CSU290, Logic and Computation — Mon, Mar 31, 2008
Pete Manolios — Lecture 26

# 1 Announcements

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

Informal homework: use ACL2 to prove all of the theorems we've seen in class.

# 2 Review

Last time we looked at the internals of the ACL2 theorem proving system. We went over the waterfall, and the basic proof techniques used by ACL2, e.g., simplification, destructor elimination, use of equivalences, generalization, elimination of irrelevance, and induction. For example, we saw that there is a pool of formulas, initially your conjecture, and ACL2 tries each of the proof techniques one at a time until one of them is applicable, in which case the formula is reduced to $n \geq 0$ formulas, which are re-inserted into the pool. If the pool become empty, ACL2 has proven your theorem. We saw that during the proof ACL2 may decide it needs lemmas that are then recorded and returned to after it finishes the proof (just like we do in class). And so on.

We also gave an overview of what the standard interaction between the user, the theorem prover, and the logical world is. The idea being that there is a logical world that corresponds to everything ACL2 knows. When you define functions, prove theorems, etc. you have to get the theorem prover's blessing. If it deems your functions admissable and your conjectures provable, then the logical world is updated. By updating the logical world, you can impact the future behavior of ACL2, as we saw. For example, we saw that after proving a theorem ACL2 was able to use the theorem to simplify future conjectures.

# 3 Using ACL2

Now, we are going to see how we can prove theorems using ACL2.

Let's look at the `rev-rev` example again.

Question: is the proof ACL2 came up with the same as the proof we came up with? No. Why not?

Exercise: Let's try to mimic the proof we gave in class. Notice that it is different from the proof ACL2 gave for `rev-rev`.

Let's do that.

When we do, we notice that we need lemmas, and that proofs fail. We can navigate failed proofs by using the proof tree viewer in ACL2s to jump to "checkpoints", interesting points during the proof process.

The first message that mentions destructor elimination, use of equalities, cross-fertilization, generalization, irrelevant terms, or induction should raise a red flag. Simplification has done all it can. The formula just above that message is the most important checkpoint in the output. We call it the *simplification checkpoint*. The important thing about the formula at the simplification checkpoint is that it is stable under simplification.

Study the conjecture at the checkpoint. Ask yourself

- Is the formula even valid? Perhaps your "theorem" is not a theorem.
- If it is valid, why? Sketch a little proof.
- Which theorems are used in that little proof that are not in the logical world?
- If all the theorems you need are in the world, why were they not applied? There are three common answers:
  - The pattern of a key rule does not match what is in the formula being proved. Perhaps another rule fired, messing up the pattern you expected.
  - Some hypothesis of the rule cannot be established.
  - The rule is not available (disabled or proven with `thm`).
- If there is a missing theorem, is it suspiciously like the one you are trying to prove? If so, perhaps the wrong induction was done or the formula you are trying to prove is too weak.
- On the other hand, if there is a missing theorem and it is different from the one you are trying to prove, then you have probably identified a key lemma. Most often, such lemmas are about new combinations of functions from the original theorem and the functions introduced by rewriting.
- Can you phrase the missing theorem as a rewrite rule so that the checkpoint goal simplifies further, ideally to true?

If answers come to you, repair the script accordingly and proceed with proof.

## 3.1 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 use 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 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.