Adding advanced rewriting tactics to Coq

Kenneth Roe
Software Engineer
The Johns Hopkins University
Baltimore, MD, USA
kendroe@hotmail.com

Abstract

Having a powerful automatic simplification mechanism is a very important piece of any interactive theorem prover, but Coq's built-in simplifiers `simpl` and `compute` are not very powerful. Here we describe an ml-plugin that significantly improves on these simplifiers. Our library contains customized simplification algorithms for functions with associative and/or communitive libraries, transitive closure, and user-extensible conditional term rewriting.

In addition, several optimizations were added: expressions are interned into a master single DAG, giving improved caching; function symbols and variable names can also be interned; and, natural numbers can be interned as members of the ml `int` type and Coq's unary notation avoided.

Keywords Coq, Theorem proving, Term rewriting, Interactive theorem prover

1 Introduction

Simplification is one of the most important pieces of any interactive theorem prover. The stronger the mechanism, the more usable the tool will be. The speed of a simplification algorithm also plays an important role in the usability of a theorem prover. A user expects simplification to consistently finish within a few seconds. Coq's [The Coq development team 2016] simplification mechanism is limited. The `simpl` tactic, for example, does little more than symbolic computation and becomes quite slow on large expressions. Isabelle provides a more powerful simplification mechanism. Simplification is considered to be one of the main advantages of Isabelle [von Tobias Nipkow 2002] over Coq.

The good news is that Coq provides the ability to add new tactics through an ml-plugin mechanism. Plugins have already been introduced for AC rewriting [Braibant and Pous 2011] and SAT/SMT solving [Armand et al. 2011; et al. 2017].

In this paper, we introduce the advanced rewriting plugin, a plugin for Coq that gives the theorem prover a simplification tactic which is much more powerful than what is available with `simpl` and in many cases what is available in Isabelle. In particular, our system provides better AC matching algorithms and enforcement of recursive path orderings to ensure termination. The tool has many builtin simplifications for common operators such as the arithmetic and boolean operators. There are rules for simplifying quantifier formulae. In addition, the user can add conditional rewrite rules for other operators.

While many of the basics of simplification and term rewriting systems have matured, this system introduces a number of important innovations:

- Recursive path orderings are used to provide a unifying framework to integrate many different simplification algorithms (and conditional rewriting). The general idea is that if an algorithm produces a simplified version of an expression, then the rewrite is accepted, if not, then the algorithm fails. Rewriting terminates when no algorithms or inner rewrites can be applied.
- AC matching and rewriting is deeply embedded into the matching algorithms. Coq expressions are converted to a custom data structure used by the advanced rewriting library. When simplification is done, the expressions are converted back to Coq's representation. Once an operator is marked as AC, in the advanced rewriting library data structure, AC operators are flattened ($a+(b+c)$ becomes $+(a,b,c)$). The pattern matching algorithms then automatically switch the ordering of parameters where necessary.
- Transitive closure computations can be used to simplify expressions with inequalities. For example, our system will simplify $x < y \wedge y < z \wedge z < x$ to `False`.
- A mechanism to intern all subexpressions is implemented. Every expression seen by the system is integrated into a single giant DAG. When a new expression is parsed or generated from rewriting, the intern mechanism adds it to the DAG working up from the leaves. Each subexpression gets a unique intern number which is used to cache useful information about the subexpression.

As a final contribution of the paper, when comparing the data structures used to represent terms in our system to those in Coq, we found three optimizations that can benefit Coq. 1) All subterms should be interned into a single DAG as described above and caching should be implemented. 2) Symbols should be interned (thus replacing string comparisons with integer comparisons). 3) Coq internally represents naturals using a unary representation. This should be replaced with a bignum representation (such as OCaml's `int` type).

Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

2 An overview of rewriting capabilities

In this section, we give a description of each of the rewriting algorithms in our library. Our rewriting system is composed of many smaller algorithms, and each is required to produce a result that is smaller than its input (or, to produce no result). We use recursive path orderings [Dershowitz 1982; Dershowitz and Mitra 1993; Kamin and Levy 1980] to form a partial ordering among expressions. This provides the basis for integrating rewriting algorithms.

The next several sections detail the various algorithms used in our tool.

2.1 Inner rewriting

All of the rules are applied in an innermost fashion. For a term $f(t_1, \ldots, t_n)$, first each of the subterms are simplified yielding $f(t'_1, \ldots, t'_n)$. Then the simplifications are applied to the term as a whole. If the top level term is simplified, then the whole process is repeated. Recursive path orderings ensure that the process terminates.

2.2 Rewrite rules

Rewriting rules/conditional rewriting rules form the basis of our system. Rewrite rules replace one subterm with a simpler one. For example, the rule,

$$x + 0 \rightarrow x$$

is built into the system. We can also have conditional rewrite rules which are of the form $l\{c\} \rightarrow r$. If the term c rewrites to `true`, then the rule can be applied. For example, one might introduce the following conditional rewrite rule

$$2 * x * n > 0\{x > 0\} \rightarrow 2 * n > 0$$

2.2.1 Multiple matches

Sometimes multiple rules will match the the term being simplified. When this is the case, one rule is chosen arbitrarily. The assumption is that all paths will lead to a single simplest state (though the system does not guarantee this). Builtin rules or simplification algorithms will generally be executed first before user algorithms.

