Rewriting in the ACL2 Theorem Prover

This lecture covers the content of §8.3.4 in the reference manual for ACL2.

When covering the functioning of the ACL2 prover in the last two lectures, we kind of went quickly over simplification—the first step in the cascade known as the waterfall.

Simplification involves using axioms and previously proved theorems to try to simplify the current goal. That's vague. How does the system know which previous theorems to use, and when?

The answer is that it follows a fixed set of rules, and you need to understand those rules to understand how ACL2 tries to prove things.

The first thing to realize is that ACL2 simplifies expressions using *rewrite rules*. A rewrite rule is simply a rule that says to rewrite part of an expression into another one.

Where do rewrite rules come from?

- Theorems of the form `(= exp exp')` yield an *unconditional rewrite rule* that rewrites expressions of the form `exp` into expressions of the form `exp'`.
- Theorems of the form `(implies (and` C_1 `...` C_k`) (= exp exp'))` yield a *conditional rewrite rule* that rewrites expressions of the form `exp` into expressions of the form `exp'`.

Note that the order in which expressions are listed in the equality formula is important when considering the rewrite rules produced by a theorem. In particular, proving:

```
(defthm app-assoc (= (app (app x y) z) (app x (app y z))))
```

and proving

```
(defthm app-assoc (= (app x (app y z)) (app (app x y) z)))
```

while meaning the same thing as *theorems*, yield two different rewrite rules, the first that can rewrite expressions of the form `(app (app x y) z)` into `(app x (app y z))`, and the second that can rewrite expressions of the form `(app x (app y z))` into `(app (app x y) z)`.

Let's now look at the *rewriting algorithm*, that says, given a list of rewrite rules *in the temporal order in which they were added to the system*, how an arbitrary expression gets rewritten. Intuitively, the algorithm says: try to apply the rewrite rules, in *last-to-first* order, to all the sub-expressions in the expressions, from *inside-out and left-to-right*. The first rewrite that is applicable, perform it, and return the result.

Formally, here it is. The harder is to formalize the *inside-out* and *left-to-right* bit: if your expression to be rewritten is of the form `(f` $\texttt{exp}_1$ $\texttt{exp}_2$ `...` $\texttt{exp}_k$`)`, try to rewrite $\texttt{exp}_1$, then try to (recursively) rewrite $\texttt{exp}_2$, then ..., then try to rewrite $\texttt{exp}_k$, before trying to rewrite `(f` $\texttt{exp}_1$ $\texttt{exp}_2$ `...` $\texttt{exp}_k$`)`.

For example, if the expression you are trying to rewrite is `(foo (bar a) (bar (baz b)))`, then you will first try to rewrite `(bar a)`, then `baz b`, then `(bar (baz b))`, then `(foo (bar a) (bar (baz b)))`.

What do I mean try to rewrite an expression? Look at all the rewrite rules, from the last one added to the system going back to the first one added to the system, and see if the left-hand side of a rewrite rule matches the sub-expression you are trying to rewrite. If it does, then the rule *fires*, and you replace the sub-expression by what the right-hand side tells you to replace it with (taking care to replace the variable by whatever the matching was).

What do I mean by a match? An expression `exp` matches another expression `exp'` if there is a way to substitute the variables in `exp'` by expressions so that the result is equal to `exp`. In other words, `exp` matches `exp'` if `exp` is *an instance of* `exp'`.

For example, `(foo (bar a))` matches `(foo x)`, by replacing `x` by `(bar a)`. Similarly, `(foo (bar a) (bar (baz b)))` matches `(foo (bar x) (bar y))` by replacing `x` by `a` and `y` by `(baz b)`.

An unconditional rewrite rule fires when the left-hand side of the equality matches the expression you are trying to rewrite. A conditional rewrite rules fires when the left-hand side of the equality matches the expression you are trying to rewrite *and* the conditions $\texttt{C}_1$, ..., $\texttt{C}_k$ are all true. (You therefore need to stop and check whether those conditions are true, either because they follow from your context, possibly by rewriting them themselves.)

