The Milawa Rewriter
and an ACL2 Proof of its Soundness

Jared Davis*
The University of Texas at Austin
Department of Computer Sciences
1 University Station C0500
Austin, TX 78712-0233, USA
(`jared@cs.utexas.edu`)

October 5, 2007

Abstract. Rewriting with lemmas is a central strategy in interactive theorem provers. We describe the Milawa rewriter, which makes use of assumptions, calculation, and conditional rewrite rules to simplify the terms of a first-order logic. We explain how we have developed an ACL2 proof showing the rewriter is sound, and how this proof can accommodate our rewriter's many useful features such as free-variable matching, ancestors checking, syntatic restrictions, caching, and forcing.

Keywords: ACL2, Milawa, theorem proving, verified rewriter

1. Introduction

We are developing a new automated theorem prover called *Milawa*. Our program follows the ACL2 [10] tradition: it is an interactive tool rather than a fully-automatic decision procedure, and it can be used to prove properties of purely-functional Lisp programs. Other discrete systems, ranging from circuits [9] to Java programs [11], can be analyzed using Lisp models. A major goal in our work is for Milawa to be a mechanically-verified theorem prover. Pursuing this goal, we have used ACL2 to prove Milawa's rewriter is sound.

Rewriting with lemmas is the driving force in this style of theorem proving. When a user begins working on a new theorem, she typically provides a bit of guidance—"induct *this* way", or "consider *these* cases"—then turns the problem over to the rewriter. The rewriter uses libraries of reusable rules (each of which has been proven earlier) to simplify the resulting cases. Some cases will be proven outright, and she can inspect the rest to decide which additional lemmas or hints are needed. Over time, her library of lemmas becomes a potent strategy for reasoning in her problem domain.

* This work was supported by NSF Grant CNS-0429591.

Our rewriter makes use of an assumptions system for keeping track of what is known, an evaluator for simplifying ground terms, and conditional rewrite rules whose hypotheses are relieved via backchaining (goal-directed, recursive rewriting). Our rewriter has many useful features: backchaining loops are prevented using a heuristic "ancestors check" which is adapted from ACL2's rewriter; when free variables are encountered in a rule's hypotheses, the assumptions system identifies potential matches; rule application can be syntactically restricted; results are cached so we do not repeatedly rewrite a commonly-occurring subterm or hypothesis; and hypotheses can be "forced" so they do not have to be established through rewriting alone.

Our rewriter is not as powerful as ACL2's, which makes use of a type reasoning system, an arithmetic procedure [8], metafunctions [7], and congruence rules for generalized equivalence relations. But neither is it a toy. We are now using it to verify some extended proof techniques for Milawa, and this effort currently includes some two thousand lemmas. Many of our rewriter's features (free variable matching, caching, forcing, etc.) were implemented to make the rewriter powerful enough to support this effort.

What does it mean when we say our rewriter is sound? Among other inputs, our rewriter takes x, a term to rewrite; *assms*, the assumptions to use; and $\equiv$, the equivalence relation to preserve (equality or Boolean equivalence). It returns a new term, x', which is a hopefully-simplified replacement for x. We say our rewriter is sound if, for any choices of x, *assms*, and $\equiv$, it is possible to prove $\mathit{assms} \rightarrow x \equiv x'$ in the Milawa logic. This statement mentions provability, so it is a metatheorem from the perspective of the Milawa logic. Our approach is to prove it using ACL2 as a metalogic: we define a Milawa-logic proof checker as an ACL2 function, and implement our rewriter as another ACL2 function; from ACL2's perspective, our soundness claim is just a regular theorem about these two Lisp programs, and there is nothing "meta" about it.

To structure our proof we introduce *traces*, which record at a high level how terms were rewritten. For example, one step in a trace might say, "The subterm y was rewritten to y' by applying the rewrite rule r, whose hypotheses were relieved as recorded by these subtraces." Traces allow us to break our soundness proof into two phases: first we show that any valid trace can be compiled into a Milawa proof which establishes $\mathit{assms} \rightarrow x \equiv x'$, then we show our rewriter always produces a valid trace. This decoupling has allowed us to implement many of our rewriter's features without substantially altering its proof of soundness. Also, much of our proof should be directly reusable when implementing new rewriters that employ different strategies (e.g., outside-in vs. inside-out rewriting).

In this paper, we first explain how we can introduce Milawa-logic provability as a concept in ACL2 (Sec. 2). We then discuss the assumptions system (Sec. 3) and the evaluator (Sec. 4) which are used by our rewriter. Next, we describe traces (Sec. 5) and explain how the rewriter builds them (Sec. 6). After that, we mention several of our rewriter's features (Sec. 7) and cover how they are useful, how they are implemented, and how they affect our soundness proof. Lastly, we remark upon some related work (Sec. 8) and conclude (Sec. 9).

2. Defining Provability

The Milawa logic can be used to reason about simple Lisp programs. The objects are the natural numbers, symbols and (recursively) ordered pairs. Structures like lists, maps, aggregates, and trees are represented by layering pairs. Computation is expressed with terminating, recursive functions defined atop Lisp primitives like *if*, *equal*, *cons*, *car*, and +. The terms of the logic are constants, variables, function applications, and *let*-like lambda abbreviations. The atomic formulas are equalities between terms, and compound formulas are formed with negation and disjunction. There are some rules of inference such as instantiation and cut, and the behaviors of primitive Lisp functions are described with axioms such as $car(cons(x, y)) = x$.

