Meta Reasoning in ACL2

Warren A. Hunt, Jr., Matt Kaufmann, Robert Bellarmine Krug, J Moore, and Eric Whitman Smith

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
E-mail: hunt@cs.utexas.edu, kaufmann@cs.utexas.edu, rkrug@cs.utexas.edu, moore@cs.utexas.edu, ewsmith@stanford.edu

Abstract. The ACL2 system is based upon a first-order logic and implements traditional first-order reasoning techniques, notably (conditional) rewriting, as well as extensions including mathematical induction and a "functional instantiation" capability for mimicking second-order reasoning. Additionally, one can engage in meta-reasoning — using ACL2 to reason, and prove theorems, about ACL2's logic from within ACL2. One can then use these theorems to augment ACL2's proof engine with custom extensions. ACL2 also supports forms of meta-level control of its rewriter. Relatively recent additions of these forms of control, as well as extensions to ACL2's long-standing meta-reasoning capability, allow a greater range of rules to be written than was possible before, allowing one to specify more comprehensive proof strategies.

1 Introduction

ACL2 is a logic, a programming language, and a semi-automatic theorem prover [6, 4, 5]. This paper is about the meta reasoning facilities of ACL2, the theorem prover. We give a brief overview of ACL2's operations, paying particular attention to ACL2's rule-based rewriter, which is generally considered to be its main proof procedure. We then present a sequence of increasingly complex problems that cannot be solved with normal rewrite rules and show how they can be solved using ACL2's meta-reasoning facilities. Although occasionally simplified, all but one of these examples are based upon items from actual proof efforts. One can find these as well as many additional examples in the proof scripts distributed with ACL2 by searching for the keywords syntaxp, meta, and bind-free.

The facilities presented in this paper correspond to "computational reflection" as described by Harrison [3]. We do not, however, wish to argue with his thesis that there is often a "too-easy acceptance of reflection principles as a practical necessity." Rather, we argue that these facilities, when carefully integrated into a system such as ACL2, can greatly enhance the user experience. Although ACL2 does have a tactic programming language — for its interactive utility (the so-called *proof-checker*) — experience has shown that the techniques described here are often simpler and more profitable to use.

The facilities described in this paper fall into three categories: meta functions (dating back to [1]), syntaxp directives, and bind-free directives. In addition, each of these three facilities can be divided into a "plain" and an "extended" version. The plain version of meta functions has been present in ACL2 from its inception, and syntaxp was added not long thereafter (in the early 1990s). Extended meta functions were added in Version 2.6 (2001). The bind-free directive was added to Version 2.7 (2002), in both its plain and extended versions. The extended syntaxp directive was added at the same time.

Syntaxp and bind-free might be appropriate within an LCF-style system such as HOL [2]. We do not intend to argue that point vigorously, although it seems that they could serve to free users from the need to work in the tactic meta-language.[1] We do believe that these facilities bring many of the benefits of HOL's programmability to ACL2. Additionally, even experienced ACL2 users may find it simpler to add a syntaxp or bind-free directive than to prove a *meta rule* that installs a new simplification procedure.

We begin with a few words about the ACL2 language. The ACL2 language is based upon a subset of Common Lisp. As such, it uses a prefix notation. For example, one might write an expression `3*f(x,y+3)` in a traditional notation, or in the C programming language, which would be written in Lisp notation as `(* 3 (f x (+ y 3)))`. We should also mention that ACL2 terms are themselves objects, which therefore can be constructed and analyzed by ACL2. Without this ability, the features we describe in this paper would not have been possible. In this paper, however, we stick with a traditional (or C language) notation for pedagogical purposes.

We will also use function names that are self-explanatory. For example, the ACL2 term `(quotep x)`, which we could write as `quotep(x)`, would be written in this paper as `constant(x)` (so that we need not explain that `quotep` is the recognizer for constant terms). We will also avoid Lisp's quote notation by writing, for example, `fn-symb(x) == +` to indicate that the top function symbol of the term `x` is the symbol `+`, in place of the ACL2 notation `(equal (fn-symb x) '+)`. We therefore assume no familiarity with Lisp on the part of the reader, yet with the comments above we also expect it to be readable by those familiar with Lisp or ACL2.

We now give our brief overview of ACL2. The user submits a purported theorem to ACL2, which applies a series of procedures in an attempt to prove the theorem. These procedures include, among others, the following: simplification that includes rewriting and linear arithmetic; generalization; and induction. ACL2 is fully automatic in the sense that this process, once started, cannot be further guided.

