Term Rewriting: Some Experimental Results *

Richard C. Potter and David A. Plaisted

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27514

Abstract

We discuss term rewriting in conjunction with **sprfn**, a Prolog-based theorem prover. Two techniques for theorem proving that utilize term rewriting are presented. We demonstrate their effectiveness by exhibiting the results of our experiments in proving some theorems of von Neumann-Bernays-Gödel set theory. Some outstanding problems associated with term rewriting are also addressed.

Key words and phrases. Theorem proving, term rewriting, set theory.

1 Introduction

Term rewriting is one of the more powerful techniques that can be employed in mechanical theorem proving. Term rewriting allows us to prove fairly sophisticated theorems that are beyond the ability of most resolution-based theorem provers. Unlike resolution, term rewriting seems to duplicate a rule of inference that humans use in constructing proofs. In this paper, we will describe our research and results in proving theorems via term rewriting. The body of theorems we prove are set theoretic; the axiomatization of set theory employed is derived from the work of von Neumann, Bernays, and Gödel. For a list of these axioms, see [2]. The advantage of the von Neumann-Bernays-Gödel formalization is that it allows us to express set theory in first-order logic. This in turn implies that a first-order theorem prover can be used to derive set theoretic theorems. On the other hand, this formalization has a significant disadvantage in that it is very clumsy for humans to use. Second order logic is a much cleaner means for expressing the axioms of set theory.

* This research was supported in part by the National Science Foundation under grant DCR-8516243 and by the Office of Naval Research under grant N00014-86-K-0680.

We begin by introducing **sprfn**, the Prolog-based theorem prover we used in our research; we emphasize the formal deduction system underlying the prover. In the second section we describe the term rewriting mechanism built into **sprfn**. In the third and fourth sections we describe two theorem proving techniques utilizing term rewriting and the results of these approaches when employed in connection with **sprfn**. In each of these two sections we give examples of sample theorems that we were able to derive. We conclude by summarizing our results and addressing some problems that face term rewriting in general as well as some problems specific to term rewriting with **sprfn**.

2 The Simplified Problem Reduction Format

The theorem prover we used -- **sprfn** -- is based on a natural deduction system in first-order logic which is described in [1]. However, before we present this formal system, we would like to motivate it by describing the format on which it is based; namely, the problem reduction format. The formal deduction system implemented by **sprfn** is a refinement of the problem reduction format. Both of them embody the same goal-subgoal structure, as can be seen from what follows. The following description omits many details. For a complete discussion of the problem reduction format, see [5].

The structure of the problem reduction format is as follows. One begins with a conclusion G to be established and a collection of assertions presumed to be true. Assertions are of the form $C{:-}A_1,A_2,\ldots,A_n$ (implication) or P (premises) where A_i, P and C are literals or negations of literals. The implication assertion is understood to mean $A_1 \& A_2 \cdots \& A_n \rightarrow C$. The A_i's are antecedent statements, or simply antecedents, and C is the consequent of the implication. We call the conclusion G the top-goal. The process of attempting to confirm the conclusion begins with a search of the premises to see if one premise matches (is identical with or can be made identical by unification with) the goal G. If a premise P_g matches G then the conclusion is confirmed by P_g. Otherwise, the set of implications whose consequents match G is found. If the antecedent of one implication can be confirmed then one has confirmed the consequent, and hence G, which the consequent matches. Otherwise we consider the antecedents as new subgoals to be confirmed, one implication at a time. These goals are called *subgoals* because none of them is the primary goal. The process of confirming these subgoals involves repeating the method just described in connection with the top-goal.

The natural deduction system underlying **sprfn** -- the modified problem reduction format -- is based on the problem reduction format just described, although refinements are added for the sake of completeness of the deduction system. We do not have room to describe these refinements. The following description of the modified problem reduction format omits many details. For a complete discussion, see[4].

A **clause** is a disjunction of literals. A **Horn-like clause**, converted from a clause, is of the form $L :- L_1, L_2,..., L_n$ where L and the L_i's are literals. L is called the head literal. The L_i's constitute the clause body. A clause is converted to a Horn-like clause as follows. For a given clause containing at least one positive literal, one of its positive literals is chosen as the head literal and all other literals are put in the clause body negated. For an all-negative clause, we use **false** as the head literal and form the body from positive literals corresponding to the original literals.