To develop a proof checker for Milawa in ACL2, we introduce a simple Lisp-like encoding of the Milawa terms. We extend this encoding to include formulas, and represent Milawa proofs as trees of proof steps. Each proof step includes a *method* and *conclusion*, and perhaps some *subproofs* and *extras*. The *method* names the inference rule being used, e.g., "cut", while the *conclusion* is the formula proved by this step. The *subproofs* are needed to justify non-axiom steps, and the *extras* are sometimes needed to provide additional information, e.g., instantiation steps require a substitution list.

Our proof checker is an ACL2 function which inspects all the steps in a proof for validity. As an example of what this entails, Milawa's instantiation rule allows us to prove A/σ given a proof of A (where A is understood to be a formula, σ is a substitution list, and A/σ denotes the result of applying σ to A). So, a proof step of the "instantiation" method is valid only if:

- its subproofs contain a single proof, call its conclusion A;
- its extras are a substitution list, call it σ; and
- its conclusion is the formula A/σ.

We call our proof checker *Proofp*, which is short for "proof predicate" in the Lisp tradition. After implementing our proof checker, we can define Milawa-logic provability as an ACL2 concept: the formula ϕ is provable if there exists an object, p, that *Proofp* accepts and whose conclusion is ϕ.

3. The Assumptions System

Our rewriter takes part in a larger strategy for proving theorems in Milawa: to prove a new formula, we first coerce it into conjunctive normal form and try to prove each of the resulting clauses. To prove a clause, $L_1 \vee \ldots \vee L_n$, we try to show that at least one L_i is true. To show L_i is true, we first rewrite it to L_i', then see if the truth of L_i' is evident[1].

Since our goal is to show at least some literal is true, we can freely assume the other literals are false as we rewrite L_i. These assumptions are quite important. For example, suppose our clause is $\neg x \vee x$. As we rewrite the first literal, $\neg x$, we can assume the second literal, x, is false. Then $\neg x$ rewrites to $\neg \mathit{false}$, which we can see is true. If we had not considered the other literals, we would not have been able to simplify x or $\neg x$ to prove the clause.

The simplest way to record our assumptions would be to keep them in a list. Then, each time we were rewriting a subterm, we could consult our list to see if this subterm was known to be *true* or *false*. But this approach would not be very powerful. For example, suppose we knew the *subsetp* function was reflexive, expressed as the rewrite rule,

$$\mathit{subsetp}(x, x) \mapsto \mathit{true},$$

which denotes "terms matching $\mathit{subsetp}(x, x)$ may be rewritten to *true*." Further suppose we were trying to rewrite $\mathit{subsetp}(t_1, t_2)$ after assuming that $t_1 = t_2$ was true. Since the subterm "$t_1 = t_2$" does not literally occur anywhere within the term we are rewriting, we would not use our assumption and our rule would not match. Instead, we would like to canonicalize t_2 to t_1 so that $\mathit{subsetp}(t_1, t_2)$ can be rewritten to $\mathit{subsetp}(t_1, t_1)$, which our rule can match.

To address this, we store assumptions in an *assumptions structure.* These structures still keep a list of the terms we have assumed, but also store databases of inferred facts. For example, when we assume

[1] Our literals actually have the form $\mathit{term} = \mathit{false}$. So, for example, if L_i rewrites to $\mathit{false} = \mathit{false}$, we know it is true; if it rewrites to $5 = \mathit{false}$, we know it is false; if it rewrites to $x = \mathit{false}$ or $\mathit{foo}(x) = \mathit{false}$, its truth is not evident.

$t_1 = t_2$, a disjoint-set structure for equalities is updated. Now, when the rewriter asks the assumptions system if it can simplify t_2, our find operation will canonicalize t_2 to t_1. If we additionally infer $t_2 = t_3$, the structure is updated with an appropriate union so that both t_2 and t_3 may be canonicalized to t_1. A similar structure is used to record and propagate Boolean equivalences.

Since the rewriter can use assumptions to simplify a term, the soundness of the rewriter depends upon the soundness of the assumptions system. This is not trivial, but in this paper we want to focus on the rewriter's soundness proof instead. In brief, our tables are composed of *assumption traces* which are similar to the rewriter traces which will be described in Sec. 5. Each trace includes enough information to see which assumptions justify its conclusion, and we have defined a compiler which can translate these traces into a *Proofp*-acceptable proof of $\mathit{assms} \rightarrow x \equiv x'$, where *assms* are the terms we have explicitly assumed and $x \equiv x'$ is the fact we have inferred. Finally, as in our proof of the rewriter, we show that our assumption-making code adds only valid traces to our tables, so everything we have inferred can be justified using our compiler.

With enough work it would be possible to add other kinds of forward-direction reasoning, e.g., a type reasoning system, or a database of inequalities for an arithmetic procedure. We have not yet needed these kinds of features in the proofs we are working on, so we have not implemented them. It is tempting to think, "we should be inferring more," since a more intelligent assumptions system may be able to lend greater assistance to the rewriter. But maintaining and consulting tables of inferences does take time, and we do not want to impede the rewriter by inferring lots of facts that it will never use.

As a final note, the clause is not the only source of assumptions. Our rewriter may encounter conditional expressions of the form,

$$\mathit{if}(x, y, z) = \begin{cases} y & \text{if } x \text{ is true, or} \\ z & \text{otherwise.} \end{cases}$$

Here, we can locally assume x is true as we recursively rewrite y, and similarly we can assume x is false as we rewrite z. Other typical control-flow operations, like *and* and *or*, are abbreviations defined in terms of *if*. So, for example, when we are rewriting $\mathit{or}(x, y)$, we can assume x is false while rewriting y.