2.2.2 Termination

Recursive path orderings are used in ensure termination. The basic idea is that we first assign a partial ordering among function symbols (precedence ordering). Usually more expensive functions have higher precedence. Also, if a function f uses g in its definition, then f has higher precedence than g. In general, a term $t = f(t_1, \ldots, t_n)$ is bigger than a term $u = g(u_1, \ldots, u_n)$ if u is a subterm of t or if $f > g$ and t is greater than some $u_i in \{u_1, \ldots, u_n\}$. One of the interesting properties of the ordering is that it is closed under substitution, if $t > u$, then for any substitution θ, $t\theta > u\theta$. The implication of this is that for any rewrite rule $l\{c\} \rightarrow r$, if $l > r$, then any term x rewritten by the rule to x' will be smaller (ie $x > x'$). We also require $l > c$. We test the condition of the rewrite rule by doing a substitution and then recursively calling our rewrite system to simplify.

Many extensions to the basic concept such as lexicographic orderings [Kamin and Levy 1980] have been developed.

2.2.3 Exponential explosion

In addition to termination, it is also necessary to prevent exponential explosions when rewriting. There is no strictly enforced rule. However, the general rule of thumb is that for any rule $l\{c\} \rightarrow r$, the variable should only appear once in either c or once in r. The reason for this is that a single variable may be replaced with fairly complex expression when applying the rule. Duplicating that expression may require work to be repeated for that subterm.

2.3 Builtins for arithmetic and boolean operators

In addition to rewrite rules, our simplifier contains many builtin algorithms. Many algorithms are just rewrite rules stored in a library such as $x + 0 \rightarrow x$. However, there are also algorithms that are slightly more complex. For example, there are algorithms that can simplify linear equations. For example, $3 * x = 9$ can be simplified to $x = 3$. However, the algorithm needs to check that the 3 and 9 are indeed constants (not really possible with a rule) and that 3 divides 9 evenly.

2.4 Contextual rewriting

Consider simplifying the expression $x = 0 \wedge x + y = 3$. This simplifies to $x = 0 \wedge y = 3$. In order to do this, we need knowledge of the fact that $x = 0$ when simplifying $x + y = 3$. The idea is to add rewrite rules when simplifying $x + y = 3$. To do this, we need to introduce a system of marked variables (denoted with x'). These marked variables in a rewrite rule match a variable directly. For example, x' in l of a rule $l\{c\} \rightarrow r$ matches the variable x and not the variable y or any other variable. Our system will introduce the rule $x' \rightarrow 0$.

2.4.1 Derived rules

The next question is how the system decides on the rule $x' \rightarrow 0$ when it sees the expression $x = 0$. This is done in two steps. First, the rule $x' = 0 \rightarrow$ `true`. This is done by simply marking the variables in the term. The second step is done via the use of generating a derived rule. The following transformation meta-rule is used, $x' = t \rightarrow$ `true` $\Rightarrow x' \rightarrow t$ where t is a constant term such as a number, `true`, `false` or a constructor all of whose parameters are constant terms such as `Cons(1,Nil())`. Our system uses recursive path orderings to orient rules that are generated from equalities. In the case above, we have made a special augmentation to the recursive path orderings to recognize constants as being smaller than variables. As another example, in our system, the expression $f(x) = x \wedge f(f(x)) = x$ will simplify to $f(x) = x$. When

the system simplifies $f(f(x)) = x$, it will do it in the context of $f(x) = x$ because x is smaller than $f(x)$ with respect to the recursive path ordering, the system will introduce the rewrite rule $f(x) \rightarrow x$. Applying this rule twice to the expression $f(f(x)) = x$ will yield $x = x$. This simplifies to `True` and eliminates the term.

2.5 Quantifiers and their rules

Our system contains many specialized rules for simplifying expressions that involve quantifiers. As an example there is a rule to propagate negation inside a universal or existential quantifier, $\exists x.\texttt{not}(t) \rightarrow \texttt{not}(\forall x.t)$. There are also some more complex rules. For example, the expression $\exists x.x = t \wedge e$ can be simplified to $e[t/x]$ when t does not contain the bound variable x.

2.6 Lambda and higher order functions

Our rewrite system supports higher order reasoning by introducing an `apply` operator. a lambda quantifier and the rewrite rule $\texttt{apply}((\lambda x.t), e) \rightarrow t[e/x]$.

2.7 AC functions and pattern matching algorithms

Our rewriting framework provides extensive support for functions with associative and/or commutative properties. First, any expression involving an AC operator is flattened. For example, $x + (y + z)$ is converted to $'+'(x, y, z)$.

