An Overview of Rewrite Rule Laboratory (RRL)

Deepak Kapur *
Department of Computer Science
State University of New York at Albany
Albany, NY 12222
kapur@cs.albany.edu

Hantao Zhang†
Department of Computer Science
The University of Iowa
Iowa City, IA 52242
hzhang@cs.uiowa.edu

Abstract

RRL (*Rewrite Rule Laboratory*) was originally developed as an environment for experimenting with automated reasoning algorithms for equational logic based on rewrite techniques. It has now matured into a full-fledged theorem prover which has been used to solve hard and challenging mathematical problems in automated reasoning literature as well as a research tool for investigating the use of formal methods in hardware and software design. We provide a brief historical account of development of *RRL* and its descendants, give an overview of the main capabilities of *RRL* and conclude with a discussion of applications of *RRL*.

Key words. RRL, rewrite techniques, equational logic, discrimination nets

1 Introduction

The theorem prover *RRL* (*Rewrite Rule Laboratory*) is an automated reasoning program based on rewrite techniques. The theorem prover has implementations of completion procedures for generating a complete set of rewrite rules from an equational axiomatization, associative-commutative matching and unification, algorithms for orienting equations into terminating rewrite rules, refutational methods for first-order theorem proving, and, most important, methods for proving first-order equational formulas by induction. *RRL* has served for us as well as our colleagues as an excellent test bed for developing new ideas in automated reasoning, trying them out to assess their effectiveness, thus weeding out bad and insignificant ideas from good ones. When the project to build a rewrite rule laboratory was initiated in 1982, we did not expect that an outcome of this project would be powerful theorem provers that could be used for attacking mathematically challenging problems as well as for application in specification analysis, verification, and experimentation with formal methods in system design. For us, that is yet another vindication of experimental approach to automated reasoning that emphasizes implementing ideas and building systems.

*Partially supported by the National Science Foundation Grant no. CCR-9303394.
†Partially supported by the National Science Foundation Grants nos. INT-9016100, CCR-9202838, and CCR-9357851.

Short system abstracts of *RRL* and its descendants outlining extensions to *RRL* have appeared in the proceedings of *CADE* [33, 36, 16, 61] and *RTA* conferences [38, 1]. In this paper, we give a more detailed overview of the system and its capabilities, and we discuss some applications for which the theorem prover has been successfully used.

1.1 A Brief History of the RRL Project

The work on the *RRL* project was started at General Electric Corporate Research and Development (GECRD) and RPI in fall 1983, following a workshop on rewrite rule laboratories in Schenectady, New York [11]. Sivakumar wrote the first version of *RRL* as his M.S. degree project at RPI under Kapur's supervision. The first report on the *RRL* project appeared in the proceedings of the workshop [32]. The performance of an *RRL* implementation of the completion procedure discussed in a seminal paper by Knuth and Bendix [19] was analyzed on small examples for different rewriting strategies and selection criteria for picking equations to generate next inferences. The second report on *RRL* appeared in the 1986 CADE proceedings after Zhang began working at RPI as a Ph.D. student [33]. Since then, Kapur and Zhang have been working together on the *RRL* project. Their collaborations, as well as joint work with other colleagues including Sivakumar, Narendran, and Musser, have led to several theoretical results about automated reasoning algorithms and to implementation of many interesting algorithms in *RRL*.

RRL was originally written in Franz Lisp and was made compatible with a small subset of Zeta Lisp so that it could run on Lisp machines, Vax machines, and Sun workstations. In 1989, Sivakumar and Zhang ported *RRL* to Common Lisp. *RRL* has been distributed to over fifty universities and research laboratories. The source of *RRL* is freely available through anonymous ftp at herky.cs.uiowa.edu.

1.2 Descendants of RRL

As more and more capabilities have been added to *RRL*, it has become difficult to fine-tune *RRL* to particular applications. Consequently, *RRL* has been extended and modified to be suitable for specific purposes.