Now assume S is a set of Horn-like clauses. A set of inference rules, derived from S, is obtained as follows. For each clause $L :- L_1, L_2, \ldots, L_n$ in S, we have the following clause rule:

Clause Rules

$$\frac{\Gamma_0 \rightarrow L_1 \Rightarrow \Gamma_1 \rightarrow L_1,\ \Gamma_1 \rightarrow L_2 \Rightarrow \Gamma_2 \rightarrow L_2, \ldots,\ \Gamma_{n-1} \rightarrow L_n \Rightarrow \Gamma_n \rightarrow L_n}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_n \rightarrow L}$$

We also have assumption axioms and a case analysis (splitting) rule. Let L be a positive literal. Then the assumption axioms and case analysis rule can be stated as follows:

Assumption Axioms

$$\Gamma \rightarrow L \Rightarrow \Gamma \rightarrow L \quad \text{if } L \in \Gamma$$

$$\Gamma \rightarrow \overline{L} \Rightarrow \Gamma, \overline{L} \rightarrow \overline{L}$$

Case Analysis (splitting) Rule

$$\frac{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1, \overline{M} \rightarrow L,\ \Gamma_1, M \rightarrow L \Rightarrow \Gamma_1, M \rightarrow L}{\Gamma_0 \rightarrow L \Rightarrow \Gamma_1 \rightarrow L}$$

The goal-subgoal structure of this deduction system is evident. The input clause $L :- L_1, L_2, \ldots, L_n$ merely states that $L_1, L_2, \cdots, L_n$ have to be confirmed in order to confirm L. The corresponding clause rule for $L :- L_1, L_2,..., L_n$ states that, if the initial subgoal is $\Gamma \rightarrow L$, then make $L_1, \ldots, L_n$ subgoals in succession; add to

Γ successively the literals that are needed to make each one provable; and finally, return $\Gamma_n \rightarrow L$ where Γ_n contains all the literals needed to make $L_1, \ldots, L_n$ provable.

Sprfn implements the natural deduction system just described. **Sprfn** exploits Prolog style depth-first iterative-deepening search. This search strategy involves repeatedly performing exhaustive depth-first search with increasing depth bounds. For a description of the strategy, see [6]. This search strategy is complete and can be efficiently implemented in Prolog, taking advantage of Prolog's built-in depth-first search with backtracking.

3 SPRFN and Term Rewriting

3.1 Input Format

The input to **sprfn** is formatted in Horn-like clauses. Given a set S of clauses, we convert them into Horn-like clauses as follows. For a clause containing at least one positive literal, we select one such literal to be the head, negate the remaining literals, and move them to the body of the clause. For an all-negative clause, we use **false** as the head of the clause and form the body from the positive literals corresponding to the original literals. The following example shows how to translate from clause form into the format accepted by **sprfn**. Notice the similarity of the input format syntax to Prolog program syntax.

Clause Form

$P(x) \vee Q(x)$
$\sim P(x) \vee R(x)$
$\sim Q(x) \vee R(x)$
$\sim R(a)$

Input Format for **sprfn**

$p(X) :- not(q(X))$
$r(X) :- p(X)$
$r(X) :- q(X)$
$false :- r(a)$

For input to **sprfn**, the convention is that a name starting with a capital letter is a variable name; all other names are predicate names, function names or constants. *Not* and *false* are reserved for negation and for the head of the top-level goal, respectively.

3.2 The Method of Proof

The prover attempts to prove that *false* is derivable from the input clauses. For example, given the following set of clauses:

```
p(X) :- not(q(X))
r(X) :- p(X)
r(X) :- q(X)
false :- r(a)
```

sprfn will derive the following proof:

```
false :- cases(
            (not q(a):
               (r(a) :- (p(a) :- not q(a)))),
            (q(a):
               (r(a) :- q(a)))
            )
```

Thus, *false* can be proven from the input clauses. For there are two cases to consider: (1) Suppose not q(a) is true; then we can derive *false* as follows. Since we are given that false :- r(a), we make r(a) our subgoal. Now we can derive r(a) if we can prove p(a), since we are given r(X) :- p(X). Meanwhile, we can derive p(a) if we can prove not q(a), since we are given p(X) :- not q(X). However, we are assuming not q(a), so this subgoal can be proven. (2) Suppose q(a) is true; then we can derive *false* as follows. Once again, we make r(a) our subgoal, since we are given that false :- r(a). Now we can derive r(a) if we can prove q(a), since we are given r(X) :- q(X). But we are assuming q(a), so this subgoal can be proven.

3.3 The Term Rewriting Mechanism in SPRFN

Replace. An assertion of the form **replace(<expr1>, <expr2>)** in the input signifies that all subgoals of form **<expr1>** should be replaced by subgoals of the form **<expr2>** before attempting to solve them. This is like a rewrite applied at the 'top level'. This is sound if **<expr1> :- <expr2>** is valid.

Rewrite. An assertion of the form **rewrite(<expr1>, <expr2>)** in the input signifies that all subexpressions of form **<expr1>** should be replaced by subexpressions of the form **<expr2>**. This is like a rewrite applied anywhere, not just at the top level. This is sound if the logical equivalence **<expr1> <-> <expr2>** is valid, or, in case when the expressions are terms, if the equation **<expr1>** = **<expr2>** is valid.

In our experiments, we translated the axioms of von Neumann-Bernays-Gödel set theory into a list of rewrite rules and then attempted to derive various theorems based on these rules. For example, consider the axiom for Subset below:

$(\forall x,y)[x \subseteq y \leftrightarrow (\forall u)[(u \in x \rightarrow u \in y)]]$

This would be translated into the following two rewrite rules, which would be given as input to the prover:

rewrite(sub(X,Y), or(not(el(f17(X,Y),X)), el(f17(X,Y),Y))).
rewrite(not(sub(X,Y)), and(el(U,X), not(el(U,Y)))).