Our rule matching algorithm has special match handling for AC properties. If one tries to apply the rule $x + f(y) \rightarrow g(x, y)$ to the term $f(q) + h(r)$, the result will be $g(h(r), q)$. The match algorithm automatically switches the order of the parameters. Now consider what happens if we match the above rule to the expression $f(x) + h(y) + i(r)$. The application can produce three possible results, $g(h(y) + i(r), x)$, $g(h(y), x) + i(r)$ or $g(i(r), x) + h(y)$. The basic rule here is that the variable x in the rule can be matched to one or more terms in the expression. If it is matched to more than one term, then the terms are combined with the AC operator (in this case '+'). The result of our special handling for AC operators is that not only can multiple rules fire on a single expression, but a single rule on its own may produce multiple results. As a simplification to limit the number of matches, if a rule has more than one variable in an AC operator, only one can be matched to multiple terms. For example, when using using the rule $x + y + q(z) \rightarrow qq(x, y, z)$, either the variable x or y can match multiple terms but not both. This means for example, that when simplifying the term $a + b + c + d + q(e)$ with the above rule, we will not generate the result $qq(a + b, c + d, e)$.

One limitation of our implementation is that it does not support neutral elements. For example, if we have a rule $f(x) * 2 + y \rightarrow g(x) + y$, our current implementation cannot simplify the expression $f(x) * 2$. To do this, y would need to be matched to a neutral element 0. This is an extension we are planning in the near future.

2.8 Transitive relations, equality relations and the associated rewriting rules

Our system provides mechanisms to compute transitive closures in simplifying expressions. For example, the expression $a < b \wedge b < c \wedge c < a$ will be simplified to `false`. The transitive closure reasoning is embedded within contextual rewriting. The idea here is that when simplifying $a < b$, there will be two contextual rules $b' < c' \rightarrow \texttt{true}$ and $c' < a' \rightarrow \texttt{true}$. From these two rules, a third rule is produced $b' < a' \rightarrow \texttt{true}$. The transitive closure reasoning then recognizes that having $b' < a' \rightarrow \texttt{true}$ means we can replace $a < b$ with `false`. This gives us $\texttt{false} \wedge b < c \wedge c < a$. By applying other rules, this is simplified to `false`.

When computing transitive closures our system will insert inequalities for constants. For example, if both the constants 3 and 5 are detected within the expression, then the inequality $3 < 5$ will be generated and inserted before computing the transitive closure. This allows us to reduce an expression such as $5 < x \wedge x < y \wedge y < 3$ to `false`.

3 Implementation

In this section we discuss a few of the details of our implementation.

3.1 Interning of symbols and subexpressions

figure 1 shows the data type for expressions. One should note that many of places in which a function or variable name is needed, the type is `int`. This is because all symbols are stored in an intern table. This makes comparisons when matching more efficient.

As a further optimization, all subexpressions are interned. All expressions are organized into a giant dag. Each subexpression is assigned a unique integer identifier. The `REF of int` case of the type in fgure 1 represents the interned version of an expression. As an example, a sub-expression such as `x+y` is assigned a unique intern value (such as 25). Whenever this expression is encountered in computation, the same intern value will be returned. This gives us the ability to cache information related to an expression such as its canonical form. If the expression `x + y - x` is simplified to `y`, then this simplification can be cached so that the computation need not be done a second time.

3.1.1 Efficiently looking up intern values for subexpressions

In order to make interning of subexpressions practical, one needs an efficient algorithm to intern an expression. The good news is that with the exception of when an expression is parsed, most of the time it will only be necessary to intern a top level expression given the interned form of all its subterms. The key to doing this efficiently is that a reverse index needs to be stored. Given a set of subterms (and a functor), the index needs to return the intern number for the parent

expression if it exists. In our system, we simplified this index a bit. Instead of indexing all subterms, we have an index from any subterm to all parents and then we search for the appropriate parent.

3.1.2 Interning of AC subexpressions

Subterms produced with AC functors (such as `+(x,y,z)`) are interned by first sorting the arguments to the AC operator. This causes the expressions `+(x,y,z)` and `+(z,y,x)` to have the same intern value. It also causes the side effect that if the user enters the expression `z+y+x`, it may print out as `x+y+z` as information on the order of the parameters is lost.

3.1.3 Equality test

One interesting result of the interning of subexpressions is that test for equality between subexpressions reduces to testing the equality between two intern numbers.

3.2 Caching of rewrite results

When computing the results of rewriting, whenever a sub term is rewritten, its simplified form is cached. Hence, rewriting only needs to be computed once. This means that the Fibonacci rules shown in figure 2 will exhibit linear complexity despite not being written with memoization.

3.2.1 Contextual rewrite cache

Rewriting may produce different results when done within the context of another expression. For example, simplifying the subterm $x + y$ in the expression $x + y = z \wedge x = 0$ will yield $y = z$ whereas simplifying $x + y = z$ on its own will yield $x + y = z$. When rules are added to the context, to find (or save the cached rewrite), we first generate a term $\texttt{CC}(t, \texttt{rules}(r_1, \ldots, r_n))$ where `rules` is registered as an AC operator. We then save the result of rewriting under the intern number produced for the above expression. For example within the expression above, we generate $\texttt{CC}(x + y, \texttt{rules}(x \rightarrow 0))$. The result of the rewriting (y) is saved under the intern number for this expression.

