ACL2 Computed Hints: Extension and Practice

Jun Sawada
IBM Austin Research Laboratory
Email: sawada@us.ibm.com

Abstract

ACL2 computed hints dynamically calculate advice to the ACL2 theorem prover during a mechanical proof. We wrote an ACL2 book that adds a number of useful functions and macros to ease the use of computed hints. The combination of these macros can specify a complex condition under which a certain hint is invoked. We will also review the usage of computed hints in the FM9801 project.

1 Introduction to ACL2 Computed Hints

ACL2 is an automated theorem prover, in the sense that it attempts to prove theorems without step-by-step human assistance. When the user wants to guide the direction of a mechanical proof, he can supply *hints* to the prover. Typically, a hint is given as an alternating list of hint keywords and arguments, preceded by a *goal-spec*, A goal-spec is a character string, such as `"Goal"` and `"Subgoal *1.2/3.4"`, that specifies the subgoal to which the hint is applied.

For example, the following `defthm` applies the `:in-theory` hint to the subgoal named `"Goal'"`.

```
(defthm test-one (implies (f x) (g x y))
  :hints (("Goal'" :in-theory (enable g-is-true))))
```

ACl2 provides the computed hint mechanism [KM99] which dynamically calculates hints to the prover. A computed hint can be either a function of three arguments, or an expression possibly with free variables, `id`, `clause`, and `world`, which represent the goal-spec, the target clause and the property list storing the state of ACL2. All computed hints discussed in this paper are in fact macros that generate expressions of the latter form.

We defined a computed hint book that extends the ACL2 computed hint mechanism. The small book with 270 lines of ACL2 functions and macros eases the specification of computed hints. In the following sections, we will look at the major features implemented by this library.

2 Goal-Spec Extension

An ordinary ACL2 hint can be applied to a single subgoal, whose goal-spec is provided by the user. The first extension provided by our computed hint library enables us to collectively specify the subgoals to which the hint is applied.

Our computed hint library extends *clause identifiers* to specify sets of subgoals. A clause identifier is an internal representation of a goal-spec. For example, the goal spec `"Subgoal *1.2.3/4.5.6''"` is represented by a clause identifier ((0 1 2 3) (4 5 6) . 2). (See [KM99] for details.) We specify a set of clause identifiers using wild cards `*`. Table 1 lists a correspondence between clause identifiers with wild cards and goal-specs.

Macro `when-GS-match` takes a clause identifier with wild cards, and an alternating list of hint keywords and arguments. In the following `defthm`, the `:in-theory` hint is invoked for any `"Subgoal` n`"` for arbitrary n.

```
(defthm f-iterated
  (and (f x) (f (f x)) (f (f (f x))))
  :hints ((when-GS-match ((0) (*) . 0)
                         :in-theory (enable f-is-true))))
```

The current version of ACL2 removes invoked computed hints from the list of hints passed to the descendent subgoals. Therefore, the computed hint may not be invoked on all subgoals specified by the clause identifier with wild cards. An easy workaround to apply a computed hint to descendent subgoals is adding duplicate copies of the computed hint to the list of hints.

Clause ID with wild cards	Corresponding goal-specs
`(* * . *)`	Any goal-specs
`((0 1 2) * . 2)`	`Subgoal *1.2/`$n_0.n_1 \ldots n_i$`''`
`((0) (1 2 *) . 0)`	`Subgoal *1.2.`n_0
`((0) (1 2 . *) . 0)`	`Subgoal *1.2.`$n_0.n_1 \ldots n_i$
`((3 *) (1 2) . *)`	`[3]Subgoal *`n_0`/1.2'`$\cdots$`'`

Table 1: A list of example clause identifiers with wild cards and the corresponding goal-specs. Variable n_k ranges over positive integers, and i ranges over non-negative integers. The goal-spec of the last example is followed by an arbitrary number of apostrophes. To be precise, ACL2 prints out the sequence of n apostrophes as “'n'” when n is larger than 3.

3 Occurrence Check and Pattern Matching

Macro `when-occur` in our library invokes hints when a particular sub-term occurs in the target clause. For example, the following `defthm` uses the lemma `foo-is-true` whenever a sub-term `(foo z)` is found in the clause.

```
(defthm p-is-true-1  (p z)
  :hints ((when-occur (foo z)
              :use (:instance foo-is-true (x z)))))
```

A simple extension of this macro is `when-multiple-occur`, which invokes a given hint when all the terms in the list given as an argument occur in the clause.

```
(defthm p-is-true-2  (p z)
  :hints ((when-multiple-occur ((buz z) (bar z))
              :use (:instance bar-or-buz (x z)))))
```

The user should know that all macros in the clause are expanded before occurrence check takes place. Thus, expanded forms of macros, such as `(binary-+ x y)` instead of `(+ x y)`, should be given to the computed hint. Expressions containing logical operators, such as `AND`, may not be detected either, even though they may appear in the pretty-printed subgoal.

We can enhance the occurrence checking with pattern matching. First, we extend the ACL2 syntax by introducing *meta-variables*. Meta-variables are the variables that can be instantiated during pattern matching. A meta-variable is denoted by a list of the symbol `@` and another symbol. For instance, pattern `(f x (@ z))` can be matched to the term `(f x y)` by instantiating meta-variable `(@ z)` with `y`.