4. The Evaluator

In addition to the assumptions system, our rewriter is supported by an evaluator which can simplify ground terms. Our evaluator is similar to McCarthy's [12] *apply* function. As inputs, it takes a ground term to evaluate, a database of function definitions, and a stack-depth counter to ensure termination. It produces an encoded constant when evaluation is successful, or *false*[2] if an error such as a stack overflow occurs. These errors are propagated, e.g., if our attempt to evaluate a_i has produced *false*, then $f(a_1, \ldots, a_n)$ will also evaluate to *false*.

The evaluator has cases for constants, variables, function applications, and lambda abbreviations. Constants always evaluate to themselves, while variables always fail to evaluate since only ground terms (i.e., variable-free terms) can be evaluated. Arguments to most functions are evaluated eagerly, but like Lisp we lazily evaluate the arguments to *if* (hence also *and* and *or*). Without this laziness, recursive functions like

$$\mathit{length}(x) = \mathit{if}(\mathit{consp}(x), 1 + \mathit{length}(\mathit{cdr}(x)), 0)$$

would cause a stack overflow.

Our rewriter makes use of evaluation, so before we can prove our rewriter is sound, we must first be able to justify the evaluator's operation in the Milawa logic. We have proven, in ACL2:

> Suppose all the *defs* are axioms.
> Suppose $\mathit{eval}(x, \mathit{defs}, \mathit{depth})$ successfully evaluates to x'.
> Then $x = x'$ is provable in the Milawa logic.

To do this, we developed an "evaluator builder" function, which takes the same arguments as the evaluator, and builds a *Proofp*-checkable proof of $x = x'$. The constant case is easy to justify: since constants always evaluate to themselves, the conclusion is $x = x$, which is provable since equality is reflexive. The variable case is also trivial: since evaluating a variable is an error, the "$\mathit{eval}(x, \mathit{defs}, \mathit{depth})$ successfully evaluates" hypothesis is false, and the conjecture is vacuously true. But how can our compiler justify the use of function definitions? Suppose we have recursively evaluated the arguments to a function, and have recursively built proofs of $a_1 = a_1{}'$, etc. Our first step is to use the following axiom schema, which holds since our language is purely functional:

[2] This is unambiguous since *false* does not represent any term in our encoding.

$$\frac{\begin{array}{l} a_1 = a_1{}' \\ \ldots \\ a_n = a_n{}' \end{array}}{f(a_1, \ldots, a_n) = f(a_1{}', \ldots, a_n{}')}$$

Now we turn our attention to simplifying $f(a_1{}', \ldots, a_n{}')$. At this point, all the $a_i{}'$ are constants. The Milawa logic includes a small set of *base functions* such as *consp*, *cons*, *car*, $+$, and $-$, and provides an axiom schema which allows us to prove these functions, applied to constants, produce the appropriate value. For example, $\mathit{cons}(1, 2) = \langle 1, 2 \rangle$ is provable in one step by base evaluation. But we also need to justify the evaluation of defined functions. Here, we begin with the axiom for the function's definition, e.g., $f(x_1, \ldots, x_n) = \mathit{body}$. We instantiate this with the substitution $\sigma = [x_1 \leftarrow a_1{}', \ldots, x_n \leftarrow a_n{}']$, producing $f(a_1{}', \ldots, a_n{}') = \mathit{body}/\sigma$. Since *body* may only mention the variables $x_1, \ldots, x_n$, body/σ is a ground term, and we recursively evaluate it to obtain our final result; this recursion is well-founded because we decrease the *depth* parameter. In the end, we have shown:

$$f(a_1, \ldots, a_n) = f(a_1{}', \ldots, a_n{}') = \mathit{body}/\sigma = \mathit{result}.$$

The case for lambda abbreviations is similar. We show the actuals can be simplified in place, then perform beta reduction and recursively evaluate the result.

5. Rewriter Traces

Until now, we have suggested our rewriter takes an input term, x, and returns the simplified term, x'. Actually, it returns a *trace* explaining how x was rewritten, and x' is only one component of this trace. A rewriter trace is a tree of steps. Each step is an aggregate which includes a *method* explaining what kind of step it is, the *assms* structure it used, an *lhs* and *rhs* (the input and output terms, respectively), the $\equiv$ being maintained, some *subtraces* if necessary, and any *extras* needed to justify the step. Each trace stands for the formula,

$$[\mathit{assms} \rightarrow]\ \mathit{lhs} \equiv \mathit{rhs},$$

where the brackets indicate *assms* might be empty.

We currently support the fifteen types of trace steps presented in Figures 1 and 2. These traces have two important attributes. First, we can compile them into Milawa-logic proofs which can be accepted by

Proofp. Second, they form a convenient basis for rewriting. For example, our rewriter's handling of *if* (x, y, z) can probably be inferred by looking at the different ways we can build traces for *if*-expressions.

Compiling trace steps can be tedious. Even for something as simple as transitivity, we have four cases—there may or may not be *assms*, and the equivalence may be = or *iff*. In each case, our compiler needs to be able to build a *Proofp*-checkable proof of $x \equiv z$. These proofs can be difficult to build since the Milawa logic, like most logics, provides only very basic rules of inference. For example, "Modus Ponens" is not a one of our rules, and emulating it takes us 5 basic steps when *assms* are empty, or 14 basic steps when they are non-empty.

For each kind of trace step, we write a *step compiler* to build proofs of the step's conclusion from proofs of its premises. For example, the transitivity-step compiler can build a proof of $[\mathit{assms} \rightarrow]\ x \equiv z$ from proofs of $[\mathit{assms} \rightarrow]\ x \equiv y$ and $[\mathit{assms} \rightarrow]\ y \equiv z$. We combine these step compilers into a whole-trace compiler, which recurs through the trace and applies the appropriate step compiler on each step. This is the only place where we need to consider all the different kinds of traces, and it is easy to verify our whole-trace compiler using the lemmas we proved for each step compiler. A nice feature of this design is that new kinds of traces can be added without having to change and re-verify the existing step compilers: only the proof of the whole-trace compiler needs to be updated, and this update is trivial once the soundness of the new step has been established.