That said, however, ACL2 will rarely succeed at proving any but the simplest of theorems on its own. The user usually must assist ACL2 in finding a proof, generally by attaching hints to a theorem or by proving additional rules. New

[1] We thank a referee for pointing out that Isabelle's Isar user interface is an example of a "recent trend in some higher order logic theorem provers to shield users from having to learn the tactic meta-language."

rules will be added to ACL2's database and used in subsequent proof attempts. The judicious development of a library of rules can make ACL2 not only powerful but — in the words of a game review — strangely glee to play with.

ACL2's rewriter uses conditional rewrite rules and proceeds in a left-to-right, inside-out manner, maintaining a context as it goes. This will be clearer after an example. When ACL2 is rewriting a goal of the form:

```
    <hyp 1> &&
    <hyp 2>
==> <from> == <to>
```

it acts as if it were rewriting the equivalent clause (i.e., disjunction, represented here using ||)

```
not(<hyp 1>) || not(<hyp 2>) || (<from> == <to>)
```

and attempts to rewrite, in (left-to-right) order, each of the following to true: `not(<hyp 1>)`, `not(<hyp 2>)`, and `(<from> == <to>)`. As ACL2 rewrites each disjunct above, it does so in a context in which it assumes the falsity of the others. Thus, when ACL2 rewrites `not(<hyp 1>)`, it assumes both `<hyp 2>` and `(<from> != <to>)`. Similarly, when it rewrites `(<from> == <to>)`, it assumes rewritten forms of `<hyp 1>` and `<hyp 2>`. These assumptions are the context.

Suppose ACL2 is rewriting a function application, say `foo(<arg1>, <arg2>)`. In this case, ACL2 will recursively rewrite (proceeding left to right) each of `<arg1>` and `<arg2>`, yielding `<arg1'>` and `<arg2'>` and then rewrite the expression `foo(<arg1'>, <arg2'>)`. Note that this inside-out rewriting order mimics that for evaluation — a function's arguments are evaluated before the function is applied to them[2].

Let us now examine the rewriter in more detail, and see how this last expression, `foo(<arg1'>, <arg2'>)`, may be rewritten. Assume that the conditional rewrite rule

```
GIVEN p(x)
      q(y)
REWRITE foo(x, y) TO bar(y, x)
```

had been previously proved. The left-hand side of the above conclusion — `foo(x, y)` — can be matched with our target term — `foo(<arg1'>, <arg2'>)` — by replacing the variables `x` with `<arg1'>` and `y` with `<arg2'>`. Instantiating the above theorem thus yields:

[2] And as for evaluation, the rewriter handles if-then-else terms in a "lazy" manner: in order to rewrite the term

```
if <test> then <true-branch> else <false-branch>
```

ACL2 first rewrites the test, and if the result is true or false then ACL2 rewrites only the true branch or the false branch, respectively. Otherwise the resulting `if` term will generally lead, ultimately, to a case split.

```
GIVEN p(<arg1'>)
      q(<arg2'>)
REWRITE foo(<arg1'>, <arg2'>) TO bar(<arg2'>, <arg1'>).
```

If ACL2 can relieve the hypotheses — recursively rewrite them to true — it will replace the expression `foo(<arg1'>, <arg2'>)` with `bar(<arg2'>, <arg1'>)`.

Conditional rewriting, as illustrated above, is quite restrictive. This paper presents techniques that allow a much greater range of replacements to be specified — they allow one to specify solutions to classes of problems and to experiment speculatively with several rewriting strategies, selecting among these based upon the predetermined outcome of these strategies. Note, in particular, that we are not claiming that these facilities allow us to prove things we could not, in principle, before. Rather, we developed these facilities to help the user to prove theorems more easily and naturally, by removing much of the tedium of repeatedly carrying out simple and "obvious" steps.

2 Syntaxp

When reasoning about arithmetic expressions, it is usual to have some rules like the following to assist with normalizing sums:

```
RULE: associativity-of-+
REWRITE (x + y) + z TO x + (y + z)

RULE: commutativity-of-+
REWRITE y + x TO x + y

RULE: commutativity-2-of-+
REWRITE y + ( x + z) TO x + (y + z)
```

Although it may appear that the second and third rules could each loop or be applied repeatedly, they permute the individual summands into a pre-defined term-order.[3] Rules that merely permute their elements without introducing any new function symbols, such as the aforementioned two rules, are recognized by ACL2 as potentially looping. It will apply such rules only when doing so will move a smaller term to the left of a larger one. Thus, for instance, ACL2 will use `commutativity-of-+` to rewrite `x + 3`, `y + x`, and `(y + z) + x` to `3 + x`, `x + y`, and `x + (y + z)` respectively, but will not apply it to any of these latter expressions.