3.2.2 Issues of interning and decoding expressions

Through out the code for the advanced rewriting library, there are lots of calls to

- `intern_exp`,
- `decode_exp`,
- `decode_one_exp` or
- `decode_two_exp`

to intern or decode an expression (or at least the top one or two levels). When ever an expression needs to be matched for potential rewrites, it needs to be decoded. When a result is produced, the result is interned so that the result can be cached. It turns out there is no one place to cleanly do the interning and decoding. The result of this is the author decided to develop a C implementation of the library which would be cleaner as there would be no need for the interning or decoding. Expressions would be in stored in a giant DAG. The intern numbers are simply the memory addresses for the corresponding records. Pattern matching was done directly on this DAG.

4 The interface to Coq

The interface implements a single tactic `arewrite`. This tactic is similar to `simpl`. The current goal is replaced with one that is simplified. However, since `arewrite` currently cannot construct a proof in Coq's logic for the simplification, there are in fact two goals generated. The first is a simplification of the original goal. The second is a goal to prove that the two are equivalent; this second goal is discussed in more detail in Section 7.1.4 below. `arewrite` is built on top of Coq's `replace` tactic. The goal replacing the original is the simplification generated by our library.

The current tactic is limited to working with only arithmetic operators, boolean operators and the exists and forall quantifiers. This limitation is in the interface between our library and Coq. The actual library is a bit more robust and can handle user defined functions. The library was written many years ago for another project (but the work was never published). One can run our library in a standalone mode and add additional functions. We detail the work needed in section 7.1.

We also have a couple of debugging commands `printExp` and `printAST`[Ringer 2017] which show the AST of a Coq expression and what happens when we convert a Coq expression to an Advanced Rewriting Library expression (without simplifying).

5 Results

While our integration of the rewriting library into Coq is relatively new, we do have some promising initial results that we now detail.

5.1 Comparing Coq, Isabelle and the Advanced Rewriting library on many samples

To compare the capabilities of the various prover rewriting systems, we have produced the table in Figure 3 of specific examples comparing our Advanced Rewriting Library with Coq's built-in simplifier and Isabelle's simplifier.

The first two examples show how the advanced rewriting library can incorporate one operand of a conjunct into context while simplifying the other. For the first sample, the right operand $f(f(x)) = x$ is simplified twice with $f(x) \rightarrow x$ from the left operand to yield $x = x$. This operand is then removed leaving just $x = f(x)$. The second example shows how $f(f(f(x)))$ is simplified with $f(f(x)) \rightarrow x$ to yield $f(x) = x$. This is then used to simplify $f(f(x)) = x$ to $x = x$ and that term is eliminated leaving only $f(x) = x$. The third example shows how the right side of in implicant is rewritten

```
type exp = VAR of int
         | MARKED_VAR of int
         | QUANT of (int * (int * Rtype.etype) list * exp * exp)
         | APPL of (int * exp list)
         | LET of (exp * Rtype.etype * exp *exp)
         | CASE of exp * Rtype.etype * ((exp * exp) list)
         | INDEX of (exp * int * int)
         | HIGHLIGHT of exp
         | NORMAL of exp
         | NUM of int
         | RATIONAL of int * int
         | STRING of string
         | CHAR of char
         | REF of int
         | NOEXP ;;
```

Figure 1. AST for expressions

```
fib(Z)          →  1
fib(S(Z))       →  1
fib S(S(n)))    →  fib(S(n))+fib(n)
```

Figure 2. Rules for Fibonacci numbers

using the left hand side. The result here is that $x + 1 = 4$ is rewritten to $3 + 1 = 4$ which simplifies the whole implicant to `True`. The next three are variants. When $x + 1 = 4$ is on the left of the implicant, it is first simplified to $x = 3$ and then used to simplify the right hand side. The seventh expression in the figure shows how an existential can be eliminated if part of its operand assigns a value to the quantified variable. Examples 8 and 9 show how transitive closures can be used to simplify expressions. Example 10 shows how the advanced rewriting library performs algebraic simplifications. The last example shows how AC ewriting is used. $x + y \rightarrow 0$ is generated from the left hand side expression. It is used to simplify $y + 3 + x$ to $0 + 3$ or just 3.

Based on the experimental results in the table, we conclude that Isabelle has a reasonable system for performing contextual rewriting. It succeeded on the third and seventh test cases. It does not do algebraic simplifications. Test cases 5 and 10 demonstrate this. There is no transitive reasoning in either Coq or Isabelle as demonstrated by cases 8 and 9. Coq's `omega` tactic also failed on these examples. Test case 11 shows that AC matching in Isabelle is limited. We also note that Coq's `simpl` tactic is limited as it failed on all of these test cases.