6. The Rewriter

The primary inputs of our rewriter are the term to rewrite, the assumptions structure being used, and the equivalence relation to maintain (equality or Boolean equivalence). Other inputs include the rewrite rules and function definitions to use, counters to ensure termination, and various control parameters like whether to beta-reduce λ-abbreviations.

The rewriter has base cases for constants and variables, and recursive cases for *if*-expressions, *not*-expressions, other function applications, and λ-abbreviations. It operates in an inside-out fashion, so to rewrite $f(a_1, \ldots, a_n)$ we first rewrite $a_1, \ldots, a_n$ to $a_1', \ldots, a_n'$, then try to address $f(a_1', \ldots, a_n')$. This process is iterated until a fixed point or an artificial limit is reached.

As examples of how the rewriter builds its traces, we will now consider one base case (constants) and one recursive case (functions other than *if* and *not*) in detail. We will ignore considerations like caching,

Failure

$$\frac{}{[assms \rightarrow]\ x \equiv x}$$

If, false case

$$\frac{\begin{array}{l}[assms \rightarrow]\ x_1\ \mathit{iff}\ \mathit{false}\\ [assms \rightarrow]\ z_1 \equiv z_2\end{array}}{[assms \rightarrow]\ \mathit{if}(x_1, y_1, z_1) \equiv z_2}$$

If, true case

$$\frac{\begin{array}{l}[assms \rightarrow]\ x_1\ \mathit{iff}\ \mathit{true}\\ [assms \rightarrow]\ y_1 \equiv y_2\end{array}}{[assms \rightarrow]\ \mathit{if}(x_1, y_1, z_1) \equiv y_2}$$

Not congruence

$$\frac{[assms \rightarrow]\ x\ \mathit{iff}\ x'}{[assms \rightarrow]\ \mathit{not}(x) \equiv \mathit{not}(x')}$$

Equiv by args

$$\frac{\begin{array}{l}[assms \rightarrow]\ a_1 = a_1{}'\\ \ldots\\ [assms \rightarrow]\ a_n = a_n{}'\end{array}}{[assms \rightarrow]\ f(a_1, \ldots, a_n) \equiv f(a_1{}', \ldots, a_n{}')}$$

Transitivity

$$\frac{\begin{array}{l}[assms \rightarrow]\ x \equiv y\\ [assms \rightarrow]\ y \equiv z\end{array}}{[assms \rightarrow]\ x \equiv z}$$

If, same case

$$\frac{\begin{array}{l}[assms \rightarrow]\ x_1\ \mathit{iff}\ x_2\\ x_2, assms \rightarrow y \equiv w\\ \neg x_2, assms \rightarrow z \equiv w\end{array}}{[assms \rightarrow]\ \mathit{if}(x_1, y, z) \equiv w}$$

If, general case

$$\frac{\begin{array}{l}[assms \rightarrow]\ x_1\ \mathit{iff}\ x_2\\ x_2, assms \rightarrow y_1 \equiv y_2\\ \neg x_2, assms \rightarrow z_1 \equiv z_2\end{array}}{[assms \rightarrow]\ \mathit{if}(x_1, y_1, z_1) \equiv \mathit{if}(x_2, y_2, z_2)}$$

If-not normalization

$$\frac{}{[assms \rightarrow]\ \mathit{if}(x, \mathit{false}, \mathit{true}) \equiv \mathit{not}(x)}$$

Lambda equiv by args

$$\frac{\begin{array}{l}[assms \rightarrow]\ a_1 = a_1{}'\\ \ldots\\ [assms \rightarrow]\ a_n = a_n{}'\end{array}}{[assms \rightarrow]\ (\lambda x_1 \ldots x_n\ .\ \beta)\ a_1 \ldots a_n \equiv (\lambda x_1 \ldots x_n\ .\ \beta)\ a_1{}' \ldots a_n{}'}$$

Beta reduction

$$\frac{}{[assms \rightarrow]\ (\lambda x_1 \ldots x_n\ .\ \beta)\ a_1 \ldots a_n \equiv \beta/[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n]}$$

Figure 1. Basic Rewrite Traces

Ground evaluation
(Where *lhs* evaluates to *rhs*)

$$\frac{}{[\mathit{assms} \rightarrow]\ \mathit{lhs} \equiv \mathit{rhs}}$$

Assumptions
(Justified by an assumption)

$$\frac{}{[\mathit{assms} \rightarrow]\ \mathit{lhs} \equiv \mathit{rhs}}$$

Rule application
(Justified by a rewrite rule)

$$\frac{\begin{array}{l}[\mathit{assms} \rightarrow]\ \mathit{hyp}_1\ \mathit{iff}\ \mathit{true} \\ \ldots \\ [\mathit{assms} \rightarrow]\ \mathit{hyp}_n\ \mathit{iff}\ \mathit{true}\end{array}}{[\mathit{assms} \rightarrow]\ \mathit{lhs} \equiv \mathit{rhs}}$$

Forcing
(Must be justified later)

$$\frac{}{[\mathit{assms} \rightarrow]\ \mathit{lhs}\ \mathrm{iff}\ \mathit{true}}$$

Figure 2. Advanced Rewrite Traces

ancestors checking, backchain limits, and free-variable matching in this discussion. For a constant, c:

- If we are maintaining equality, we cannot simplify c and we generate a *Failure* trace concluding $c = c$.
- Otherwise we are maintaining Boolean equivalence. In Milawa (and analogous to Lisp) all constants other than *false* are considered to be *true*. If our input is *true* or *false*, we cannot simplify it and we return a *Failure* trace. Otherwise, we generate an *Evaluation* trace which handles the canonicalization and concludes c *iff true*.