Note that although there was no meta-level reasoning used to justify these rules, and although there were no meta-level heuristics explicitly given by the user, the behavior of `commutativity-of-+` and `commutativity-2-of-+` are restricted based upon the syntactic form of the instantiations of the variables `x`, `y`, and `z`.

[3] The details of this term-order are irrelevant to the present paper; but, crudely, it is a lexicographic order based upon the number of variables, the number of function symbols, and an alphabetic order.

Let us now consider the term `x + (3 + 4)`. Recall that ACL2 rewrites inside-out. Thus, ACL2 will first rewrite the two arguments, `x` and `(3 + 4)`. The first of these is a variable, and so rewrites to itself. The second of these is a ground term and, since ACL2 implements an executable logic, this term will get evaluated to produce `7`. Finally, ACL2 will use `commutativity-of-+` to rewrite `x + 7` to `7 + x`.

But what about `3 + (4 + x)`? Ideally, this would rewrite to the same thing, but there is nothing the above rules can do with this. If we could only get the `3` and `4` together, things would proceed as for `x + (3 + 4)`. The following rule will do this for us:

```
RULE: fold-consts-in-+
GIVEN syntaxp(constant(c))
      syntaxp(constant(d))
REWRITE c + (d + x) TO (c + d) + x.
```

This rule is just the reverse of `associativity-of-+`, with the addition of two syntaxp hypotheses. Without these extra hypotheses, this rule would loop with `associativity-of-+`.

How do these syntaxp hypotheses work? Logically, a syntaxp expression evaluates to true. The above rule is, therefore, logically equivalent to

```
GIVEN t
      t
REWRITE c + (d + x) TO (c + d) + x
```

or

```
REWRITE c + (d + x) TO (c + d) + x
```

and this is the meaning of `syntaxp` when one is proving the correctness of a rule. (Note that `t` denotes true.)

However, when attempting to apply such a rule, the test inside the syntaxp expression is treated as a meta-level statement about the proposed instantiation of the rule's variables, and that instantiated statement must evaluate to true to establish the syntaxp hypothesis. Note, in particular, that the statement must *evaluate* to true, rather than be proved true as for a regular hypothesis. Thus, just as term-order is automatically used as a syntactic restriction on the operation of `commutativity-of-+` and `commutativity-2-of-+`, so we have placed a syntactic restriction on the behavior of `fold-consts-in-+` — the variables `c` and `d` must be matched with constants.

Here, we are considering the application of `fold-consts-in-+` to the term `3 + (4 + x)`. The variable `c` of the rule is matched with `3`, `d` with `4`, and `x` with `x`. Since `3` and `4` are, indeed, constants, `3 + (4 + x)` will be rewritten to `(3 + 4) + x`. This last term will then be rewritten in an inside-out manner, with the final result being the desired `7 + x`.

We have thus used `syntaxp` to assist in specifying a strategy for simplifying sums involving constants. `Fold-consts-in-+` merely places the constants into a

position in which ACL2 can finish the job using pre-existing abilities — in this case evaluation of constant sums. Without such a rule we would have had to write a rule such as:

```
RULE: crock
REWRITE 3 + (4 + x) TO 7 + x
```

for each combination of constants encountered in the proof. Without `syntaxp` the necessity for rules such as `crock` would make ACL2 much more tedious to use.

Although the example presented here uses a very simple syntactic test in the syntaxp hypothesis, this need not be the case in general. There are ACL2 functions, not presented here, to deconstruct a term and examine the resulting pieces. Although rare, quite sophisticated syntaxp hypotheses are possible.

Before concluding this section, we wish to emphasize an important fact about syntaxp hypotheses that is easily overlooked. As mentioned above, a syntaxp hypothesis is logically true, and is treated as such during the verification of the rule containing it. Consider the rule

```
RULE: example
GIVEN integer(x)
REWRITE f(x, y) TO g(y, x)
```

in which the `integer(x)` hypothesis is required for the rule to be correct. We would not, then, be able to prove:

```
RULE: synp-example-bad
GIVEN syntaxp(x == 0)
REWRITE f(x, y) TO g(y, x)
```

Even though we, as users, know that the syntaxp hypothesis requires `x` to be the constant `0` (which is an integer), ACL2 does not get to use this fact during the proof of `synp-example-bad`. Rather we must use:

```
RULE: synp-example-good
GIVEN syntaxp(x == 0)
      integer(x)
REWRITE f(x, y) TO g(y, x)
```

ACL2 must impose this seemingly arbitrary restriction in order to maintain logical soundness. Recall that, logically speaking, syntaxp always returns true; hence the hypothesis (integerp x) does not follow logically from the syntaxp hypothesis.