Herky (High-Performance Key Operations) is a fast implementation of the completion procedure of *RRL* to make it more suitable for equational reasoning. Several novel techniques have been implemented in *Herky*, such as fast rewriting and term-sharing, specialized completion methods for certain equational theories, and capabilities for handling nonterminating rewrite rules as constrained rewrite rules.

An interactive proof manager based on the ideas of the proof manager in the *AFFIRM* system has been developed for *RRL* by Hua and Zhang [16]. This interactive manager to support *failure-resistant induction* was used to prove many benchmark problems for inductive theorem proving.

Another interactive verification/proof management system called *Tecton* [1] has been developed on the top of *RRL* by Kapur, Musser, and Nie [1, 24] using a hyper-text system called *KMS* (Knowledge Management System). The emphasis in the development of Tecton is on proof visualization and management. Tecton supports a simple program modeling language (which includes assignment, sequencing, conditionals, while, and procedure calls) used in most of the literature on Hoare's axiomatic semantics formalism. Properties of programs in the modeling language can be proved in Tecton (a sample transcript appears in [24]). Presburger arithmetic has been integrated into the rewriting inference mechanism of the theorem prover.

More details about *RRL*'s descendants and their capabilities are given in later sections.

1.3 Capabilities of RRL

As mentioned earlier, *RRL* was initially implemented to study inference methods such as the Knuth-Bendix completion procedure [19] and lexicographic recursive path ordering, a method for orienting equations into terminating rewriting rules [8]. However, this environment soon began serving as a vehicle for generating new ideas, concepts, and algorithms that could be implemented and tested for their effectiveness. Experimentation using *RRL* has led, among other things, to development of criteria for identifying redundant inferences [22, 71], complexity studies of primitive operations such as matching, unification [25, 4, 26], efficient algorithms for primitive operations [27], approaches for first-order theorem proving [20, 70], methods for proving formulas by induction [30, 72], algorithms for checking the sufficient completeness property of specifications [28], and specialized completion procedures for equational theories [66].

RRL provides implementations of inference methods for

- generating decision procedures for equational theories using completion procedures;
- refutational methods for proving theorems in first-order predicate calculus with equality;
- proving formulas by induction using the explicit induction approach based on the cover set method, as well as using the *proof by consistency* approach [46, 21] (also called the *inductionless-induction* approach); and
- checking the consistency and completeness of equational specifications.

RRL is perhaps one of the few theorem provers in the world providing such extensive capabilities for equational logic, first-order theorem proving, and theorem proving by induction. *RRL* does not support any methods for higher-order reasoning.

In the next section, we discuss inference methods supported in *RRL* for first-order theorem proving from both clausal and nonclausal input, as well as for generating a complete set of rewrite rules from equational theories. In the third section, we discuss inference methods for proving equations by induction. In each subsection, we identify some interesting research issues being investigated by our group to further enhance the capabilities of *RRL*. The final section discusses some applications in which *RRL* and its descendants have been successfully used.

2 Automated Deduction in RRL

Automated deduction is supported in *RRL* using the extended Knuth-Bendix completion procedure. The input is either a set of pure equations or a set of first-order formulas (with or without equality); in the latter case, the formulas are converted into equations. Equations are transformed into *one-way terminating rewrite rules* based on a *well-founded order*. Rewrite rules are used to *interreduce* each other using rewriting. New equations are generated as *critical pairs* from rewrite rules by the mechanism of *superposition*. These equations are simplified, and any generated new rules are added for generating new inferences.

Two modes of proof search, forward reasoning and backward reasoning, are supported in *RRL*. In the forward reasoning mode, a canonical (terminating and confluent) rewrite system is attempted from the hypotheses, and during its generation it is checked whether the given conjecture can be rewritten to equal terms. If the completion procedure is successful in generating a canonical set of rules, these rules associate a unique normal form for each congruence class in the congruence relation induced by the theory. These rules thus serve as a decision procedure for the theory. *RRL* has been coded carefully to generate canonical systems conveniently and efficiently from equational theories.