For functions other than *if* and *not*, say $f(a_1, \ldots, a_n)$:

- We recursively rewrite each a_i to a_i', maintaining equality. We combine these traces into an *Equiv by args* trace which establishes $f(a_1, \ldots, a_n) \equiv f(a_1', \ldots, a_n')$. This is the beginning of an "evolving" trace which we try to strengthen as described below.
- We first try evaluation. If each a_i' is a constant, then $f(a_1', \ldots, a_n')$ is a ground term and we try to evaluate it. If this is successful, we build an *Evaluation* trace showing $f(a_1', \ldots, a_n') \equiv \mathit{result}$. We combine this with our *Equiv by args* trace using *Transitivity* to conclude $f(a_1, \ldots, a_n) \equiv \mathit{result}$. Since *Evaluation* traces ensure their *result* is canonical, we return early without trying to strengthen this trace further.
- Otherwise we try rewriting with rules. We can efficiently look up the rules about the function f, and we try to match each of these against $f(a_1', \ldots, a_n')$. If we find a matching rule, we instantiate

each of its hypotheses with the unifying substitution and try to rewrite them all to *true*. If this is successful, we can build a *Rule application* trace showing $f(a_1{}', \ldots, a_n{}') \equiv \mathit{result}$, and use *Transitivity* to conclude $f(a_1, \ldots, a_n) \equiv \mathit{result}$. We are not sure the *result* is canonical, so we continue.

– We now try assumptions. If progress is possible, we use *Transitivity* to extend the evolving trace with an *Assumptions* trace.

– Finally, we decide if another pass is needed. If we have applied rules or assumptions, more simplification might be possible, so we recursively rewrite our most-reduced term and extend our evolving trace with the result.

There are many details here, but for the soundness proof we only need to establish that each time we build a trace, it is done according to one of the rules laid out in Figures 1 and 2. Since we sometimes build traces using recursively-constructed subtraces, this proof is by induction and follows the recursive structure of our rewriter.

7. Rewriter Features

We now take a quick tour of our rewriter's features, and how their implementation affects our proof of soundness.

7.1. The "Ancestors Check" Heuristic

To prevent certain kinds of backchaining loops, our rewriter incorporates the "ancestors check" heuristic used by ACL2's rewriter. To see how these loops can occur, consider the following rewrite rule about *listp*, a function for recognizing "proper" lists,

$$[\mathit{listp}(\mathit{cdr}(x))] \Rightarrow \mathit{listp}(x) \mapsto \mathit{true},$$

which denotes "terms matching $\mathit{listp}(x)$ may be rewritten to *true*, provided that the hypothesis, $\mathit{listp}(\mathit{cdr}(x))$, holds." We call this kind of rule a *pump* since it can lead us to consider a sequence of "inflating" terms. For example, if we use our rule to rewrite $\mathit{listp}(t_1)$, we will need to show $\mathit{listp}(\mathit{cdr}(t_1))$. But our rule matches this as well; if we use it again we will need to show $\mathit{listp}(\mathit{cdr}(\mathit{cdr}(t_1)))$, and so on.

This kind of looping tends to be expensive and useless, and we would like to avoid it. We do this by maintaining an *ancestors stack*. Each time we backchain to relieve a new hypothesis, h, we push h onto the stack along with some other information about the rule being used. We do not allow backchaining to occur if the new hypothesis looks heuristically worse than some previously-attempted hypothesis considered to be "on behalf of" this rule.

Since the ancestors check is only used to decide whether to backchain, there is no soundness burden associated with it. We were able to implement the check with only minimal updates to our soundness proof (e.g., adding the extra ancestors stack argument to all calls of the rewriter).

7.2. Free-Variable Matching

A typical example of a rule with free variables is a transitivity rule. For example, the transitivity of *subsetp* is the following rule:

$$[\mathit{subsetp}(x,y), \mathit{subsetp}(y,z)] \Rightarrow \mathit{subsetp}(x,z) \mapsto \mathit{true}.$$

The variable y does not occur in the rule's pattern, $\mathit{subsetp}(x,z)$, so it will not be bound in the substitution created by matching this pattern against terms to rewrite. For example, suppose we have assumed $\mathit{subsetp}(t_1,t_2)$ and $\mathit{subsetp}(t_2,t_3)$ and we are rewriting $\mathit{subsetp}(t_1,t_3)$. When we match $\mathit{subsetp}(x,z)$ against $\mathit{subsetp}(t_1,t_3)$, it only produces the substitution $[x \leftarrow t_1, z \leftarrow t_3]$. If we try to use this substitution directly, we will obtain the unrelievable hypotheses $\mathit{subsetp}(t_1,y)$ and $\mathit{subsetp}(y,t_3)$. We need a mechanism for suggesting useful bindings for y.

Here there is some tension. The rewriter will try to relieve the rule's hypotheses using every binding we suggest. This is potentially expensive, so we want to suggest relatively few bindings. On the other hand, if we fail to suggest a useful binding when there is one, the rule will not be applied and we will fail to make progress. Our approach is fairly conservative. For each free variable, v, we say the first hypothesis that mentions v is *suggestive*, and we try all the bindings which, in a fairly trivial way, can be sure to satisfy all the suggestive hypotheses. To do this, we ask our assumptions system for a list of all the terms which are known to be *true*. We then try to match the suggestive hypotheses against these terms, using the partial substitution as a constraint. At the end of the process, all the bindings we suggest will be sure to satisfy at least the suggestive hypotheses.