3 Meta functions

In the previous section we discussed certain aspects of the process of normalizing sums, and saw how syntaxp hypotheses can be used to achieve a greater degree

of control than was possible without them. They allowed us to specify a rule based upon the ability to analyze the lexical structure of an ACL2 expression.

In this section we present facilities not only for examining a term, but also for constructing a new term, via so-called *meta rules*. These were first implemented in Nqthm, ACL2's predecessor; see [1], and we refer the reader to that paper, or the "Essay on Correctness of Meta Reasoning" in the ACL2 source code for a careful description. In a nutshell, meta rules install user-defined simplification code into the rewriter, where the user's proof obligation for those rules guarantees that each application of that code returns a term provably equal to its input. Here, we review the basics of meta rules before describing their extension in Section 4. More details may also be found in the extensive documentation distributed with ACL2, specifically within the topic "meta".

Consider the following example: arrange that for an equality between two sums, cancel any addends that are common to both sides of the equality. For instance, `x + 3*y + z == a + b + y` should be simplified to:

```
x + 2*y + z == a + b.
```

If one knew ahead of time the maximum number of addends that could appear in a sum, one could write (a large number of) rules to handle all the potential permutations in which common addends could appear; but this will not work in general and is potentially expensive in terms both of the user's labor to develop the set of rules and of ACL2's labor in sorting through such a number of rules, any particular one of which is unlikely to be needed.

Instead, we will use a *meta function*. A meta function is a custom piece of code that transforms certain terms into equivalent ones. When this transformation is proved to be correct via a *meta rule*, the meta function will be used to extend the operations of ACL2's simplifier.

Here is pseudo-code for our meta function, which cancels common summands from both sides of an equality.

```
FUNCTION: cancel-plus-equal(term)
1  if (fn-symb(term) == EQUAL
2       && fn-symb(arg1(term)) == +
3       && fn-symb(arg2(term)) == +) then
4     {lhs = sum-fringe(arg1(term))
5      rhs = sum-fringe(arg2(term))
6      int = intersect(lhs, rhs)
7      if non-empty(int) then
8         make-term(sum-tree(diff(lhs, int)),
9                   <,
10                  sum-tree(diff(rhs, int)))
11        else term}
12    else term
```

And here is the associated meta rule.

```
RULE: cancel-plus-equal-correct
```

```
META-REWRITE term TO cancel-plus-equal(term).
```

Unlike the `syntaxp` example, which merely performed a simple textual test on a term, `cancel-plus-equal-correct` takes a term and constructs an equivalent one under programmatic control.

We now examine its action on:

```
3 + f(x) + g(x, y) == x + g(x, y) + h(y, x).
```

The test of the if expression, lines 1–3, ask whether `term` is an equality between two sums. If `term` were not, `cancel-plus-equal` would return it unchanged — line 12. By returning a term unchanged, a meta function signals lack of applicability, i.e., failure. But since in the present case `term` is bound to `3 + f(x) + g(x, y) == x + g(x, y) + h(y, x)` and so is such an equality, ACL2 will execute the true-branch of the if expression in lines 4–11. Lines 4 and 5 assign to the variables `lhs` and `rhs` lists of the addends of the left-hand side and right-hand side respectively. In line 6, the intersection of these two lists is assigned to `int`. If this intersection is empty, ACL2 evaluates line 11 and returns `term` unchanged, signaling lack of applicability. Since in our case the addend `g(x, y)` is common to both sides of the equality, `int` is non-empty and so ACL2 constructs two new sums and a new equality in lines 8–10:

```
3 + f(x) == x + h(y, x).
```

The addends of these sums are the (bag-wise) difference of the two lists of addends, `lhs` and `rhs` — `{3, f(x), g(x, y)}` and `{x, g(x, y), h(y, x)}`, with the list `int` — `{g(x, y)}`.

Thus, meta rules allow one to write a custom simplifier for entire classes of terms, rather than having to write rules for a myriad of subclasses. We are able to do so because an ACL2 term is merely a structure consisting of a function symbol and the function's arguments, and we can deconstruct, examine, and reconstruct such structures using ACL2 functions.