5.2 Experience with PEDANTIC

The integration is in its early stages and we do not have extensive experience with the use of the system. A simplified version of this rewriting system is implement as part of our PEDANTIC framework for separation logic [Roe and Smith 2017]. The rewriting in PEDANTIC plays a key role in bringing states into a canonical form. However, it does not do all of the work. User intervention is still needed for some of the steps. One of our key motivations in developing this library is to improve the performance of simplification in PEDANTIC. Simplification takes over 90% of PEDANTIC's CPU time. We plan to integrate our rewriting library to improve performance.

Let's look at a sample sequence of rewrites to illustrate how contextual rewriting plays a key role in PEDANTIC. Below is shown a sample a portion of a separation logic expression that may appear in PEDANTIC.

$$TREE(r_1, v_1, 2, [0, 1])*$$
$$TREE(r_2, v_2, 2, [0])*$$
$$(ALL(x \in records(v_2))nth(find(x, v_2), 2) \in records(v_1))$$

For details of PEDANTIC's separation logic, the user is referred to [Roe and Smith 2017]. TREE is a recursive predicate representing a list or tree. In the above, there are two TREE predicates. The first represents a tree each node of which has two elements both of which are pointers to child nodes. The second TREE is a list predicate. Each node has two elements. The first is a pointer to a child element and the second is a data field. The ALL on the third line states that for all elements in the list, the second element points to one of the nodes in the tree. The second parameter of the two TREE predicates, v_1 and v_2 are functional representations. They are a list of list representation of all the nodes, their location in memory and the values of all fields. As an example, v_2 may have the value $[4, [6, [10, 0, 20], 40], 80]$. This value represents a list with three nodes at locations 4, 6 and 8. The second fields of the three nodes are 80, 40 and 20 respectively.

	Expression	Coq's `simpl`	Isabelle	Advanced Rewriting Library
1	$x = f(x) \wedge x = f(f(x))$	$x = f(x) \wedge x = f(f(x))$	fails	$x = f(x)$
2	$f(f(x)) = x \wedge f(f(f(x))) = x$	$f(f(x)) = x \wedge f(f(f(x))) = x$	fails	$f(x) = x$
3	$x = 3 \rightarrow x + 1 = 4$	$x = 3 \rightarrow x + 1 = 4$	solved	`True`
4	$\forall x, x = 3 \rightarrow x + 1 = 4$	$\forall x, x = 3 \rightarrow x + 1 = 4$	fails	`True`
5	$x + 4 = 1 \rightarrow x = 3$	$x + 1 = 4 \rightarrow x = 3$	fails	`True`
6	$\forall x, x + 1 = 4 \rightarrow x = 3$	$\forall x, x + 1 = 4 \rightarrow x = 3$	fails	`True`
7	$\exists x. f(x) = g(4) \wedge x = 3$	$\exists x. f(x) = g(4) \wedge x = 3$	$f(3) = g(4)$	$f(3) = g(4)$
8	$x < y \wedge y < z \wedge z < x$	$x < y \wedge y < z \wedge z < x$ omega fails to solve as well.	fails	`False`
9	$5 < y \wedge y < z \wedge z < 3$	$x < y \wedge y < z \wedge z < x$ omega fails to solve as well.	fails	`False`
10	$\forall x, 2 * x + 1 = 7$	$\forall x, x + (x + 0) + 1 = 7$	fails	$\forall x, x = 3$
11	$x + y = 0 \rightarrow y + 3 + x = q$	$x + y = 0 \rightarrow y + 3 + x = q$	fails	$0 = y + x \rightarrow 3 = q$

Figure 3. Comparison of different rewriting systems on different expressions

One of the operators in PEDANTIC is an unfold operator. We may use it to unfold the second TREE. It may replace our above expression with

$$
\begin{aligned}
&TREE(r_1, v_1, 2, [0, 1])* \\
&r_2 \mapsto r_2' * r_2 + 1 \mapsto vr_2* \\
&TREE(r_2', v_2', 2, [0])* \\
&(ALL(x \in records([r_2, v_2', vr_2])) \\
&\quad nth(find(x, [r_2, v_2', vr_2]), 2) \in v_1)
\end{aligned}
$$

PEDANTIC's simplification will simplify the ALL construct. First, the $x \in records([r_2, v_2', vr_2])$ is simplified to $x = r_2 \vee x \in records(v_2')$. Further simplification of the ALL splits it into two predicates yielding the following separation logic expression:

$$
\begin{aligned}
&TREE(r_1, v_1, 2, [0, 1])* \\
&r_2 \mapsto r_2' * r_2 + 1 \mapsto vr_2* \\
&TREE(r_2', v_2', 2, [0])* \\
&nth(find(r_2, [r_2, v_2', vr_2]), 2) \in v_1* \\
&(ALL(x \in records(v_2')) \\
&\quad nth(find(x, [r_2, v_2', vr_2]), 2) \in v_1)
\end{aligned}
$$