The second mode of reasoning is refutational (proof by contradiction). ¿From the hypotheses and the negated conjecture, *RRL* attempts to generate a contradiction, which is a system including the rule *true* $\rightarrow$ *false*. This is especially useful for arbitrary first-order theories as well as for equational theories that may not possess a canonical rewrite system.

In the first subsection, we discuss forward-reasoning approach for theorem proving in equational theories. In subsequent sections, we discuss refutational approaches with a focus on theorem proving for full first-order predicate calculus with equality.

2.1 Primitive Inference Steps

An important aspect of the completion procedure-based approach to automated deduction is heuristic methods for orienting equations into rewrite rules. That is why a study of termination orderings has been an important research area of rewriting techniques. In *RRL*, rules can be made from equations manually or using an algorithm for *lexicographic recursive path ordering* (*lrpo*) [8] and the *associative-commutative lexicographic recursive path ordering* (*aclrpo*) [34]. Both *lrpo* and *aclrpo* extend a precedence relation on function symbols to terms.

Whenever an equation cannot be oriented into a terminating rewrite rule by the algorithm using an existing precedence relation, *RRL* presents the user with various options:

- postpone the equation for consideration later, with the hope that it would get simplified and the result can be made into a terminating rule;
- extend the precedence, that is, add some relation on operators to the precedence, or change the status of some operators (which decides how the arguments of these operators are compared against their counterparts);
- orient the equation by hand (either left to right or right to left) (the termination of the rewrite rule is to be ensured by the user);
- make rule $eq(t_1, t_2) \rightarrow true$ from equation $t_1 = t_2$;
- introduce a new operator (this option is very useful when both sides of an equation have a variable(s) whose value can be arbitrarily chosen without affecting the validity of the equation); or
- backtrack to a previous choice, or interrupt and start with a new ordering using the existing rule and equation sets.

The feature of introducing new operators turns out to be especially useful. It has been successfully employed to show the equivalence of different axiomatizations of free groups [37].

Two key operations in a completion procedure are *rewriting* and *superposition*. They are illustrated by using rewrite rules made from the three axioms of free groups.

$$\begin{array}{llcl} (1) & (e * x) & \rightarrow & x, \\ (2) & (i(x_1) * x_1) & \rightarrow & e, \\ (3) & (x_2 * y_2) * z_2 & \rightarrow & x_2 * (y_2 * z_2). \end{array}$$

Let t_1 be the expression $(i(x) * i(e * y)) * (e * y)$; t_1 can be reduced using the associativity rule (3) above, because t_1 is an instance of the left side of rule (3) (when $i(x)$, $i(e * y)$ and $e * y$ are respectively substituted for x_2, y_2, z_2). The result of *rewriting* t_1 using rule (3) is $i(x) * (i(e * y) * (e * y))$, which can be further reduced using rule (2) to a normal form $i(x) * e$.

Rule (2) above can be superposed on rule (3): the overlapping obtained by unifying $(x_2 * y_2)$ in the left side of rule (3) with the left side of rule (2), is:

$$(i(x_1) * x_1) * z_2.$$

This term can be rewritten in two different ways as follows: The pair of terms $\langle e * z_2, i(x_1) *$

$$\begin{array}{ccc} & (i(x_1) * x_1) * z_2 & \\ (2) \swarrow & & \searrow (3) \\ (e * z_2) & & i(x_1) * (x_1 * z_2). \end{array}$$

$(x_1 * z_2)\rangle$ constitutes a *critical pair* generated from rules (2) and (3). As should be obvious, this pair is in the congruence relation generated by the above three axioms. The process of computing critical pairs is called *superposition*. In fact, if we repeatedly apply rewriting and superposition to rules (1)–(3) and the generated new rules, we will obtain a canonical rewrite system that consists of ten rewrite rules.