We now briefly examine the "meaning" of a meta rule and touch upon how to prove its correctness. A meta rule not only has an associated meta function, but also has an associated evaluator that operates in an environment. An evaluator is a function that can evaluate terms by first looking in the environment for the values of any variables present in the term and then evaluating the resulting ground term. The meta rule then states that, using this evaluator, the evaluation of a manipulated term in the environment is equal to the evaluation of the original term in the same environment.

4 Extended Meta-functions

Sometime prior to ACL2 Version 2.6, one of the authors of this paper became dissatisfied with some of the limitations inherent in meta functions. In particular, he wanted to write a rule similar to `cancel-plus-equal-correct` that would cancel like factors from either side of an inequality, but was unable to do so. The

difficulty stemmed from the fact that within ACL2's logic, as opposed to standard mathematics, complex numbers are linearly ordered using the dictionary order on their real and imaginary parts respectively. Thus, for example, `0 < i` and `i < 1`. It is therefore *not* true that for numbers `x`, `y`, and `z`:

```
    0 < x
==> x*y < x*z == y < z.
```

For a counterexample, let `x` and `y` be `i`, and `z` be `0`. In contrast with a high school mathematics exam, within ACL2, one can perform such a simplification only if one also knows that `x` is rational[4]. The correct theorem in ACL2 is

```
    rational(x) &&
    0 < x
==> x*y < x*z == y < z.
```

In this section, we describe an extension to meta functions that allows us to perform such simplifications. This extension will allow us to gather information, for heuristic purposes only, that would not be otherwise available.

A "plain" meta function takes one argument — the term under consideration. This is what we saw in the previous section. An "extended" meta function takes two additional arguments, `mfc`[5] and `state`. These extra arguments give one access to functions that can be used for heuristic purposes, with names of the form `mfc-xxx`. These functions allow one to access and examine ACL2's internal data structures as well as giving one the ability to call a couple of the major functions of ACL2's rewriter.

We will see below how to make use of the following function.

```
FUNCTION: provably-pos-rat(x, mfc, state)
  mfc-rw(make-term(make-term(RATIONAL, x),
                  &&,
                  make-term(0, <, x))
         t t mfc state)
```

It asks whether ACL2 can rewrite an expression of the form `rational(x) && 0 < x` to true. We wish to emphasize here that the ability to construct and examine ACL2 terms within ACL2's logic is fundamental to such capabilities.

Here is pseudocode for our meta function, which cancels a common positive rational factor (if any) from both sides of an inequality.

```
FUNCTION: cancel-times-<(term, mfc, state)
1  if (fn-symb(term) == <
2      && fn-symb(arg1(term)) == *
3      && fn-symb(arg2(term)) == *) then
```

[4] There are no irrational numbers in ACL2.

[5] Mfc stands for "Meta Function Context." The meta function context is a large and complex data structure that contains the current dynamic environment of ACL2's rewriter.

```
4     {lhs = product-fringe (arg1(term))
5      rhs = product-fringe (arg2(term))
6      int = intersect(lhs, rhs)
7      pos-rat = find-pos-rat(int, mfc, state)
8      if non-empty(pos-rat) then
9         make-term(IF,
10                  make-term(make-term(RATIONAL, pos-rat),
11                            &&
12                            make-term(0, <, pos-rat)),
13                  make-term(product-tree(diff(lhs, pos-rat)),
14                            <,
15                            product-tree(diff(rhs, pos-rat))),
16                  term)
17        else term}
18     else term)
```

And here is the associated meta rule.

```
RULE: cancel-times-<-correct
META-REWRITE term TO cancel-times-<(term).
```

The function above is similar to `cancel-plus-equal`, but with three distinctions. First, in lines 2 and 3 `cancel-times-<` tests for the presence of products rather than sums and in lines 13 and 15 produces new products rather than sums. Second, line 7 is new. `Find-pos-rat` takes three arguments — `int` (the list of common factors), `mfc`, and `state`. `Find-pos-rat` steps through the elements of `int`, searching for one for which `provably-pos-rat(element, mfc, state)` returns true. If it is able to find one, `find-pos-rat` returns a list containing that positive, rational factor. If it is unable to find one, it returns the empty list.

Third, in lines 9–16 `cancel-times-<` constructs a more complex return value than just a simple equality between two sums or products. We must do so because, although we as users know that if `pos-rat` is non-empty it must contain a positive rational, ACL2 does not know this logically. Just as any information gathered by a syntaxp hypothesis cannot be used during verification of the rule with that hypothesis, so any information gathered by an `mfc-xxx` function is not available to ACL2 during a meta rules verification. ACL2 has no knowledge about the `mfc-xxx` functions, other than that they are functions.

