CSU290 Lecture Notes   Lecture 27 Part B   24 Nov 2008

Peter Dillinger

Theorems as Rules in ACL2

When you prove a theorem with DEFTHM, ACL2 by default will create a "rewrite rule" for use in future proofs. Let’s examine the anatomy of rewrite rules with examples from Part A of the lecture notes:

```
  (defthm len--insert
    (equal (len (insert x y))
           (+ 1 (len y))))
```

When you prove something of the form

```
  (equal <lhs>
         <rhs>)
```

Then ACL2 creates a rewrite rule to rewrite instances of <lhs> into <rhs>. <lhs> stands for Left Hand Side and <rhs> for Right Hand Side. Variables in <lhs> can match anything, because we can use instantiation to replace each variable with any expression.

So the theorem named LEN--INSERT above tells ACL2 to look for instances of

```
  (len (insert x y))
```

in what it is proving (where x and y can be anything), and rewrite those into

```
  (+ 1 (len y))
```

where y is replaced with what matched it in the left hand side. Conceptually, this is desirable because the right hand side, (+ 1 (len y)) is simpler to work with. In fact, we’ve completely removed x from expression; it turns out to be irrelevant to what (len (insert x y)) is equal to.

The next example is

```
  (defthm len--isort
    (equal (len (isort x))
           (len x)))
```

This of course searches for instances of

```
  (len (isort x))
```

where x can be anything, and rewrites them into

```
  (len x)
```

This is obviously simpler, even though we weren’t able to eliminate any variables in the left hand side from the right hand side.

This brings up an interesting point. If we had written

```
  (defthm len--isort2
    (equal (len x)
           (len (isort x))))
```

that is an equivalent proposition (because of the symmetry of equality), so it is also a theorem. But when given to ACL2, the rewrite rule created is different from LEN--ISORT because the right and left hand sides are switched.

This new one will look for instances of

```
  (len x)
```

and rewrite them to

```
  (len (isort x))
```

That is not at all useful. In fact, if ACL2 encounters

```
  (len (blah x))
```

it would rewrite that to

```
  (len (isort (blah x)))
```

That too is an instance of (len x), so it would rewrite that to

```
  (len (isort (isort (blah x))))
```

and that to

```
  (len (isort (isort (isort (blah x)))))

  etc.
```

The lesson is that the order of the equality matters when it is used to create a rewrite rule. It is best to rewrite more complicated things into simpler ones but it is not always clear which is simpler.

The next example is

```
  (defthm true-listp--isort
    (true-listp (isort x)))
```

This is not an equality and not an implication (see below), so basically the whole thing is the left hand side and the right hand side is T.

There is a subtle limitation though: just because (true-listp (isort x)) is a theorem does not mean it is always equal to T. It just means it is always non-nil. There are many places in theorems where it only matters whether something is nil or not; the actual value does not matter. These places are called "boolean context" and these are where ACL2 rewrites formulas like this to T. For example

```
  (if (true-listp (isort y))
    a
    b)
```

is equal to

```
  (if T
    a
    b)
```

by the rewrite rule, because the test of an IF is boolean context.

Also,

```
  (implies <something>
           (true-listp (isort z)))
```

is equivalent to

```
  (implies <something>
           T)
```

which, by propositional reasoning, is equivalent to

```
  T
```

However, in

```
  (integerp (true-listp (isort x)))
```

the rewrite rule does not apply because the argument to integerp is not
a boolean context. It usually works to think of ACL2 as rewriting formulas
like (true-listp (isort x)) to True only if it makes sense for them to be
only True or False.

The next example is

```
  (defthm true-listp--insert
    (implies (true-listp y)
             (true-listp (insert x y))))
```

When the formula is an implication, the first part are hypotheses:

```
  (implies <hyp>
           <conc>)

  (implies (and <hyp1>
                <hyp2>
                ...
                <hypn>)
           <conc>)
```

And <conc> can be just a <lhs> to rewrite to T or (equal <lhs> <rhs>).

Hypothesis identify the conditions under which a rewrite rule can be used.
When a rewrite rule matches, ACL2 quietly attempts a miniature proof of
the hypotheses of the rewrite rule. That short proof attempt is called
Backchaining. If it succeeds, the rule is applied, but if it fails, ACL2
proceeds with the original proof without applying the rule.

```
  (implies (true-listp y)
           (true-listp (insert x y)))
```

means that ACL2 will search for instances of

```
  (true-listp (insert x y))
```

where x and y can be anything. When it finds that, it will attempt to prove
(true-listp y), where y is what was matched for y in (true-listp (insert x y)).

For example, if I were to ask ACL2 to prove

```
  (true-listp (insert e (isort x)))
```

It will see that the rule TRUE-LISTP--INSERT matches (x is e, y is (isort x)),
but there is a hypothesis to prove. Backchaining will call for proof of

```
  (true-listp (isort x))
```

This rewrites to T by the rewrite rule TRUE-LISTP--ISORT, so the rule
TRUE-LISTP--INSERT can be applied, which rewrites the conjecture to T.