Besides from rules (1)–(3), a canonical rewrite system for the free group theory can be also generated from the single axiom

$$i(z * i(z)) * (i(i((x * w) * y) * x) * i(y)) = w.$$

The following *RRL* commands are sufficient:

```
add i(x * i(x)) * (i(i((y * z) * u) * y) * i(u)) == z ]
operator status * left-to-right
operator precedence i *
kb
```

The precedence commands are given to specify a well-founded ordering used for orienting equations into rewrite rules. Below is a partial transcript generated, which illustrates how new operators are introduced by *RRL*:

```
Type Add, Auto, Break, Cmd, Close, Delete, Gc, Grammar, Init, Kb, List,
     Makerule, Option, Operator, Prove, Quit, Read, Stats, Test, Write or Help.
```

```
HERKY-> kb

Add Rule: [1] (i((x * i(x))) * (i((i(((y * z) * u)) * y)) * i(u))) ---> z

 ... ...

This equation cannot be oriented into a terminating rule:
   (i((x * i(x))) * (i((i((z1 * u)) * i((x1 * i(x1))))) * i(u))) ==
   (i((i(((y1 * z1) * u1)) * y1)) * i(u1))

Introduce a new operator:
   (i((x * i(x))) * (i((i((z1 * u)) * i((x1 * i(x1))))) * i(u))) == f1(z1)

Add Rule: [2] (i((i(((x * y) * z)) * x)) * i(z)) ---> f1(y)

Add Rule: [3] (i((x * i(x))) * (i((i((u * y)) * i((z * i(z))))) * i(y))) ---> f1(u)

  ... ...

Your system is canonical:

 [106] i(f4) ---> f4
 [165] i(i(x)) ---> x
 [186] (x * i(x)) ---> f4
 [232] (x * f4) ---> x
 [247] (i(x) * x) ---> f4
 [254] (f4 * x) ---> x
 [255] f1(x) ---> x
 [256] (i(y) * (y * x)) ---> x
 [258] (x * (i(x) * y)) ---> y
 [265] i((y * x)) ---> (i(x) * i(y))
 [291] ((z * x) * y) ---> (z * (x * y))

Number of rules generated             = 291
Number of rules retained              = 13
Number of critical pairs              = 656 (of which 32 are unblocked.)
Time used (incl. garbage collection)  = 17.470 sec
```

For equations that cannot be oriented into terminating rewriting rules, as well as for special equational theories (i.e., when function symbols satisfy certain properties such as an operator being commutative or associative and commutative, etc.), rewriting and critical pair computation may be done modulo an equational theory (see [43, 51] for extensions of the Knuth-Bendix completion procedure to associative-commutative theories). For first-order theorem proving, rewriting and critical pair computations are defined differently, making use of the properties of the representations of the formulas.

2.2 High-Performance Completion Procedures for Equational Theories

To implement rewriting and critical pair operations efficiently, we have adopted in *RRL* novel techniques for

- detecting unnecessary critical pairs,

Prob.	Final Rules	Otter		Hiper		Herky	
		Pairs	Time	Pairs	Time	Pairs	Time
A_{10}	36	8083	20.9	3205	17.8	800	6.8
A_{20}	66	46013	381.8	20165	175.0	2560	25.0
A_{30}	96	224543	2850.0	62925	688.3	6374	79.9
A_{40}	126	307873	10976.7	143485	2251.6	10884	178.9
A_{50}	156	579803	35384.8	–	–	16594	323.3
A_{60}	186	–	>10 hr.	–	–	23504	622.5
A_{80}	246	–	–	–	–	40924	1609.9
A_{100}	306	–	–	–	–	73144	2963.9

Table 1: Statistics of OTTER, Hiper, and Herky (in seconds)

- fast rewriting and term-sharing,
- specialized completion methods for certain equational theories, and
- methods for handling nonterminating rewrite rules as constrained rewrite rules.