PEDANTIC has a rewrite rule of the form $find(x, [x, ...]) \rightarrow [x, ...]$. Applying this rule to the fourth line of the above yields $nth([r_2, v_2', vr_2], 2)$. Evaluating the nth function yields the separation logic expresion below.

$$
\begin{aligned}
&TREE(r_1, v_1, 2, [0, 1])* \\
&r_2 \mapsto r_2' * r_2 + 1 \mapsto vr_2* \\
&TREE(r_2', v_2', 2, [0])* \\
&vr_2 \in v_1* \\
&(ALL(x \in records(v_2') \\
&\quad nth(find(x, [r_2, v_2', vr_2]), 2) \in v_1)
\end{aligned}
$$

The final tricky step is to reduce $find(x, [r_2, v_2', vr_2])$ to $find(x, v_2')$. The trick is to recognize that x will never be r_2. Several pieces of information from the context need to be used to verify this fact. We need to combine $x \in records(v_2')$ with $TREE(r_2', v_2', 2, [0])$ to verify that x is one of the records in that tree. We then need to combine this with the $r_2 \mapsto r_2'$ predicate to verify that any element in the tree is distinct from r_2. The final result is:

$$
\begin{aligned}
&TREE(r_1, v_1, 2, [0, 1])* \\
&r_2 \mapsto r_2' * r_2 + 1 \mapsto vr_2* \\
&TREE(r_2', v_2', 2, [0])* \\
&vr_2 \in v_1* \\
&(ALL(x \in records(v_2'))nth(find(x, v_2'), 2) \in v_1)
\end{aligned}
$$

Now, the ALL phrase looks very much like the one we had at the beginning except that v_2 has been replaced with v_2'. This allows other operations to fire that match this unfolded tree to the original.

PEDANTIC uses an inner rewrite algorithm. Many of the rewrites are implemented as mini-algorithms. All reductions reduce terms with respect to recursive path orderings. One of our goals is to reimplement the PEDANTIC separation logic routines using our advanced rewriting library.

5.3 The opportunity to substantially improve the performance of Coq

In the process of integrating our rewriting system with Coq, we found that many of our optimizations are not implemented in Coq. Introducing these optimizations will likely both improve the speed and reduce the memory foot print of Coq.

Binary representation of integers The Coq theorem prover uses a unary representation for natural numbers (integers are built up from the constructors 0 and S). Figure 4 shows a Coq session illustrating some of the issues. Large integers will use up large amounts of memory. In fact Coq will generate a stack overflow for any integer larger than about 33000. A binary representation will substantially improve performance.

```
Coq < printAST 100.
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (App (Construct (Name nat) 2) (App (Construct (Name nat) 2)
(App (Construct (Name nat) 2) (Construct (Name nat)
1)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
Coq < Check 1.
1
     : nat

Coq < Check 1000.
1000
     : nat

Coq < Check 10000.
Warning: Stack overflow or segmentation fault happens when working with large
numbers in nat (observed threshold may vary from 5000 to 70000 depending on
your system limits and on the command executed).
10000
     : nat

Coq <
```

Figure 4. Transcript of a session with coqtop that demonstrates how natural numbers are represented in Coq. tt printAST’ comes from the printAST plugin [Ringer 2017].

Interning of symbols and subterms Both of these optimizations produced substantial performance enhancements when implemented in our rewriting package. We suspect there will be similar results when implemented in Coq.

5.4 Improvements to our representation

The Coq theorem prover uses de Bruijn numbers to encode quantified variables. This is not done in our rewriting framework. Localized subexpression contexts are used to convert these indices to variable names. We suspect this will improve the performance of our system could be improved by using de Bruijn encodings.

5.5 Relationship to SMT solving

It is interesting to compare the algorithms in our library to those of an SMT solver. A C implementation of our library was developed (after the ml version) and became the basis of the Heuristic Theorem Prover [Roe 2006]. Researchers have often observed that SMT solvers are substantially faster than Coq in verifying theorems. We suspect one of the issues may be the use of interned subexpressions in most SMT solvers.

6 Related research

Term rewriting theorem provers/program derivation systems

Many of the rewriting concepts come from work done in the Focus program derivation/theorem proving environment [Reddy 1989, 1990]. The Focus system at its heart had a term rewriting system. Their system was exteneded to rewrite expressions with quantifiers. Some of our work on contextual simplification originated from work done in the Focus system. We also based our conditional rewriting system on the conditional rewriting available in Focus. However, Focus never defined exact semantics or path orderings for its quantifiers and many of its rewriting strategies were not fully worked out. The Focus system in turn was built upon program transformation systems developed by Burstall and

Darlington [Burstall and Darlington 1977; Darlington and Burstall 1976].

The Isabelle simplifier

Isabelle is recognized for having a fairly powerful and extensible simplification system. However, our simplifier provides a number of advantages over Isabelle's simplifier. First, recursive path orderings ensure termination. The isabelle [von Tobias Nipkow 2002] manual shows an example where introducing the equation $f(x) = g(f(g(x)))$ could cause the simplifier to loop infinitely. In our system, recursive path orderings will translate this equation into $g(f(g(x))) \rightarrow f(x)$. The rule cannot be used the other way. Recursive path orderings will similarly disallow rules of the form $x + y = y + x$.

