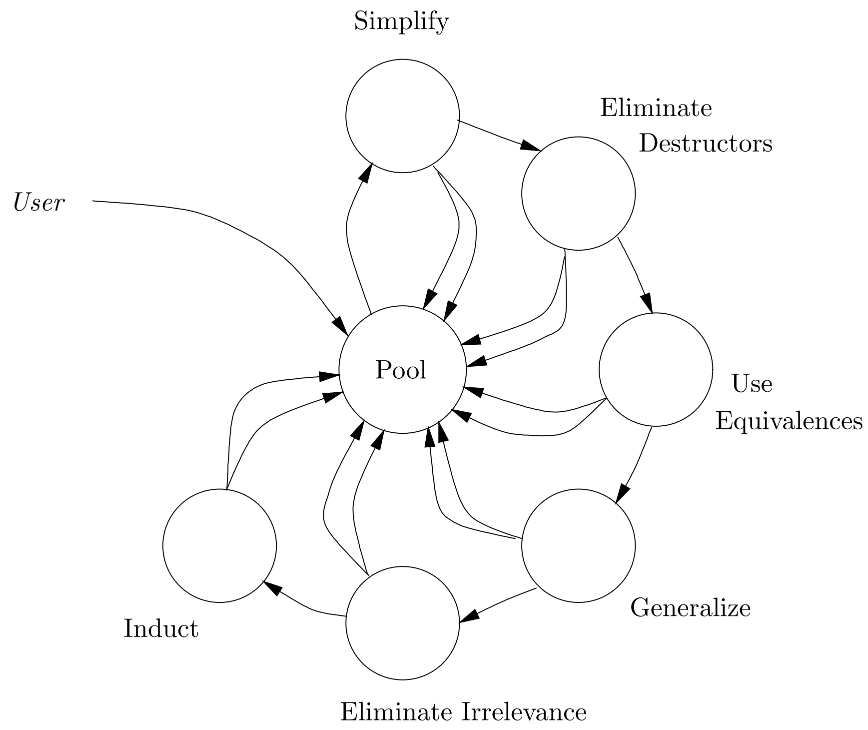

Figure 2: Organization of the Theorem Prover

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".