Example 2.1 A family of problems were used by Jim Christian [7] as benchmark problems for comparing the performance of different implementations of completion procedures. The problems are represented by the following schema (which is a slight variation of group axioms):

$$\begin{array}{rrcll} A_n: & f(f(x,y),z) & = & f(x,f(y,z)). & \\ & f(e_j,x) & = & x & \text{for } 1 \leq j \leq n. \\ & f(x,i_j(x)) & = & e_j & \text{for } 1 \leq j \leq n. \end{array}$$

That is, each A_n defines a binary operator f that is associative, and has n left-units and n right-inverses. □

We have tried to complete A_n for $n = 10i$, $1 \leq i \leq 10$, in three theorem provers: OTTER [45], Hiper [7], and Herky, a descendent of *RRL*. Table 1 gives the statistics of these runs. All the run times are collected using a Sun SPARCstation 1 (16-megabyte memory).

In all these runs, the same termination ordering was used, and, as a result, all the three theorem provers produced the same canonical rewrite system for each A_n. (The number of the rules in each canonical system is given in the second column of the table.) OTTER is a well-engineered, fast resolution-based theorem prover implemented in C by Bill McCune at Argonne National Laboratory [45]. OTTER supports a rich set of inference rules, including the Knuth-Bendix completion procedure. Hiper was claimed by its creator Jim Christian as "the fastest completion procedure in the world" [7]. Both Hiper and Herky are written in Common Lisp. For A_{50}, Hiper fails to produce a complete system because the theorem prover ran out of memory. This is not surprising because, according to Christian, Hiper's speedup comes at the price of extra memory consumption—typically five to six times that required for the ordinary version.

Both OTTER and Hiper were implemented with the performance as the main consideration. As a result, in terms of the number of equations processed per second, OTTER is the fastest theorem prover and, in contrast, Herky is the slowest of the three. Since Herky generates far less critical pairs as a result of the numerous critical pair criteria used to identify unnecessary critical pairs as well as different ways of selecting rules to generate critical pairs, Herky can complete A_{100} while both OTTER and Hiper have difficulty in completing A_{60}.

It is worth mentioning that soon after Hyper was implemented in 1988, Christian approached us to run some of his benchmark problems on *RRL* for comparison. At that time, *RRL* did not use discrimination nets and term indexing for representing rules, etc. Hyper performed much better than *RRL* but at the same time, *RRL* did not fare poorly primarily because its good critical pair computation and selection strategies. An older version of *RRL* which does not have implementations of discrimination nets for term-indexing managed to complete A_{60} on a SPARC 10 with 32 MB of memory in about 2200 seconds, and A_{120} on a SPARC 2 with 16 MB of memory.

2.2.1 Detecting Unnecessary Critical Pairs

It has been well known that for testing the confluence of a terminating rewrite system as well as for generating a canonical rewrite system, not all critical pairs need to be explicitly checked for their joinability. Sufficient conditions have been developed that allow us to bypass the joinability check for some critical pairs or even to avoid some superpositions without compromising the Church-Rosser property of the resulting rewrite system (i.e., these conditions are *safe* with the completion procedures). These conditions are known as *critical pair criteria.* The ability to identify redundant critical pairs is important to the effectiveness and efficiency of the completion procedures. For years, researchers have been working in this area, and a number of such criteria have been developed. Among them are the blocked superposition criterion [44], connectedness criterion [55], prime superposition criterion [22], subconnectedness criterion [42], general superposition criterion, and symmetric superposition criterion [71]. More recently, strict and basic superpositions are being investigated in the context of clausal superposition techniques for first-order theorem proving.

Most of the papers on critical pair criteria have focused on the safety of a critical pair criterion. Hardly any published work exists on the important questions of the effectiveness of various criteria, in particular, how many extra computation steps are needed to perform the check for a criterion, and how many computation steps the criterion can save. Even though the primary goal of identifying critical pairs criteria is to improve the performance of completion procedures, not every discovered critical pair criterion is proven to be practically useful because of the cost of performing the check for a criterion. There is a tradeoff between performing criterion check and simply executing the joinability check. In [60], some properties of critical pair criteria with respect to this tradeoff are established: (i) *dynamic safety* with respect to a completion procedure, (ii) *power* to detect a large portion of redundant critical pairs, (iii) *low cost* to perform a critical pair criterion check, and (iv) *high saving* by avoiding more redundant computation.