Several points deserve mention. First of all, note that the single axiom gives rise to *two* rewrite rules -- a "positive" as well as a "negative" rule. This is to preserve soundness, since **sprfn** performs outermost term rewriting. The presence of the negative rewrite rule insures that whenever **sprfn** rewrites a term of the form *sub(X,Y)* with *or(not(el(f17(X,Y),X)), el(f17(X,Y),Y)))* (which implies that **sprnf** is using the positive rule) we know that this term does not appear in a negative context; for if it did, the prover would already have rewritten it using the negative rule.

We should also point out what may seem at first to be a counter-intuitive feature of these rewrite rules. Note the presence of the skolem function *f17(X,Y)* in the positive rewrite rule and the unbound variable *U* in the negative rule. One might think that the situation should be reversed. However, the correctness of this procedure can be seen by reflecting upon the following. Recall that **sprfn** performs subgoaling in attempting to prove *false*. Thus if the prover is attempting to prove *A*, let's say, and it tries to do this by trying to prove the subgoal *B*, this procedure will only be sound if it is the case that $B \rightarrow A$. Our rewrite rules must observe this fact. Hence, if we are trying to prove *A* and we attempt to do so by rewriting *A* with *B* and then trying to prove the subgoal *B*, it must be the case that $B \rightarrow A$. Or, to put the matter in Prolog symbolism, it must be the case that *A :- B*. When we skolemize the original axiom, we see that the following are logical consequences of the skolemized input clauses:

```
sub(X,Y) :- or(not(el(f17(X,Y),X)), el(f17(X,Y),Y))
not(sub(X,Y)) :- and(el(U,X), not(el(U,Y)))
```

Thus, we must express our two rewrite rules as given above.

4 Term Rewriting with a Tautology Checker

In our first experiment, we modified **sprfn** to make use of a **tautology checker**. Suppose that we wish to prove the set theoretic theorem *T*, which, in accordance with the procedure outline above, has been converted into the top-level goal: "false :- X".

If the flag t_test is set, then the prover will call the tautology checker tautology3(X,Y), where X is the input theorem (derived from the top-level goal "false :- X") and Y is the output consisting of the non-tautologous part (if any) of X. If X is a tautology, then the prover will halt; else, the original goal: "false :- X" is retracted and replaced in the database with the new goal: "false :- Y". The prover then proceeds to attempt to prove *false* by means of the subgoaling method described above. This method seems to work quite well. For one thing, if X is a tautology, the tautology checker allows the prover to spot this fact much sooner than if it had attempted to achieve its top-level goal by means of its subgoaling mechanism alone. For another, we have found that when X is not a tautology, by removing the tautologous portion of X and returning Y as the subgoal to be proved, we save the prover considerable time and avoid needlessly duplicated effort.

Because tautology3(X,Y) does not unify variables (it only eliminates a disjunction as a tautology if some literal L appears both negated and un-negated in the clause), as a standard practice we have included the axiom: "or(X,Y) :- prolog(tautology(or(X,Y)))" to handle cases where unifying is necessary to eliminate tautologous clauses. This allows us to invoke Prolog from within **sprfn**, and to call the Prolog predicate tautology/1 which succeeds if its input can be converted into a tautology via unification.

Thus backtracking over the elimination of a tautologous clause is still possible, but it only occurs with respect to the "or" rewrite rule. This seems more efficient than permitting backtracking into the tautology3 routine itself (which would be required if we allowed unification within tautology3).

We now exhibit two examples of the prover at work, utilizing the tautology checker.

4.1 Example 1

In this first example, we show how the tautology checker returns the non-tautologous portion of its input theorem, which is then proven by **sprfn**'s subgoaling mechanism.

Proof of Difference and Join Theorem

Our top-level goal is:

```
false:-eq(diff(a,b),join(a,comp(b)))
```

After reading in the input clauses, which contain our set theoretic rewrite rules as well as a few axioms, the prover begins by calling our tautology checker:

```
t_test is asserted
b_only is asserted
solution_size_mult(0.1) is asserted
proof_size_mult(0.4) is asserted

calling(tautology3(eq(diff(a,b),join(a,comp(b))),_9812))

    .
    .
    .
```

After removing the tautologous portion of the theorem, tautology3 returns the following:

```
conjunct:
   m(f17(diff(a,b),comp(b)))
   not el(f17(diff(a,b),comp(b)),a)
   el(f17(diff(a,b),comp(b)),b)

Continue?: yes.
```