In our *subsetp* example, the only suggestive hypothesis is $\mathit{subsetp}(x,y)$, and the known-true terms are $\mathit{subsetp}(t_1,t_2)$ and $\mathit{subsetp}(t_2,t_3)$. We try to match $\mathit{subsetp}(x,y)$ against these terms under the substitution $[x \leftarrow t_1, z \leftarrow t_3]$. The works for the first term, $\mathit{subsetp}(t_1,t_2)$, by binding y to t_2. But it fails for the second term, $\mathit{subsetp}(t_2,t_3)$, since our partial substitution requires x to match with t_1. So in this example, the only binding we suggest for y is t_2.

Adding free-variable matching to our rewriter was initially difficult. In ACL2, loops must be expressed as recursive functions, so each loop in our rewriter ("rewrite each of these arguments", "try each of these rules", "try each of these substitutions", and "relieve each of these hypotheses") has a corresponding function which is part of our mutually-recursive rewriter. We originally only considered a single substitution for each rule, so we did not have the "try each of these substitutions" loop. Adding this to our rewriter required an additional case in all of our inductive proofs. But now that the loop is in place, we could change the bindings we suggest without impacting the soundness proof at all.

7.3. Syntactic Restrictions

Our rewrite rules may include *syntaxp* [7] style constraints on their application. An associative and commutative function like addition provides some good examples of how these restrictions can be useful:

- Without syntactic restrictions, the commutativity rule, $x + y \mapsto y + x$, would cause a loop. We can fix this by requiring y to be a syntactically smaller term than x, which allows us to normalize the operand order.

- Similarly, the commutativity-two rule, $x + (y + z) \mapsto y + (x + z)$, will not loop with itself if we require y to be syntactically smaller than x. Combined with commutativity, this allows us to normalize any right-associated sum.

- Adding the right-associativity rule, $(x + y) + z \mapsto x + (y + z)$, allows us to normalize any sum regardless of how its operands are associated.

- Adding the left-associativity rule, $x + (y + z) \mapsto (x + y) + z$, would loop with right-associativity. But if we require x and y to be constants, this will only aggregate x and y when $(x + y)$ can be evaluated away to a new constant. For example, now we can rewrite $1 + (2 + x) \mapsto (1 + 2) + x \mapsto 3 + x$.

How do we determine if syntactic restrictions are satisfied? Suppose we want to apply a rule with the substitution $[x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n]$. The rule's restriction, R, is a term which may only mention the x_i (i.e., the variables from the rule). Recall that we have an encoding for terms, and let $\ulcorner s_i \urcorner$ represent the constant which encodes s_i. Now $R/[x_1 \leftarrow \ulcorner s_1 \urcorner, \ldots, x_n \leftarrow \ulcorner s_n \urcorner]$ is a ground term, and we say the restriction is satisfied if this term evaluates to *true*. For example, suppose we are trying to apply commutativity, $x+y \mapsto y+x$, to rewrite the term t_1+t_2. The rule matches our term using the substitution $[x \leftarrow t_1, y \leftarrow t_2]$, and the rule's restriction is $y <_{term} x$. Following the procedure above, we evaluate $\ulcorner t_2 \urcorner <_{term} \ulcorner t_1 \urcorner$, and we are allowed to apply the rule only if the result is *true*.

Just as the ancestors check is only used to decide whether to backchain, syntax restrictions are only used to decide whether to try to apply a rule. Hence, supporting syntax restrictions does not appreciably impact our soundness proof.

7.4. Rewriter Caching

Adding free-variable matching to our rewriter caused its performance to degrade, but we were able to correct this by caching the results of rewriting. We implement the cache using Boyer and Hunt's [2] memoization extension to ACL2. We have found that our caching scheme usually provides a hit rate of about 10–30%, with arithmetic-oriented proofs landing on the higher end of the range. The time savings is as high as 50-60% on several of our slowest proofs.

Implicit in the word *caching* is the idea of transparency, i.e., using a cache should not change the results of a computation. But in our rewriter, this is a subtle matter. For example, to ensure our rewriter terminates, we decrement a "backchain limit" counter each time we try to relieve a hypothesis; once the counter is exhausted, backchaining is not permitted. This counter ruins simple attempts to memoize calls of the rewriter, e.g., just because we were able to show $consp(t_1)$ is *true* with a backchain limit of 998 does not necessarily mean we can show it with a backchain limit of 997. We are willing to sacrifice this degree of transparency to obtain a much more effective cache.

The ancestors check requires more delicate handling. Ignoring the ancestors stack would be dangerous: we might "poorly" rewrite x in a context where we have many ancestor restrictions, then reuse this result in a less-restricted context. To prevent this, we develop the notion of an *ancestors-limited* rewrite. The idea is to identify which rewrites may have run afoul of the ancestors check, so that we can avoid caching them.

– An attempt to relieve a hypothesis is ancestors-limited if (1) the ancestors check prevents it from being relieved, or if (2) it rewrote to a non-constant and this rewriting was ancestors-limited.

– An attempt to apply a rule is ancestors-limited if all of the matches we tried have failed, and at least one of them has an ancestors-limited hypothesis.

– An attempt to rewrite a term is ancestors-limited if (1) none of the rules we attempted were successful, (2) at least one of them was ancestors-limited, and (3) other simplification methods such as evaluation and assumptions failed.

There are some limits to what we can cache. Recall that when we encounter an expression of the form $\mathit{if}(x, y, z)$, we assume x is true while rewriting y, and we assume x is false while rewriting z. We did not see a good way to use and extend our input cache while rewriting y and z in this new context, so instead we create two temporary, initially-empty caches—one for y and one for z—which we discard after y and z have been rewritten.