Macro `when-pattern` invokes a hint when a subexpression of the target clause is an instance of a given pattern. Meta-variables in the hint are instantiated in the way the pattern is instantiated. In the following example, macro `when-pattern` finds that the pattern `(f (@ z))` matches `(f (f y))` by instantiating `(@ z)` with `(f y)`. Thus, the hint is instantiated to `:use (:instance f-is-true (x (f y)))` before invoked.

```
(defaxiom f-is-true (f x) :rule-classes nil)

(defthm f-of-f-of-x (f (f y))
  :hints ((when-pattern (f (@ z))
            :use (:instance f-is-true (x (@ z))))))
```

In fact, there are two possible pattern matchings for this example, depending on whether meta-variable `(@ z)` is instantiated with `y` or `(f y)`. The current implementation of `when-pattern` uses the first instantiation found by the pattern matcher, and ignores the rest.

4 Literal Matching

The ACL2 theorem prover converts a given ACL2 formula into a conjunctive normal form, and then attempts to prove each clause separately. A clause is a disjunction of literals, where a literal is an atomic formula or the negation of one. We say that an atomic formula occurs positively for the former case, and negatively for the latter case.

For example, consider an expression (IMPLIES (AND (NOT p) q) r), where p, q, and r are either constants, variables, or applications of functions other than built-in boolean functions. ACL2 converts the expression into a clause $p \vee \neg q \vee r$. In this clause, p and r occur positively and q occurs negatively.

Macros `when-occur-negative` and `when-occur-positive` invoke hints when a given term occurs negatively and positively, respectively, in the target clause. For example, let us consider the following `defthm`.

```
(defthm complex-lemma
  (and (implies (f y) (f (g x x)))
       (implies (h (h y)) (and (h y) (g x y))))
  :rule-classes nil
  :hints ((when-occur-negative (h (h y))
             :use (:instance h-h-x-is-false (x y)))
          (when-occur-positive (f (g x x))
             :use (:instance f-is-true (x (g x x))))))
```

From the body of this `defthm`, three clauses are generated: ¬`(f y)` ∨ `(f (g x x))`, ¬`(h (h y))` ∨ `(h y)`, and ¬`(h (h y))` ∨ `(g x y)`. The attached computed hints invoke Lemma `f-is-true` for the first clause, and Lemma `h-h-x-is-false` for the last two clauses.

The computed hint library also defines macro `when-pos/neg-occur`, which combines literal matching with pattern matching.

5 Combination of Computed Hints

In the preceding sections, we have seen separate use of macros defined in our computed hint book. However, the real strength of these macros can be seen when they are combined.

Most of the macros in our computed hint book have the versions with `-&` suffices, which take a computed hint as an argument instead of an alternating list of hint key words and their arguments. For example in the following `defthm`, macro `when-not-GS-match-&` and `when-pattern` are combined.

```
(defthm f-iterated-2
  (and (f x) (f (f x)) (f (f (f x))))
  :hints ((when-not-GS-match-& ((0) nil . 0)
            (when-pattern (f (@ v))
                 :use ((f-is-true ((x (@ v)))))))))
```

This hint is invoked when a pattern `(f (@ v))` is found in the clause and when the goal-spec is not `"Goal"`. With the simple use of `when-pattern` macro, the ACL2 does not successfully prove the theorem, because the hint is invoked for `"Goal"` and removed from the list of hints passed to the descendent subgoal. By combining two macros, we can successfully prove the theorem by invoking the computed hint for `"Subgoal 1"`, `"Subgoal 2"` and `"Subgoal 3"` with different instantiations.

6 Computed Hints in Practice

We discuss the use of computed hints in the FM9801 project. The FM9801 [Saw99a] is a microprocessor model with various features found in today's microprocessors: speculative execution with branch prediction, out-of-order execution of instructions with multiple pipelined execution units, and exceptions. We mechanically verified that the entire FM9801 microarchitectural model implements its ISA model using the ACL2 theorem prover. Its proof script is publicly available [Saw99b].

There are two merits of using computed hints in a large-scale project like FM9801. One is the improved robustness of the proof script, and the other is the conciseness of the proof script and the improved proof automation.

The FM9801 proof script contains more than 700 function definitions and 3800 theorems. For a hardware verification project of this size, the robustness of the proof script is important. Otherwise, the mechanical proof fails on already proven theorems after a slight modification of a hardware model.

Goal-specs given to an ordinary hint are one source of fragility in proof scripts. Slight modification of function definitions may change the goal-spec for a particular subgoal. With the computed hint with occurrence check, we can invoke hints for proper subgoals without specifying the exact goal-specs.

Unlike an ordinary hint, the computed hint can specify hints for multiple subgoals. The proof of a complex theorem generates a number of subgoals, many of which can be solved by applying the same lemma. Such subgoals often contain common sub-terms, which can be the triggers of the computed hint that invokes the necessary lemma. This type of computed hints substantially improve the automation of the mechanical proof, eliminating the need of explicitly specifying all the subgoals to which the lemma should be applied.

ACL2 computed hints have huge potential, but our book only covers very limited applications. We have not used the state variable `world` yet. In fact, the implementation of our computed hint extensions are relatively easy because internal structure of variables `id` and `clause` are simple. In order to implement macros that access the `world` variable, we need know how the variable stores the ACL2 state.

Even though we did not explore all the possibilities of computed hints, our exercise clearly points to the advantage of computed hints. We hope in the future more ACL2 users will use computed hints in their projects.

References

[KM99] Matt Kaufmann and J Strother Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual*. 1999. URL:http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html#User's-Manual.

[Saw99a] Jun Sawada. *Formal Verification of an Advanced Pipelined Machine.* PhD thesis, University of Texas at Austin, December 1999. Also available from http://www.cs.utexas.edu/users/-sawada/dissertation/diss.html.

[Saw99b] Jun Sawada. Verification scripts for FM9801 pipelined microprocessor design, 1999. URL:http://www.cs.utexas.edu/users/-sawada/FM9801/.