At this point, the tautology checker informs the user that it has a conjunction of disjunctions (in this case there is only one such disjunction) left, which it could not eliminate via tautology checking alone. It asks the user if he wishes to proceed, and in this case, we answer in the affirmative. The prover's subgoaling procedure is now invoked, and in a short time **sprfn** returns with the following:

```
proof found
false:-cases(
        (not el(f17(diff(a,b),comp(b)),a):
           (or(m(f17(diff(a,b),comp(b))),or(not el(f17(diff(a,b),comp(b)),a),
            el(f17(diff(a,b),comp(b)),b))):-(or(not el(f17(diff(a,b),comp(b)),a),
            el(f17(diff(a,b),comp(b)),b)):-not el(f17(diff(a,b),comp(b)),a)))),

        (el(f17(diff(a,b),comp(b)),a):
           (or(m(f17(diff(a,b),comp(b))),or(not el(f17(diff(a,b),comp(b)),a),
            el(f17(diff(a,b),comp(b)),b))):-(m(f17(diff(a,b),comp(b))):-
            el(f17(diff(a,b),comp(b)),a)))))

size of proof 7

8.73 cpu seconds used
5 inferences done
```

It is worth pointing out that by using the term rewriting facility *without* invoking the tautology checker, the prover was able to derive the theorem in 128.43 cpu seconds with 34 inferences. We attempted to prove the theorem using neither the tautology checker nor rewrite rules; but after letting the prover run for over two hours without finding the proof, we put it out of its misery.

4.2 Example 2

In this second example, we show the prover's term rewriting facility in action. In this particular case, the tautology checker is able to establish that the entire input theorem is a tautology; hence it is unnecessary to invoke **sprfn**'s subgoaling mechanism, since the theorem is already proven.

Proof of Power Set Theorem

Our top-level goal is:

```
false:-eq(pset(join(a,b)),join(pset(a),pset(b)))
```

After reading in the input clauses, which contain our set theoretic rewrite rules as well as a few axioms, the prover begins by calling our tautology checker:

```
t_test is asserted
b_only is asserted
solution_size_mult(0.1) is asserted
proof_size_mult(0.4) is asserted

calling(tautology3(eq(pset(join(a,b)),join(pset(a),pset(b))),_9818))
```

The rewriting mechanism displays the results of its outermost term rewriting operation:

```
rewrite(eq(pset(join(a,b)),join(pset(a),pset(b))),and(sub(pset(join(a,b)),
   join(pset(a),pset(b))),sub(join(pset(a),pset(b)),pset(join(a,b)))))

rewrite(sub(pset(join(a,b)),join(pset(a),pset(b))),and(sub(pset(join(a,b)),
   pset( a)),sub(pset(join(a,b)),pset(b))))

rewrite(sub(pset(join(a,b)),pset(a)),or(not el(f17(pset(join(a,b)),pset(a)),
   pset(join(a,b))),el(f17(pset(join(a,b)),pset(a)),pset(a))))

rewrite(not el(f17(pset(join(a,b)),pset(a)),pset(join(a,b))),not sub(f17(pset(
   join(a,b)),pset(a)),join(a,b)))

rewrite(not sub(f17(pset(join(a,b)),pset(a)),join(a,b)),or(not sub(f17(pset(
   join(a,b)),pset(a)),a),not sub(f17(pset(join(a,b)),pset(a)),b)))

rewrite(el(f17(pset(join(a,b)),pset(a)),pset(a)),sub(f17(pset(join(a,b)),
   pset(a)),a))

rewrite(sub(pset(join(a,b)),pset(b)),or(not el(f17(pset(join(a,b)),pset(b)),
   pset(join(a,b))),el(f17(pset(join(a,b)),pset(b)),pset(b))))

rewrite(not el(f17(pset(join(a,b)),pset(b)),pset(join(a,b))),not sub(f17(pset(
   join(a,b)),pset(b)),join(a,b)))

rewrite(not sub(f17(pset(join(a,b)),pset(b)),join(a,b)),or(not sub(f17(pset(
   join(a,b)),pset(b)),a),not sub(f17(pset(join(a,b)),pset(b)),b)))

rewrite(el(f17(pset(join(a,b)),pset(b)),pset(b)),sub(f17(pset(join(a,b)),
   pset(b)),b))

rewrite(sub(join(pset(a),pset(b)),pset(join(a,b))),or(not el(f17(join(pset(a),
   pset(b)),pset(join(a,b))),join(pset(a),pset(b))),el(f17(join(pset(a),pset(b)),
   pset(join(a,b))),pset(join(a,b)))))

rewrite(not el(f17(join(pset(a),pset(b)),pset(join(a,b))),join(pset(a),pset(b))),
   or(not el(f17(join(pset(a),pset(b)),pset(join(a,b))),pset(a)),not el(f17(
   join(pset(a),pset(b)),pset(join(a,b))),pset(b))))

rewrite(not el(f17(join(pset(a),pset(b)),pset(join(a,b))),pset(a)),not sub(f17(
   join(pset(a),pset(b)),pset(join(a,b))),a))

rewrite(not el(f17(join(pset(a),pset(b)),pset(join(a,b))),pset(b)),not sub(f17(
   join(pset(a),pset(b)),pset(join(a,b))),b))

rewrite(el(f17(join(pset(a),pset(b)),pset(join(a,b))),pset(join(a,b))),sub(f17(
   join(pset(a),pset(b)),pset(join(a,b))),join(a,b)))

rewrite(sub(f17(join(pset(a),pset(b)),pset(join(a,b))),join(a,b)),and(sub(f17(
   join(pset(a),pset(b)),pset(join(a,b))),a),sub(f17(join(pset(a),pset(b)),
   pset(join(a,b))),b)))
```