Adding caching to our rewriter took some work. Our rewriter was modified to take the cache as an additional argument, and now returns three values: the trace, an updated cache, and an ancestors-limited flag. We also had to add an invariant to our inductive proofs to ensure that all the traces in the cache are valid, and that only valid traces are added to the cache.

7.5. Forced Hypotheses

Before our rewriter can apply a conditional rewrite rule, it must show that all of the rules' hypotheses are *true*. Sometimes we think a particular hypothesis will always hold. For example, if we encounter $\mathit{factorial}(x)$ in a proof attempt, then we might reasonably expect x to be a natural number instead of a pair or a symbol. Some logics could handle this with a type system, but Milawa functions (like ACL2 functions) are untyped, so this is not an option. Even in a typed logic, it may be difficult to restrict the domains of functions to precisely the sensible inputs. For example, in our work with verifying extended proof techniques, a new technique is often valid only when certain formulas are present in our list of axioms. This would be hard to encode as a type.

When we think a hypothesis should always hold, we add a *force* annotation to it. Ordinarily, if we fail to rewrite a hypothesis to *true*, the rewrite rule will not be applied since we were unable to justify its application. But when we fail to relieve a forced hypothesis, we "pretend" it can be rewritten to *true* anyway. Eventually we have to come up with proofs to justify these pretended steps, but forcing allows us to defer this justification until later.

This deferral is often useful. Upon seeing a forcibly-assumed hypothesis, the user often realizes he has not properly stated his theorem. In fact, he often needs to add the very hypothesis being forced to make his conjecture true. Here the use of forcing provides nice feedback: if the assumption had not been forced, he would first need to identify the rule which did not apply (which might not be easy) and then understand why it did not apply. In other cases, the forcibly-assumed hypothesis may indeed be true, but might not be provable using rewriting alone. Now he has an opportunity to apply other techniques (e.g., induction, generalization, etc.), or to strengthen his rule library as appropriate.

Adding forcing required some updates to our soundness proof. First we extended our traces with *Forcing* steps, which simply record our current assumptions and the term we are assuming to be *true*. Our whole-trace compiler was updated to additionally require, as an input, a list of proofs which establish all of the forced assumptions. Compiling a *Forcing* step involves looking up the appropriate proof from this list. Now, in addition to producing a simplified goal, rewriting a clause also produces goals for all the forced assumptions.

8. Related work

Theorem provers have sometimes been extended with verified proof techniques using *reflection* [5] principles. For example, Smith, et al [17] have developed some custom routines for efficient, lightweight reasoning about multisets in ACL2; Chaieb and Nipkow [3] have written a verified quantifier-elimination procedure for Presburger arithmetic in Isabelle/HOL; and Grégoire and Mahboubi [4] have verified a procedure for reasoning about equality between polynomials in commutative rings in Coq. Our work is not reflective since we use ACL2 as a metalogic, but otherwise it is quite similar to these efforts—we have a complex proof procedure we want to trust, and we gain this trust by proving it is sound with a mechanical theorem prover.

There have also been some non-reflective efforts to mechanically verify theorem-proving algorithms. Shankar [16] used NQTHM to write a proof checker for a first-order logic, and proved the soundness of a tautology checker and some other routines with respect to this proof checker. Ridge and Margetson [14] used Isabelle/HOL to write a first-order theorem prover, and proved the program to be sound and complete. Their program does some proof search, but they mention it is not competitive with other resolution provers. McCune and Shumsky [13] used ACL2 to develop a verifier for proof objects emitted by the resolution prover Otter. No attempt was made to verify Otter itself (a complex C program), but the correctness of their verifier was established with ACL2.

We are not aware of previous efforts to verify rewriters mechanically. Ruiz-Reina et al [15] used ACL2 to investigate properties from the theory of rewriting (confluence, normal forms, etc.), but this work was more about rewriting in the abstract than any particular implementation. In contrast, our work is about a practical rewriter whose features are inspired by ACL2's rewriter.

An alternative to verifying theorem-proving algorithms is to require all proofs to be carried out using only primitive rules of inference. Harrison [5] has a good treatment of how this has been made practical (at least for expressive logics like higher-order logic) using the well-known LCF approach: theorems are represented using an abstract data type and may only be created with constructors corresponding to the primitive inference rules; unverified programs can then be used to build proofs by calling upon these constructors. Since the type system ensures all theorem objects are created in valid ways, the intermediate proof steps can be thrown away for space efficiency. But using only primitive proof constructors may still impose a time penalty, e.g., Chaieb and Nipkow report their verified procedure is some 200 times faster than an equivalent, LCF-style solution. On the other hand, Boulton [1] describes how separating proof search from construction can make LCF-style theorem proving more time efficient, and taken together, our rewriter and trace compiler actually resemble an LCF-style algorithm with this separation.

Finally, some work has been done to verify proof checkers. J. von Wright [18] used HOL to verify that an imperative implementation of a proof-checker accepted only the valid proofs in higher-order logic. Also, Harrison [6] has mimicked the implementation of HOL Light, an OCaml program, as a HOL Light specification to show "something close to the actual *implementation* of HOL" is sound. Verifying proof checkers nicely complements our work: we are willing to trust our proof checker, and are interested in verifying a more sophisticated proof technique.

9. Conclusions

We are now working towards recreating the ACL2 proof of our rewriter's soundness in a *Proofp*-checkable form. If we can do this, we will no longer need to use ACL2 as a metalogic. Instead, we will have a Milawa-logic proof of the Milawa rewriter's soundness. We hope to use this result in a reflective way to extend *Proofp*.