Several critical pair criteria have been implemented in *RRL* and tested for various examples. We are particularly interested in criteria that can be implemented cost-free, that is, by applying such a criterion is just part of the normalization process. Two criteria, the blocked superposition criterion [22] and the left-composite superposition criterion [60], are known to be such criteria. Both of them are implemented in *RRL*; this fact explains partially why *RRL* generates much less critical pairs than both OTTER and Hyper on Christian's benchmark examples (see example 2.1). For associative-commutative operators, identifying redundant critical pairs becomes all the more important because of numerous unifiers generated by an associative-commutative unification procedure [27]. We consider this to be one of the major strengths of *RRL* because of which it is possible to tackle challenging mathematical problems such as ring commutativity problems. An interested reader may consult [71, 60] for more details.

2.2.2 Term-Indexing Techniques

A key operation in theorem provers based on completion is to compute a normal form of a term with respect to a set of rewrite rules. A normal form of a term is computed by repeatedly applying an applicable rewrite rule to the term. By the use of discrimination nets for term indexing, the efficiency of normalization process can be dramatically improved. Except for Herky, other versions of *RRL* have used only the outermost function symbol of a rewrite rule for discrimination, which has turned out to be quite effective.

A discrimination net is a variant of the trie data structure used to index dictionaries. The basic idea is to partition terms based upon their structure. At the end of each path in the net, there is a linked list of terms or rewrite rules sharing the same structure.

Discrimination nets are used for the following three operations: Given a discrimination net T, a term t, (a) find a rewrite rule r in T such that r can rewrite t; (b) find a list of terms in T that are instances of t; and (c) find a list of terms in T with which t is unifiable. Operation (a) is used in normalization; (b) is used in simplifying rewrite rules; and (c) is used in generating new inferences by computing critical pairs.

Even though we have been able to use the discrimination nets quite effectively, there are still many issues about the use of discrimination nets that should be further explored:

- Are there situations under which it is more efficient to make the search in the discrimination net deterministic (by duplicating some nodes of the net)? Similarly under what conditions, is it better to use nondeterministic search in the net (so that the space is saved)? Currently, all of the three provers OTTER, Hiper, and *RRL*, use nondeterministic search.
- Under what conditions, is it better to maintain several discrimination nets (one for the left-hand sides of rewrite rules, one for subterms in equations and rules, etc.), and under what conditions is it more effective to use a single discrimination tree (the distinction of different items is made at the leaf of each path)? Currently, Hiper uses several discrimination nets, whereas *RRL* uses a single discrimination net.
- Should distinct variables be represented differently in a discrimination net, or is it more effective to consider all variables as a single wild-card symbol? In OTTER, distinction is made among different variables. In Hiper and *RRL*, all variables are treated as a single wild-card symbol.
- Should variable bindings (i.e., substitution) be constructed as discrimination nets are searched through? Under what conditions is it better to separate the matching algorithm into two phases: the first phase checks only function symbol compatibility, and the second

phase checks variable binding consistency? OTTER builds variable bindings along the path; Hiper and *RRL* use a two-phase matching algorithm.

We are also interested in the problem of handling associative and commutative (AC) function symbols in discrimination nets. One crucial assumption in the implementation of discrimination nets is that every function has a fixed arity (the number of arguments). The *flattened* AC terms do not have such a property. That could perhaps be a major obstacle to using discrimination nets for indexing AC terms. In Herky, a preliminary implementation of AC-term indexing has been tried, and its performance is not very satisfactory. Developing an efficient data structure for discarding paths not leading to a match should be explored because of the importance of AC theories in tackling challenging mathematical problems.