At this point, rewriting has been completed; the procedure **cnf_expand** is now invoked to expand the rewritten theorem into conjunctive normal form and to then eliminate all tautologous conjuncts.

```
call(0,cnf_expand(and(and(or(or(not sub(f17(pset(join(a,b)),pset(a)),a),not sub(
f17(pset(join(a,b)),pset(a)),b)),sub(f17(pset(join(a,b)),pset(a)),a)),or(or(not
sub(f17(pset(join(a,b)),pset(b)),a),not sub(f17(pset(join(a,b)),pset(b)),b)),sub(
f17(pset(join(a,b)),pset(b)),b))),or(or(not sub(f17(join(pset(a),pset(b)),pset(
join(a,b))),a),not sub(f17(join(pset(a),pset(b)),pset(join(a,b))),b)),and(sub(f17
(join(pset(a),pset(b)),pset(join(a,b))),a),sub(f17(join(pset(a),pset(b)),pset(
join(a,b))),b)))),_15815))
```

Initially, when **cnf_expand** is called, its output argument is the uninstantiated Prolog variable _15815. But when it returns, this output argument has been instantiated to the empty list, signifying that no non-tautologous portion of the theorem remains:

```
result(0,cnf_expand(and(and(or(or(not sub(f17(pset(join(a,b)),pset(a)),a),not sub(
f17(pset(join(a,b)),pset(a)),b)),sub(f17(pset(join(a,b)),pset(a)),a)),or(or(not
sub(f17(pset(join(a,b)),pset(b)),a),not sub(f17(pset(join(a,b)),pset(b)),b)),sub(
f17(pset(join(a,b)),pset(b)),b))),or(or(not sub(f17(join(pset(a),pset(b)),pset(
join(a,b))),a),not sub(f17(join(pset(a),pset(b)),pset(join(a,b))),b)),and(sub(f17
(join(pset(a),pset(b)),pset(join(a,b))),a),sub(f17(join(pset(a),pset(b)),pset(
join(a,b))),b)))),[]))

tautology3 returns: is_tautology

theorem_is_a_tautology

4.28 cpu seconds used
0 inferences done
```

We observed two very important things while running these tests. First of all, we found that including explicit rewrite rules to distribute "or" over "and" significantly slowed down the tautology checker. (Fortunately, the **cnf_expand** routine is able to test for tautologies without requiring that its input argument be in conjunctive normal form; hence employing the distribution rules is not needed.) We ran tests in which these distribution rules were used and tests in which they were not. The results are contained in the Appendix.

Secondly, we discovered that the depth to which term rewriting is allowed to take place greatly affects overall performance. For example, in the case of the Power Set theorem exhibited above, we did *not* include in our input the rewrite rules for the Subset axiom. By omitting those two rules (see the earlier section: "The Term Rewriting Mechanism in **SPRFN**") we cause the prover to regard terms of the form "sub(X,Y)" as atomic and thus it does not rewrite them. In this way, it is able to discover that the entire theorem is a tautology. On the other hand, we found that if we included the rewrite rules for the Subset axiom, then our tautology checker was no longer able to eliminate the entire theorem as a tautology; indeed, it returned a significantly long conjunction, which the subgoaling mechanism then had to prove. This took a much greater amount of time. (Cf. Table 2 of the Appendix.) For a complete summary of our test results using the tautology checker, the reader should consult the Appendix.

5 Preprocessing Input via Term Rewriting

In our second experiment, we used our term rewriting facility as a preprocessor. We discovered in our earlier experiments that, as a general rule, the more complex the theorem, the greater the number of terms that ultimately result from rewriting the theorem. In fact, we found that for certain theorems, such as the Composition of Homomorphisms theorem (see below) it was physically impossible to use the tautology checker. This was due to the fact that one term was being rewritten to a conjunction (or disjunction) of several other terms, each of which was itself subject to being rewritten into a complex of several terms and so on. Thus, nearly exponential growth of the Prolog structure occurred during the operation of the rewriting facility. This eventually caused Prolog to run out of stack long before the **cnf_expand** subroutine had a chance to eliminate any tautologous portion of the theorem.

We decided, therefore, to preprocess the theorem by reducing the size of the term that appeared as the body in the top-level goal. In general, our approach involved skolemizing the negated theorem and then using the rewriting facility to produce the initial set of input clauses. As an illustration of this technique, we present the following proof of the Composition of Homomorphisms theorem. We should point out that it was necessary to add three simple axioms in order to derive the proof; also, it was necessary once again to restrain the depth to which rewriting took place.