The rewriter itself is useful in this effort. Along with other tools, we are using our rewriter (and our trace compiler) to "translate" the ACL2 lemma libraries we used into *Proofp*-acceptable objects. So far we have translated some two thousand lemmas dealing with topics including arithmetic; list and map utilities; the encoding of terms, formulas, and proofs; substitution operations; our rules of inference; and our proof checker. We still need to translate the lemmas for our assumptions system, evaluator, traces, and the rewriter itself.

For the most difficult lemmas we have translated, the resulting proof object size is as many as one billion conses. Such proofs are near the upper bound of our patience, taking as long as 45 minutes to check on our development machine. Since these relatively simple proofs are so large, and since our rewriter is a complex program with a non-trivial soundness proof, it is difficult to imagine directly translating our ACL2 proof into a *Proofp*-checkable form.

Instead, we are now working to verify intermediate proof checkers to raise our level of abstraction. For example, using *Proofp* we have verified a “level-2” proof checker which extends *Proofp* with some new propositional rules, e.g., Modus Ponens. Proofs written at the second level are significantly smaller than proofs written for *Proofp*. (For one example theorem, the level-2 proof was 84% smaller than the equivalent *Proofp*-level proof, when measured by the number of conses it takes to represent each proof object.) We hope to build a stack of these higher-level proof checkers, each verified with the previous level, culminating in a verified proof checker that can directly use our rewriter.

A current version of our source code may be downloaded from the following web site:

`http://www.cs.utexas.edu/users/jared/milawa/Web/`

Acknowledgements

We would like to thank Matt Kaufmann and J Moore for many useful discussions about rewriting, and Warren Hunt for assistance with memoization.

References

1. Boulton, R. J.: 1993, ‘Efficiency in a fully-expansive theorem prover’. Ph.D. thesis, University of Cambridge.
2. Boyer, R. S. and W. A. Hunt, Jr: 2006, ‘Function Memoization and Unique Object Representation for ACL2 Functions’. In: *ACL2 ’06*.
3. Chaieb, A. and T. Nipkow: 2005, ‘Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic’. In: *Logic Programming, Artificial Intelligence, and Reasoning (LPAR ’05)*, Vol. 3835 of *LNCS*. pp. 367–380.
4. Grégoire, B. and A. Mahboubi: 2005, ‘Proving Equalities in a Commutative Ring Done Right in Coq’. In: J. Hurd and T. Melham (eds.): *Theorem Proving in Higher Order Logics (TPHOLS ’05)*, Vol. 3603 of *LNCS*. pp. 98–113.

5. Harrison, J.: 1995, ‘Metatheory and Reflection in Theorem Proving: A Survey and Critique’. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK.
6. Harrison, J.: 2006, ‘Towards self-verification of HOL Light’. In: U. Furbach and N. Shankar (eds.): *International Joint Conference on Automated Reasoning (IJCAR ’06)*, Vol. 4130 of *LNAI*. pp. 177–191.
7. Hunt, Jr, W. A., M. Kaufmann, R. B. Krug, J. Moore, and E. W. Smith: 2005, ‘Meta Reasoning in ACL2’. In: J. Hurd and T. Melham (eds.): *Theorem Proving in Higher Order Logics (TPHOLS ’05)*, Vol. 3603 of *LNCS*. pp. 163–178.
8. Hunt, Jr, W. A., R. B. Krug, and J. Moore: 2003, ‘Linear and Nonlinear Arithmetic in ACL2’. In: D. Geist (ed.): *Correct Hardware Design and Verification Methods (CHARME ’03)*, Vol. 2860 of *LNCS*. pp. 319–333.
9. Hunt, Jr, W. A. and E. Reeber: 2005, ‘Formalization of the DE2 Language’. In: *Correct Hardware Design and Verification Methods (CHARME ’05)*, Vol. 3725 of *LNCS*. pp. 20–34.
10. Kaufmann, M., P. Manolios, and J. S. Moore: 2000, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.
11. Liu, H. and J. S. Moore: 2004, ‘Java Program Verification via a JVM Deep Embedding in ACL2’. In: K. Slind, A. Bunker, and G. Gopalakrishnan (eds.): *Theorem Proving in Higher Order Logics (TPHOLS ’04)*. pp. 184–200.
12. McCarthy, J.: 1960, ‘Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1’. *Communications of the ACM* **3**(4), 184–195.
13. McCune, W. and O. Shumsky: 2000, ‘Ivy: A Preprocessor and Proof Checker for First-Order Logic’. In: *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, Chapt. 16.
14. Ridge, T. and J. Margetson: 2005, ‘A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic’. In: J. Hurd and T. Melham (eds.): *Theorem Proving in Higher Order Logics (TPHOLS ’05)*, Vol. 3603 of *LNCS*. pp. 294–309.
15. Ruiz-Reina, J.-L., J.-A. Alonso, M.-J. Hidalgo, and F.-J. Martín-Mateos: 2002, ‘Formal proofs about rewriting using ACL2’. *Annals of Mathematics and Artificial Intelligence* **36**(3), 239–262.
16. Shankar, N.: 1994, *Metamathematics, Machines, and Gödel’s Proof*. Cambridge University Press.
17. Smith, E., S. Nelesen, D. Greve, M. Wilding, and R. Richards: 2004, ‘An ACL2 Library for Bags’. In: *ACL2 ’04*.
18. von Wright, J.: 1994, ‘Representing Higher-Order Logic Proofs in HOL’. In: T. F. Melham and J. Camilleri (eds.): *Higher Order Logic Theorem Proving and Its Applications (TPHOLS ’94)*, Vol. 859 of *LNCS*.