We now examine the action of this rule in more detail, using

```
3 * f(x) * g(x, y) < x * g(x, y) * h(y, x)
```

as our example. We assume that in the present context, `g(x, y)` is provably a positive rational. Things will proceed much as in the previous example and, as described immediately above, `find-pos-rat` will return a list containing the single term `g(x, y)`, and this value will be assigned to the variable `pos-rat`. The test in line 8 is therefore true, and ACL2 will evaluate lines 9–16. The result is

```
if (rational(g(x, y)) && 0 < g(x, y))
   then 3 * f(x) < x * h(y, x)
   else 3 * f(x) * g(x, y) < x * g(x, y) * h(y, x).
```

During subsequent simplification of this expression, ACL2 will first rewrite the test of the `if` in order to determine which branch to use. Since (by construction) the test will rewrite to true, ACL2 will rewrite only the "then" sub-term, leading to the desired final result:

```
3 * f(x) < x * h(y, x).
```

Although this "extra" rewriting of the if expression's test might seem to be a source of inefficiency, in practice we have not found this to be true. Failure is the norm and the vast majority of the time any particular rule does not apply to the current term. Thus, only a small percentage of the work is ever duplicated, and this only when progress is (supposedly) being made.

5 Bind-free

The careful reader may have noticed that the meta rule, `cancel-plus-equal-correct`, presented in Section 3 would not actually simplify the first, motivating, example — the addends `3 * y` and `y` are not equal, and so would not be found by merely taking the intersection of the two sets of addends. While this could be fixed by using something more sophisticated than `intersection` to determine what to subtract from both sides, we instead present a solution using a `bind-free` hypothesis.

Bind-free hypotheses grew out of a discussion between several of this paper's authors dissatisfied with the difficulty of proving simple meta rules correct. In general, this proof burden is equivalent to proving the total correctness of a piece of software. Although theoretically meta rules, plain or extended, are much more powerful than a rule using bind-free, this extra power is rarely needed. Giving up this extra power, when it is not needed, can make it much easier to write and verify the appropriate rules. This exchange, in turn, encourages one to focus upon the larger picture by proving the most general rules possible and thereby helps one to avoid getting lost in the details.

A bind-free hypothesis is similar to a syntaxp hypothesis in that its treatment when verifying the rule in which it appears differs from its treatment when that rule is being applied to a term during rewriting. Both bind-free and syntaxp hypotheses are treated as being logically true during verification of a rule, and both are evaluated during the rules application. As before this differing treatment is sound, and for the same reasons.[6]

A bind-free hypothesis differs from a syntaxp hypothesis as follows. A syntaxp hypothesis evaluates to true or false, signaling success or failure. A bind-free hypothesis either evaluates to the empty list or signals success by returning a

[6] ACL2 actually implements both bind-free and syntaxp using a single primitive, synp, an implementation detail that is beyond the scope of this paper.

list of pairs binding the free variables of the rule, as illustrated by the following example.

```
FUNCTION: find-matching-addends (lhs rhs)
1  if (fn-symb(lhs) == +
2      && fn-symb(rhs) == +) then
3      {common-addends = find-common(sum-fringe(lhs),
4                                    sum-fringe(rhs))
5       if common-addends then
6          list(pair(x, common-addends))
7          else empty-list}
8      else empty-list

RULE: simplify-equality-of-sums
GIVEN rational(rhs)
      rational(lhs)
      bind-free(find-matching-addends(lhs, rhs))
REWRITE lhs == rhs TO lhs - x == rhs - x.
```

Note that it is the job of other rules, not shown here, to simplify the resulting differences.

Briefly, this rule cancels any common addends by adding their inverse to both sides of the equality. There are two things to note about this rule. First, note that the variable `x` does not appear in the left-hand side of the concluding equality of `simplify-equality-of-sums`. It is, therefore, a *free variable*. (As briefly described in the Introduction, ACL2 matches the left-hand side of a rule's concluding equality with the term currently being rewritten, binding any variables to their matching sub-terms. Since `x` does not appear in the left-hand side of `simplify-equality-of-sums`'s conclusion, it is unbound or *free*. We will make this more explicit shortly.) ACL2 has several automatic mechanisms for choosing an instantiation of such variables, which we do not discuss here. Rather, we describe how we use `bind-free` to programmatically determine the appropriate binding.

Second, the correctness of this rule does not depend upon the value that we subtract from both sides. We are free to pick this value however we want.