Proof of Composition of Homomorphisms Theorem

Our theorem is the following:

$$(\forall xh1,xh2,xs1,xs2,xs3,xf1,xf2,xf3)[(hom(xh1,xs1,xf1,xs2,xf2) \wedge$$
$$hom(xh2,xs2,xf2,xs3,xf3)) \rightarrow hom(compose(xh2,xh1),xs1,xf1,xs3,xf3)]$$

After skolemizing the negation of the theorem we have three clauses to be rewritten: hom(ah1,as1,af1,as2,af2), hom(ah2,as2,af2,as3,af3), and not(hom(compose(ah2,ah1),as1,af1,as3,af3)). Based on these clauses, the prover's term rewriting facility produced the following set of input clauses:

Clauses derived from hom(ah1,as1,af1,as2,af2):

```
eq(apply(ah1,apply(af1,ord_pair(G1,G2))),
  apply(af2,ord_pair(apply(ah1,G1),apply(ah1,G2)))) :-
    el(G1,as1), el(G2,as1).
maps(ah1,as1,as2).
closed(as2,af2).
closed(as1,af1).
```

Clauses derived from hom(ah2,as2,af2,as3,af3):

```
eq(apply(ah2,apply(af2,ord_pair(G3,G4))),
  apply(af3,ord_pair(apply(ah2,G3),apply(ah2,G4)))) :-
    el(G3,as2), el(G4,as2).
maps(ah2,as2,as3).
closed(as3,af3).
closed(as2,af2).
```

Clauses derived from not(hom(compose(ah2,ah1),as1,af1,as3,af3)):

```
el(g5,as1).
el(g6,as1).
false :-
  eq(apply(ah2,apply(ah1,apply(af1,ord_pair(g5,g6)))),
    apply(af3,ord_pair(apply(ah2,apply(ah1,g5)),apply(ah2,apply(ah1,g6))))),
  maps(compose(ah2,ah1),as1,as3),
  closed(as3,af3),
  closed(as1,af1).
```

Note that our top-level goal has become:

```
false :-
  eq(apply(ah2,apply(ah1,apply(af1,ord_pair(g5,g6)))),
    apply(af3,ord_pair(apply(ah2,apply(ah1,g5)),apply(ah2,apply(ah1,g6))))),
  maps(compose(ah2,ah1),as1,as3),
  closed(as3,af3),
  closed(as1,af1).
```

In addition to these input clauses, we added three axioms. The first two of these are trivial while the third, although non-trivial, can be derived by the prover in 24.63 cpu seconds after 15 inferences.

Axioms for proof of homomorphism theorem:

```
eq(apply(XF1,S1),apply(XF2,S2)) :-
  eq(S1,S3), eq(apply(XF1,S3),apply(XF2,S2)).

el(apply(XF,X),S2) :- maps(XF,S1,S2), el(X,S1).

maps(compose(X,Y),S1,S3) :- maps(Y,S1,S2),maps(X,S2,S3).
```

Finally, we added some extra rewrite rules which serve only to cut down on the size of data structures that result from term rewriting.

Rewrite Rules to handle large terms:

```
rewrite(f32(ah1,as1,af1,as2,af2),g1).
rewrite(f33(ah1,as1,af1,as2,af2),g2).
rewrite(f32(ah2,as2,af2,as3,af3),g3).
rewrite(f33(ah2,as2,af2,as3,af3),g4).
rewrite(f32(compose(ah2,ah1),as1,af1,as3,af3),g5).
rewrite(f33(compose(ah2,ah1),as1,af1,as3,af3),g6).
rewrite(apply(compose(XF1,XF2),S),apply(XF1,apply(XF2,S))).
```

Given this preprocessed input, **sprfn** is able to derive the following proof of the theorem:

```
proof found
false:-lemma((eq(apply(ah2,apply(ah1,apply(af1,ord_pair(g5,g6)))),
        apply(af3,ord_pair(apply(ah2,apply(ah1,g5)),
        apply(ah2,apply(ah1,g6))))):-[])),

    (maps(compose(ah2,ah1),as1,as3):-
        maps(ah1,as1,as2),
        maps(ah2,as2,as3)),

    closed(as3,af3),
    closed(as1,af1).

size of proof 18

30.2333 cpu seconds used
14 inferences done
```

Note that the proof involves a lemma, which **sprfn** derived in the course of its operation. If we so desire, we can ask the prover to show us how it came up with this lemma. When we do so, it responds with the following derivation:

```
proof of lemma:
false:-(eq(apply(ah2,apply(ah1,apply(af1,ord_pair(g5,g6)))),
      apply(af3,ord_pair(apply(ah2,apply(ah1,g5)),
      apply(ah2,apply(ah1,g6))))):-

     lemma((eq(apply(ah1,apply(af1,ord_pair(g5,g6))),
          apply(af2,ord_pair(apply(ah1,g5),apply(ah1,g6)))):-[])),

     (eq(apply(ah2,apply(af2,ord_pair(apply(ah1,g5),apply(ah1,g6)))),
      apply(af3,ord_pair(apply(ah2,apply(ah1,g5)),
      apply(ah2,apply(ah1,g6))))):-

          lemma((el(apply(ah1,g5),as2):-[])),
          (el(apply(ah1,g6),as2):-

               maps(ah1,as1,as2),el(g6,as1)))).

size of proof 11

26.9166 cpu seconds used
13 inferences done
```

6 Summary of Results

The techniques we employed allowed us to prove moderately sophisticated set theoretic theorems in rapid time with few inferences. These theorems would have been much more difficult to derive without the rewrite rules; indeed, **sprfn** was unable to derive some of them when run without the rewrite rules. Undoubtedly it would have been beyond the power of a typical resolution theorem prover to derive most of the theorems in question.