2.2.3 Nonterminating Rewrite Rules

Earlier we discussed different options a user is given when an equation cannot be oriented into a terminating rewrite rule by the algorithm implementing *lrpo* and *aclrpo*. To automatically handle equations that cannot be oriented into a terminating rewrite rule, we implemented the technique of "constrainted rewrite rules" in Herky. With this technique, a nonterminating equation is represented as an ordinary rule with a constraint (or condition) that ensures that the left-hand side of the rule is greater than its right-hand side. For instance, the equation $a(M,x) = a(x,x)$ can be represented as the rule $a(x,x) \rightarrow a(M,x)$ with the constraint $x >_{kbo} M$ (where *kbo* denotes the Knuth-Bendix ordering). This idea is a generalization (with *kbo*) of the so-called lexically dependent rewrite rules described in [57, Ch. 8]. After this was implemented in Herky, we learned that a formal account of this idea had appeared in [50] (where lexicographical recursive path ordering (*lrpo*) instead of *kbo* is used in the constraints). *Kbo* seems very suitable for this application for two reasons:

1. It is easy to reduce general constraints to primitive constraints of the form $(x >_{kbo} M)$ instead of $(a(x,x) >_{kbo} a(M,x))$, so that the check can be done inexpensively. This is very important because such tests must be frequently performed.
2. *Kbo* appears to be more flexible than *lrpo* because the user may choose various weight functions and precedence relations.

Our experience in the use of *kbo* has been quite positive. On the other hand, we were not impressed with the performance of constrained rewrite rules generated using *lrpo* as reported in [50]. A further investigation is, however, needed into the issue of an appropriate ordering for constrained rewrite rules in constrained completion procedures: we believe that a choice of an appropriate ordering is crucial for its adequate performance.

2.2.4 Special Completion Procedures

RRL has an implementation of a special completion procedure over the combined theory of the free Abelian groups with the distributivity laws. The approach does not uses an E-matching algorithm for this theory for rewriting or an E-unification algorithm for computing critical pairs. Instead, the concepts of superpositions and critical pairs are extended taking into consideration the rewrite rules of the theory and analyzing their interaction to ensure that the result is likely to be a non-redundant inference. This theory and its subsets are useful for attacking mathematically challenging problems including problems over nonassociative rings, ring commutativity problems, and problems over commutative Thue systems. In fact, this approach is used for the theory of Boolean rings also to obtain a refutational procedure for first-order theorem proving (this is briefly discussed in a later subsection).

For illustration, consider problems over alternative rings. An alternative ring is a special case of nonassociative rings (the associativity of the multiplication $*$ is not present) in which both $(x * y) * y = x * (y * y)$ and $(x * x) * y = x * (x * y)$ hold.

Example 2.2 Consider the following two rules (where (2) is derived from (1) and $(x*y)*y \rightarrow x * (y * y)$):

```
(1)  (x * (y + z)) ---> ((x * y) + (x * z))
(2)  ((x * y) * z) + ((x * z) * y) ---> (x * (y * z)) + (x * (z * y))
```

There are four critical pairs between (1) and (2); if the Abelian groups (for $+$) and the distributivity laws are built in, no critical pairs will be generated from these two rules because each variable in (2) is linear in each product and all critical pairs can be shown to be trivial. In general, the number of critical pairs between a rule r and a distributivity law is linear to the size of the left-hand side of r; using the built-in procedure, the number of generated critical pairs is equal to the number of distinct variables that appear more than once in a product term. □

In Table 2, six theorems about alternative rings are listed; Equations 3, 4, and 5 are the famous Moufang Identities. The definitions of a and f are as follows:

$$
\begin{aligned}
a(x, y, z) &= (x * y) * z + -(x * (y * z)) \\
f(w, x, y, z) &= a(w * x, y, z) + -(x * a(w, y, z)) + -(a(x, y, z) * w).
\end{aligned}
$$

The table also gives some experimental results of running these problems on *RRL*. For comparison, related results from Anantharaman and Hsiang [2] are reported. (We are not aware of other general theorem provers on which these problems have been successfully solved.) The timings of SBR2 were measured on a Sun 3/60 and those of of *RRL* are on a Sun SPARC-station 1 (in seconds). SBR2 proofs used theorems previously proved to obtain proofs of new theorems, whereas all proofs on *RRL* were obtained directly from the axioms about alternative rings.