In fact, our handling of AC operators does not require any of these types of rules. The idea in our system is that an expression such as $a + (b + c)$ are flattened into $+(a, b, c)$. The rule matcher treats the arguments as a set rather than an ordered list. Hence, no rules are needed to switch around arguments to the AC operators. There is a similar mechanism for commutative operators such as equality. The matching algorithm for rewrite rules will automatically reorder $a = b$ to $b = a$ if necessary. Isabelle introduces a concept of ordered rewriting to partially address AC matching. However, this strategy may require the user to choose an ordering and still is based on the idea of trying to get terms in the right order. Our system is fully automatic.

Isabelle provides a mechanism for introducing tactics to control simplification and introduce case analysis. We have not explored this area and in any case, one could use another mechanism such as LTac in Coq for this purpose. Isabelle also allows tactics to automatically split goals into cases or repeatly apply specialized simplification tactics as part of the simplification process. Again, this can be done using LTac in in Coq.

Other Coq ML plugins

Braibant [Braibant and Pous 2011] developed a Coq plugin for rewriting modulo AC operators. His system uses type classes to identify AC operators. His system does not provide for the more algorithmic inner rewriting or many of the built in algorithms of our system. Their system automatically validates the AC rewrites with respect to the Coq meta-theory. Our system just produces a result. The proof term required to justify the rewrite with respect to Coq's logic is not produced. Instead a second goal is generated that the user needs to prove to validate the result. He does, however, have support for neutral terms. This is something that needs to be added to our system.

SMTCoq [et al. 2017] integrates SMT solving capabilities into Coq. SMTCoq converts a formula in Coq to an SMT solver format. Once an unsat result is produced, the explanation is extracted and converted into a proof using Coq's meta theory. This integration allows formulae to be proven. However, there is no concept of simplification if the expression cannot be completely proven.

7 Conclusion and future work

The most important result of our work is identifying opportunities to improve the performance of the Coq kernel. However, in addition to finding optimizations in our system that can be integrated into Coq, we also found optimizations in Coq that could improve the performance of our system.

Based on our current work, we have identified a number of directions for continuing projects. The first (and most significant) will be to integrate some of our expression optimizations into the Coq kernel. This could substantially speed up the Coq theorem prover. Second, we still need to do work to generate justifications of our rewrites with respect to the Coq meta theory. Third, we still need to complete our integration which is in a primitive state. We could also integrate optimizations found in Coq into our system.

7.1 Work needed to complete Coq interface

We summarize in the section ongoing work to complete the integration of our library into Coq.

7.1.1 More closely modeling Coq's logic

The advanced rewriting library was written many years ago without the intention of connecting it to Coq or any other theorem prover. In fact, the intent was to develop a standalone theorem prover based on rewriting principles. The issue that arises is that many of the constructs in Coq do not have exact correspondences in our library. Two notable issues are 1) there is no support for dependent types and 2) There is no distinction between the uncomputable `Prop` and computable boolean expressions. Our short term solution is to use type inferencing algorithms to figure out the intended mappings when converting expressions back to Coq. Our longer term solution is to make changes to the advanced rewriting library to more closely follow Coq's logic.

Extraction of precedence information from Coq Most precedence information can be extracted automatically. If a function f is used in the definition of a function g, then we add the precedence $f < g$. There will also be a facility to add precedence rules as well as to declare some functors to be lexicographic [Kamin and Levy 1980].

Extraction of rewrite rules from fixpoint/function definitions Many function definitions can be fairly easily be converted into rewrite rules. The main constraint is that the rule must follow recursive path orderings constraints. Otherwise the rule will not be automatically generated. If the definition uses either a `match` or `if-then-else`, then multiple rules will be generated (one for each branch). For match variables will be appropriately instantiated in the rules and

for `if-then-else` conditional rewrite rules will be generated. Here are a couple of examples to illustrate how this will work:

```
fixpoint append(a) =
    match a with
    | nil => b
    | cons f r => cons f (append r b)
    end.
```

will generate:

$$\texttt{append}(\texttt{Nil}, b) \rightarrow b$$
$$\texttt{append}(\texttt{Cons}(f, r), b) \rightarrow \texttt{Cons}(f, \texttt{append}(r, b))$$

and

```
fixpoint fact(a) =
    if n=0
    then 1
    else fact(n-1)*n.
```

will generate:

$$\texttt{fact}(n)\{n = 0\} \rightarrow 1$$

Notice we do not get the second rule

$$\texttt{fact}(n)\{n \neq 0\} \rightarrow n * (\texttt{fact}(n-1))$$

This is because the rule does not follow the constraints of recursive path orderings.