We have found that removing the tautologous portion of a theorem by means of some filter such as our tautology checker seems to speed up the derivation time, by allowing the prover to focus its attention on the non-tautologous aspects of the theorem. Furthermore, we discovered that the depth to which term rewriting is allowed greatly affects the prover's ability to arrive at a proof. Clearly, more work needs to be done in this area. At the present time, human intervention is required to adjust term rewriting depth; hopefully this can be automated to some extent in the future.

Our research leads us to conclude that preprocessing input clauses by means of rewrite rules is also highly effective in directing a theorem prover's attention towards a fast, relatively short proof. Although this kind of preprocessing is presently being done by hand, we are confident that it can be fully automated.

Finally, among the practical results that we obtained, it bears mentioning that it pays to avoid distributing "or" over "and" by means of rewrite rules.

At the same time, we discovered that there *are* limits to the power of term rewriting in connection with proving theorems from set theory. For one thing, we found that a clause's size grows almost exponentially when terms are rewritten by terms which are themselves subject to being rewritten, and so forth. Although this problem has no effect on soundness, the physical limitations of the computer itself come into play at this point, causing the prover to run out of stack before it can complete its rewriting phase.

We also realize that our procedure is not complete, if rewriting takes place at the wrong time. For example, suppose we have the rewrite rule: $B \text{ ->> } P(x)$ and we wish to demonstrate that the following theorem is a tautology:

$$B \vee (\sim P(a) \wedge \sim P(b))$$

If we rewrite B before we distribute "or" over "and", we have:

$$P(x) \vee (\sim P(a) \wedge \sim P(b))$$

from which we can only derive:

$$(P(x) \vee \sim P(a)) \wedge (P(x) \vee \sim P(b))$$

and this is not tautologous no matter how we instantiate the variable x. Yet if we distribute "or" over "and" before rewriting B, we have:

$$(B \vee \sim P(a)) \wedge (B \vee \sim P(b))$$

from which we can derive the tautology:

$$(P(x) \vee \sim P(a)) \wedge (P(y) \vee \sim P(b))$$

since Prolog will provide a different variable each time it replaces B with $P(x)$.

This raises the following questions: Is term replacement more complete than term rewriting? How complete is term replacement for existentially quantified variables? Is replacement equivalent to delayed term rewriting? More work needs to be done before we are in a position to answer these questions.

Finally, the approaches to term rewriting that we explored are not sufficient when trying to prove theorems that require *creative insight*. For example, in one of our experiments we tried to deduce Cantor's Theorem using our rewrite rules. However, we discovered that **sprfn** was unable to find the proof without being given quite a bit of non-trivial information. Specifically, we had to provide it with axioms implying (1) that any function induces its own diagonal set and (2) that the relation which pairs a unit set with its single element is a one-one function. Once these axioms were supplied, by making use of our rewrite rules the prover was able to derive Cantor's Theorem in 33.65 cpu seconds with 12 inferences. Nevertheless, one would like the prover to be able to realize on its own that such sets and functions exist. Yet recognizing that there *is* such a thing as the diagonal of a function and that such a set might be useful in this case requires a kind of insight that goes far beyond syntactic manipulations. Unfortunately, term rewriting alone does not provide the necessary machinery for the prover to possess this kind of creative insight.

References

[1] Plaisted, D.A., 'A simplified problem reduction format', *Artificial Intelligence* **18** (1982) 227-261

[2] Boyer, Robert, Lusk, Ewing, McCune, William, Overbeek, Ross, Stickel, Mark, and Wos, Lawrence, 'Set theory in first-order logic: clauses for Gödel's axioms', *Journal of Automated Reasoning* **2** (1986) 287-327

[3] Plaisted, D.A., 'Another extension of Horn clause logic programming to non-Horn clauses', Lecture Notes 1987

[4] Plaisted, D.A., 'Non-Horn clause logic programming without contrapositives', unpublished manuscript 1987

[5] Loveland, D.W., *Automated Theorem Proving: A Logical Base,* North-Holland Publishing Co., 1978, Chapter 6.

[6] Korf, R.E., 'Depth-first iterative-deepening: an optimal admissible tree search', *Artificial Intelligence* **27** (1985) 97-109.