So, let's look at an example. Suppose that we have prove the following theorems, in the following order:

```
1:  (= (foo (bar x) (bar y)) x)
2:  (= (bar x) x)
3:  (= (foo x y) y)
```

And suppose you are asked to rewrite `(foo (bar a) (bar (baz b)))` repeatedly until no more rewrites are possible.

Start with `(foo (bar a) (bar (baz b)))`. The first rewrite goes as follows: look at all the sub-expressions, in the order we described above. First, try to rewrite `(bar a)`. Does rule 3 match? no. Does rule 2 match? Yes, with `x` replaced by `a`. So `(bar a)` rewrites into

`a`, meaning that the whole expression `(foo (bar a) (bar (baz b)))` rewrites into `(foo a (bar (baz b)))`.

Now, you have `(foo a (bar (baz b)))`. Let's rewrite that. Look at the sub-expressions in order. First, try to rewrite `a`. No rule matches. Next, try to rewrite `(baz b)`. Again, no rule matches. Next, try to rewrite `(bar (baz b))`. Rule 3 does not match, but rule 2 does, with `x` replaced by `(baz b)`. Thus, `(bar (baz b))` rewrites into `(baz b)`, meaning that the expression `(foo a (bar (baz b)))` as a whole rewrites into `(foo a (baz b))`.

Now, you have `(foo a (baz b))`. Let's rewrite that. Look at the sub-expression in order. First, try to rewrite `a`. No rule matches. Next, try to rewrite `(baz b)`. Again, no rule matches. Then, try to rewrite `(foo a (baz b))`. Rule 3 matches, with `x` replaced by `a` and `y` replaced by `(baz b)`. Thus, `(foo a (baz b))` rewrites into `a`.

So, expression `(foo (bar a) (bar (baz b)))` rewrites as follows:

$$\begin{aligned} &\texttt{(foo (bar a) (bar (baz b)))} \\ &\longrightarrow \texttt{(foo a (bar (baz b)))} \\ &\longrightarrow \texttt{(foo a (baz b))} \\ &\longrightarrow \texttt{a} \end{aligned}$$

Two very important things to note:

(1) The *order* in which you prove theorems is important: because rewrite rules are looked at from the last one added in to the first one added in, proving some theorems before others can change how rewriting is done. That's one of the main ways to control how ACL2 proves theorems.[1]

(2) It is possible to get the rewriter to loop (or generally, not terminate) by having rewrite rules that either eventually rewrite an expression to itself, or make an expression grow and grow. For instance, a theorem such as `(= (foo x) (foo (foo x)))` yields a rewrite rule that would make `(foo a)` rewrite into `(foo (foo (foo (foo (foo ...)))))`, without ever ending.

The distinction between conditional and unconditional rewrite rules means that you should try to prove theorems that give unconditional rewrite rules as much as possible. Here is an example.

Consider the following formula:

$$\texttt{(true-listp y)} \Longrightarrow \texttt{(true-listp (app x y))} \tag{1}$$

[1] There are some ways to selective enable and disable rewrite rules for some theorems—I'll let you look it up in the documentation if you're curious.

That's a valid formula, and it is in fact provable. However, the resulting rewrite rule is a conditional rewrite rule. It can only be used to replace `(true-listp (app x y))` by `T` whenever we can show that `(true-listp y)` holds. If we can't establish that `(true-listp y)` holds, then the rewrite rule cannot fire.

In particular, `(true-listp (app 5 6))` cannot be simplified with that rewrite rule.

On the other hand, if instead of (1) we prove the similar:

```
(= (true-listp (app x y)) (true-listp y))
```
(2)

then the rewrite rule is an unconditional rewrite rule. (Note that (2) is a generalization of (1): if we have (2), we can prove `(true-listp)` $\Longrightarrow$ `(true-listp (app x y))` easily.)

In particular, if we have `(true-listp (app 5 6))`, then we can simplify it to `(true-listp 6)`. So we *can* do something with that formula, even though `(true-listp 6)` may not be known.