7.1.2 Declaration rewrite rules

In addition to the generated rules, it may be useful for the user to add rewrite rules based on lemmas or that are missed by the automatic rule generation. This will be implemented through the introduction of forceRewriteRule and RewriteRule properties. The latter will introduce the rules even if they violate recursive path constraints. There is a risk that using the latter will cause the simplifier to go into an infinite loop.

7.1.3 Declaring AC, equality, order and partial order operators

The user will be able to add type class definitions to identify AC, equality, total order and partial order operators in a manner similar to [Braibant and Pous 2011].

7.1.4 The need for user validation

One drawback of our system is that the result of rewriting is not validated with respect to Coq's meta theory. Rather, the `arewrite` tactic executes a `replace` tactic which generates a second goal the user needs to prove to validate the result. For now, we do not have the code to automatically prove the second goal. The issue is that one application of our rewrite tactic may not be convertible with respect to Coq's logic and in fact may require the application of several tactics to justify. Many of the rules and algorithms (such as AC rewrites, rewrites based on transitive closure reasoning or contextual rewriting) may not easily correspond to a single Coq tactic.

As an example, our library can rewrite $y + 1 + x$ to 1 using a rule of the form $x + y = 0$. In order to do this simplification without the library in Coq, several tactics would need to be applied. The first several would be applications of associative and commutative rules to put the expression in the right form. This would be followed by a rewrite with $x + y = 0$ and a `simpl` and rewriting with a rule of the form $x + 0 = x$. The advanced rewriting library will need to do these exact same steps to justify a simplification with Coq's logic.

Our rewriting library has a trace facility that can be used to generate a trace of the individual steps in rewriting an expression. This trace can then be used to automatically generate the Coq justification for a rewrite; but at this point such an algorithm is future work.

7.2 Downloading and building

Our rewriting library is compiled with OCaml 4.04.0 and connected to Coq 8.5pl2. The library is available at https://github.com/kendroe/CoqRewriter. The source tree has two Makefiles. There is one makefile that will produce the plugin for Coq. There is another Makefile that will build the rewriting library as a standalone program (with a test suite).

Acknowledgments

The author would like to thank Scott Smith for his feedback on this paper.

References

[n. d.].

Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses.* Springer Berlin Heidelberg, Berlin, Heidelberg, 135–150. https://doi.org/10.1007/978-3-642-25379-9_12

Thomas Braibant and Damien Pous. 2011. *Tactics for Reasoning Modulo AC in Coq.* Springer Berlin Heidelberg, Berlin, Heidelberg, 167–182. https://doi.org/10.1007/978-3-642-25379-9_14

R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *Journal of the Association of Computing Machinery* 24, 1 (1977), 44–67.

J. Darlington and R. Burstall. 1976. A system which automatically improves programs. 6 (1976), 41–60.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. *The Lean Theorem Prover (System Description).* Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Nachum Dershowitz. 1982. Orderings for Term-Rewriting Systems. In *Theor. Comput. Sci. 17.*

Nachum Dershowitz. 1985. Termination. In *RTA.* 180–224.

Nachum Dershowitz and Subrata Mitra. 1993. *Path orderings for termination of associative-commutative rewriting.* Springer Berlin Heidelberg, Berlin, Heidelberg, 168–174. https://doi.org/10.1007/3-540-56393-8_13

Ekici B. et al. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq.

S. Kamin and J.-J. Levy. 1980. *Two generalizations of the recursive path ordering.* Technical Report. University of Illinois, Urbana-Champaign.

June Andronick Toby C. Murray Gerwin Klein Rafal Kolanski Mark Staples, D. Ross Jeffery. 2014. Productivity for proof engineering. In *ESEM*. 15:1–15:4.

The Coq development team. 2016. *The Coq proof assistant reference manual*. https://coq.inria.fr/refman/

G. Sivakumar Nachum Dershowitz, Mitsuhiro Okada. 1987. Confluence of Conditional Rewrite Systems. In *CTRS*. 31–44.

Mitsuhiro Okada Nachum Dershowitz. 1988. Conditional Equational Programming and the Theory of Conditional Term Rewriting. In *FGCS*. 337–346.

Tobias Nipkow. 1989. Equational Reasoning in Isabelle. *Science of Computer Programming* 12 (1989), 123–149.

U. S. Reddy. 1989. Rewriting techniques for program synthesis. *Lecture Notes in Computer Science* 355 (1989), 388–403.

U. S. Reddy. 1990. Formal methods in transformational derivation of programs. In *Proc. ACM International Workshop on Formal Methods in Software Development*. ACM, Napa, CA.

Talia Ringer. 2017. *printAST Coq plugin*. https://github.com/uwplse/CoqAST

Kenneth Roe. 2006. The Heuristic Theorem Prover: Yet Another SMT-Modulo Theorem Prover. In *Int. Conf. on Computer-Aided Verification*. Springer.

Kenneth Roe and Scott F. Smith. 2017. Using the Coq theorem prover to verify complex data structure invariants. In *MEMOCODE 2017*.

Markus Wenzel von Tobias Nipkow, Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science*. Vol. vol. 2283. Springer.