How is this rule applied to the following equality?

```
x + 3*y + z == a + b + y
```

As hinted in the Introduction, when ACL2 attempts to apply the rule `simplify-equality-of-sums` to the term under discussion, it first forms a substitution that instantiates the left-hand side of the rule's concluding equality so that it is identical to the target term. This substitution has the following value in our example.

```
((lhs == x + 3*y + z)
 (rhs == a + b + y))
```

ACL2 then attempts to relieve the hypotheses in the order they were given. Here, the first two hypotheses are regular ones, to be relieved by standard rewriting. Let us assume that in the current context these hypotheses rewrite to true; we examine the final, bind-free, hypothesis.

ACL2 evaluates `find-matching-addends(lhs, rhs)` in an environment in which `lhs` and `rhs` are instantiated as determined by the substitution. In this case we evaluate

```
find-matching-addends(x + 3*y + z, a + b + y).
```

The test of the `if` expression (lines 1 and 2 above) asks whether `lhs` and `rhs` are sums. If they weren't the expression would evaluate to the empty list in line 8, signaling failure or lack of applicability. Since they are sums, ACL2 evaluates the true branch of `simplify-equality-of-sums` in lines 3 – 7. Lines 3 and 4 assign to the variable `common-addends` a list of addends common to both `lhs` and `rhs`. `Find-common` is a much more complex function than `intersection` and examines the addends in a more intelligent manner. We do not further describe this than to say that in the present case, `find-common` returns `y`, the "matching" part of `lhs` and `rhs`. The returned value of `find-matching-addends` is, therefore, `list(pair(x, y))`, informally written as `(x == y)`, and this is then used to extend the substitution:

```
((lhs == x + 3*y + z)
 (rhs == a + b + y)
 (x == y)).
```

This is used to substitute back into the `T0` side of `simplify-equality-of-sums`'s concluding rewrite, yielding the result:

```
x + 3*y + z - y == a + b + y - y.
```

Again, we have preemptively eliminated the need for a large collection of similar rules with one rule.

This rule both was able to search for matching addends in a more sophisticated manner than in `cancel-plus-equal-correct` and was easier to prove. The authors of a meta rule might be reluctant to use such a complex search method, because it could greatly complicate the proof of correctness and the simpler method was usually "good enough." A well-constructed bind-free rule, however, is often trivial to prove.

6 Extended Syntaxp

We now return to syntaxp hypotheses, but in an extended form. Just as meta rules come in two flavors, so do syntaxp hypotheses. In this section, we describe two more `mfc-xxx` functions and show how they can be used with extended syntaxp hypotheses.

(Recall that a syntaxp hypothesis is treated as being logically true when it is one of the hypotheses of the rule being proven correct. It is only during a

rule's use, when the hypothesis must be relieved, that ACL2 will execute these functions. As before, any information gathered is of heuristic use only — it cannot be used to justify the correctness of a rule.)

- `mfc-clause(mfc)`: returns the current goal being proved. From the distributed `finite-set-theory` books[7] we take the following example:

  ```
  function: rewriting-conc-lit(term, mfc, state)
  subterm-of(term, last(mfc-clause(mfc)))
  ```

  This function asks whether the term now being rewritten is the conclusion of the current goal. It has been found useful for certain expensive rules to act only upon the conclusion of a goal. The heuristic thought behind this is that often the hypotheses of a goal merely set forth the conditions under which the conclusion is true. It is therefore reasonable to expend more effort rewriting a conclusion than a hypothesis. See the `finite-set-theory/osets` books distributed with ACL2 for examples of this.
- `mfc-ancestors(mfc)`: returns the current list of the negations of the backchaining hypotheses being pursued. The only use we envision for this function is in:

  ```
  function: rewriting-goal-literal(term, mfc, state)
  is-empty-list(mfc-ancestors(mfc))
  ```

  (Note that `term` and `state` are being ignored here.) This function asks whether we are rewriting a term from the current goal — as opposed to rewriting a hypothesis from a rewrite (or other) rule. This has been found useful in such rules as the following.

  ```
  rule: floor-positive
  given: syntaxp(rewriting-goal-literal(x, mfc, state))
         rational(x)
         rational(y)
  rewrite: 0 < floor(x, y)
  to: (0 < y && y <= x) || (y < 0 && x <= y)
  ```

  By using `rewriting-goal-literal` we avoid the expense of inducing a case-split for the two disjuncts while backchaining to relieve a rule's hypotheses (when it is unlikely to do any good).