Appendix
Test Results Using a Tautology Checker

Table 1

	Theorems
(1)	false :- eq(union(a,b),union(b,a)).
(2)	false :- eq(join(a,b),join(b,a)).
(3)	false :- eq(union(a,a),a).
(4)	false :- eq(join(a,a),a).
(5)	false :- eq(union(a,comp(a)), universe).
(6)	false :- eq(join(a,comp(a)), 0).
(7)	false :- eq(comp(universe),0).
(8)	false :- eq(comp(0),universe).
(9)	false :- eq(comp(comp(a)),a).
(10)	false :- eq(union(a,0),a).
(11)	false :- eq(join(a,universe),a).
(12)	false :- eq(union(a,universe),universe).
(13)	false :- eq(join(a,0),0).
(14)	false :- eq(union(union(a,b),c),union(a,union(b,c))).
(15)	false :- eq(join(join(a,b),c),join(a,join(b,c))).
(16)	false :- if(sub(a,b), then(eq(join(a,b),a))).
(17)	false :- eq(comp(union(a,b)),join(comp(a),comp(b))).
(18)	false :- eq(comp(join(a,b)),union(comp(a),comp(b))).
(19)	false :- eq(join(union(a,b),union(a,comp(b))),a).
(20)	false :- eq(diff(a,b),join(a,comp(b))).
(21)	false :- eq(union(a,universe),universe).
(22)	false :- eq(join(a,union(b,c)), union(join(a,b), join(a,c))).
(23)	false :- eq(union(a,join(b,c)), join(union(a,b), union(a,c))).
(24)	false :- sub(0,a).
(25)	false :-- if(and(sub(a,b),sub(b,c)),then(sub(a,c))).
(26)	false :- if(sub(a,b),then(el(a,pset(b)))).
(27)	false :- if(disjoint(a,b),then(eq(join(a,b),0))).
(28)	false :- sub(a,union(a,b)).
(29)	false :- sub(diff(a,b),a).
(30)	false :- if(sub(a,join(b,c)), then(and(sub(a,b),sub(a,c)))).
(31)	false :- eq(pset(join(a,b)),join(pset(a),pset(b))).
(32)	false :- eq(pset(join(a,b)),join(pset(a),pset(b))).
(33)	false :- sub(prod(a,join(b,c)),join(prod(a,b),prod(a,c))).
(34)	false :- if(and(sub(a,b),sub(c,d)),then(sub(prod(a,c),prod(b,d)))).
(35)	false :- if(and(meq(a,b),meq(c,d)),then(eq(ord_pair(a,c),ord_pair(b,d)))).
(36)	false :- if(eq(a,ord_pair(b,c)),then(opp(a))).
(37)	false :- if(and(m(a),m(b)),then(sub(set(a),set(a,b)))).
(38)	false :- if(and(m(a),m(b)), then(eq(set(a,b),set(b,a)))).

Note that theorems (31) and (32) are the same. However, (31) was proven using a rewrite rule for the subset axiom, while (32) was proven using a replace rule for the subset axiom. Using a replace rather than a rewrite rule prevented terms containing the "subset" predicate from being rewritten before tautology checking was performed. This allowed the prover to find the proof much faster in the case of this particular theorem.

Table 2

Summary of Results				
	With "or-over-and" Distribution Rules		Without "or-over-and" Distribution Rules	
Theorem	Time	Inferences	Time	Inferences
(1)	3.23	0	2.5	0
(2)	4.18	0	4.14	0
(3)	1.66	0	1.61	0
(4)	1.68	0	1.76	0
(5)	5.93	4	5.31	4
(6)	5.96	6	5.85	6
(7)	3.66	4	3.5	4
(8)	2.7	2	2.68	2
(9)	5.73	4	4.86	4
(10)	2.91	2	2.53	2
(11)	4.53	4	4.46	4
(12)	4.73	4	4.33	4
(13)	2.76	2	2.73	2
(14)	9.68	0	5.51	0
(15)	10.88	0	10.88	0
(16)	7.48	4	4.91	4
(17)	10.86	0	6.64	0
(18)	18.1	0	5.94	0
(19)	9.34	0	5.33	0
(20)	10.11	5	8.73	5
(21)	4.66	4	4.53	4
(22)	20.55	0	8.44	0
(23)	19.88	0	7.73	0
(24)	1.26	2	1.18	2
(25)	12.26	8	9.63	8
(26)	3.76	4	3.21	4
(27)	18.36	14	15.85	14
(28)	0.81	0	0.78	0
(29)	0.78	0	0.84	0
(30)	40.76	16	24.04	16
(31)	217.96	32	189.38	32
(32)	4.83	0	4.28	0
(33)	3.38	0	3.11	0
(34)	63.55	32	34.63	16
(35)	15.96	0	4.93	0
(36)	69.11	23	37.78	16
(37)	67.21	0	4.25	0
(38)	109.00	0	8.84	0

These results were derived by using a tautology-checker in conjunction with rewrite/replace rules.

SUMMARY: In each case, the number of inferences required is virtually the same whether or not the "or-over-and" distribution rules are used. However, in almost every instance there is a speed-up when these rules are not used. Furthermore, as a general rule it seems that as the amount of time required to prove the theorem increases, the greater the speed-up when the "or-over-and" rules are not used.