7 Extended Bind-free

Bind-free hypotheses also come in an extended form. In this section we illustrate such hypotheses by presenting a rule for simplifying terms of the form `integer(<sum>)` where `<sum>` is a sum. For example, if we can show that `y` is an integer, we would like to simplify `integer(x + y + z)` to `integer(x + z)`.

[7] *Books* are ACL2 input files that can be run through ACL2's *certification* process.

```
FUNCTION: reduce-integer-+-fn(sum, mfc, state)
1  if fn-symb(sum) != + then
2     empty-list
3     else if provably-integer(arg1(sum), mfc, state) then
4          list(pair(int, arg1(sum)))
5     else if fn-symb(arg2(sum)) == + then
6          reduce-integer-+-fn(arg2(sum), mfc, state)
7     else if provably-integer(arg2(sum), mfc, state) then
8          list(pair(int, arg2(sum)))
9     else empty-list

RULE: reduce-integer-+
GIVEN bind-free(reduce-integer-+-fn(sum, mfc, state))
      integer(int)
REWRITE integer(sum) TO integer(sum - int)
```

We emphasize here that although we (as users) know that the second hypothesis, `integer(int)` must be true by the way that we selected `int`, this information is not available to ACL2. ACL2 must rederive this fact for its own use. Again, although this might seem a source of inefficiency, in practice we have not found this to be true.

We now consider our example, `integer(x + y + z)`, where `y` is provably an integer, and describe the action of `reduce-integer-+-fn` on this term. First note that addition is actually a binary operation in ACL2; we are really looking at the term `integer(x + (y + z))`. `Reduce-integer-+-fn` recurs on the addends so since `y`, by assumption, is provably an integer, `reduce-integer-+-fn` returns `list(pair(int, y))`. The right-hand side of the concluding equality of `reduce-integer-+` is therefore rewritten to the appropriate instance of the term `integer((x + y + z) - y)`, which will be simplified by other rules to yield our desired result.

8 Conclusion

In this paper we have described three facilities afforded by ACL2 for varying levels of meta level control and reasoning. The weakest of these, syntaxp hypotheses, allow one to control the behavior of ordinary rewrite rules by restricting their operations based upon the syntactic form of their instantiated variables. A quick search reveals that there are over 800 uses of syntaxp hypotheses among more than 150 of the proof scripts distributed with ACL2. Bind-free hypotheses, which allow one to programmatically select a binding for free variables, are slightly more powerful and there are more than 50 uses of this relatively new facility in approximately 15 scripts. Finally, there are about 25 uses of meta rules in 12 scripts[8].

[8] It seems likely that all but a couple of these meta rules could, instead, be made bind-free rules. For example, the meta rule `cancel-plus-equal-correct` could be

The initial designs of these facilities sprang from users' frustrations with the effort required to carry out certain proofs which required one to continually point out to ACL2 simple and seemingly "obvious" steps. The initial implementations were then tested, generalized, and tested again. They were not placed into the main development branch of ACL2's source code until all were satisfied that they were correctly and efficiently implemented, as easy to use as possible, and sufficiently general to be broadly useful.

As we have seen, these meta level facilities allow one to specify solutions to entire classes of problems, avoiding the need for a myriad of rules in their stead. This allows one to concentrate on the structure of the desired proof, and to leave many of the details to ACL2. See [7] for an example of this in the domain of bags, or multisets.

Acknowledgements

We thank the referees, who provided significant useful feedback (after obvious careful reading). We also thank the ACL2 community for useful discussions.

References

1. R. S. Boyer and J Moore. Metafunctions: proving them correct and using them efficiently as proof procedures. The Correctness Problem in Computer Science, R. S. Boyer and J Moore, editors. Accedemic Press, 1981.
2. M. J. C. Gordon and T. F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, 1993.
3. J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique.
4. M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, 2000.
5. M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, 2000.
6. M. Kaufmann and J Moore. ACL2: An Industrial Strength Version of Nqthm. Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96), pp. 23-34, IEEE Computer Society Press, June 1996.
7. Eric Smith, Serita Nelesen, David Greve, Matthew Wilding, and Raymond Richards. An ACL2 Library for Bags (MultiSets). Fifth International ACL2 workshop, 2004. `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/index.html`
8. Markus Wenzel. Isar — a Generic Interpretive Approach to Readable Formal Proof Documents. Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, LNCS 1690, Springer, 1999.

---

replaced with the more general bind-free rule `simplify-equality-of-sums`. That this is not the case is due to the fact that bind-free rules were not available at the time of many of these meta rules' creation.