Many extensions of the Knuth-Bendix completion procedure for special equational theories have been proposed over the years. Most of these extensions have aimed at extending the scope of problems accepted by the completion procedure by building nonterminating or nonconvergent equations into the matching and unification procedures. We get the impression from the literature that most of these approaches have failed to prove any interesting nontrivial theorem in reasonable amount of time. We believe that our approach of carefully identifying relevant critical pair computations from the rewrite rules of the special theory is one of the few attempts of building a theory into the completion procedure with the sole objective of improving the performance of the completion procedure. Our experimental results suggest that this approach is very promising. An interesting subcase is that of commutative monoids, as there appear to be interesting problems in this theory. We are also considering building into *RRL* theories other than the theory of Abelian groups and distributivity laws.

Problem	SBR2 time	SBR2 pairs	SBR2 rules	Herky time	Herky pairs	Herky rules
1. $(x*y)*x = x*(y*x)$	32	67	15	0.58	19	9
2. $a(x,y,z)+a(y,x,z)=0$	48	127	19	0.48	12	8
3. $x*(y*(x*z)) = ((x*y)*x)*z$	2:30:49	452	41	16.08	172	48
4. $((z*x)*y)*x = z*((x*y)*x)$	1:56:36	427	39	15.91	172	48
5. $(x*y)*(z*x) = (x*(y*z))*x$	1:56:09	638	47	16.03	172	48
6. $f(x,x,y,z)=0$	2:07:58	463	39	22.10	190	59

Table 2: Statistics of SBR2 and Herky on alternative ring problems

2.3 Boolean-Ring-based Theorem Proving

RRL supports the Boolean-ring based approach and the clausal superposition approach for proving theorems in full first-order predicate calculus with (or without) equality. The performance of the implementations of these approaches are reasonable; however, they do not, in general, compare well with OTTER [45], PTTP, and similar high-performance theorem provers, because we have not devoted much effort to enhance these techniques. We think the clausal superposition approach is very promising for proving theorems with equality because it has been used to automatically prove group isomorphisms and automorphisms theorems that no other provers (including OTTER) has been able to prove.

For the sake of completeness, we briefly discuss in this subsection the Boolean-ring-based approach. In the next subsection, we briefly review the clausal superposition approach, with an emphasis on contextual rewriting, as that mechanism is also used for theorem proving by induction.

The input to *RRL* is a first-order theory specified by a finite set of axioms that are arbitrary first-order formulas. Large formulas are split into smaller formulas by using properties of the Boolean connectives such as *and*, *or*, and *imply*, and then *Skolemized* to eliminate quantifiers. These Skolemized formulas are converted into polynomial equations by using *exclusive-or*, denoted by $+$, and *conjunction*, denoted by $*$, along with the truth values *false* and *true*, denoted by 0 and 1, respectively. The approach is based on the fact that each formula in the propositional calculus has a unique polynomial representation, as there exists a canonical rewrite system for free Boolean rings. New predicate symbols can be introduced to control the generation of large intermediate formulas in the translation.

The basic steps of the Boolean-ring-based approach toward first-order theorem proving are as follows:

1. make terminating rewrite rules from polynomials using a well-founded order on monomials,
2. simplify polynomials by these rewrite rules (reduction), and
3. derive new polynomials from these rewrite rules (superposition).

If $1 = 0$ is derived, then the input formulas are unsatisfiable (or inconsistent).

RRL supports two methods of the Boolean-ring-based approach: The Gröbner base method developed by Kapur and Narendran [20] and the so-called odd-strategy method developed by Zhang [64]. In the Gröbner base method, a rewrite rule is made from each polynomial such that the left side of the rewrite rule consists of the maximal monomials. New polynomials are generated from any two polynomials by unifying their maximal monomials. Many benchmark problems have been effectively solved using the Gröbner basis method. More details about the Gröbner basis approach can be found in [20]. Some of earlier work on hardware verification work to be mentioned later in the section on applications was done using this method.

In the odd-strategy method, new polynomials are generated from two polynomials only if one of the two polynomials has *odd* number of monomials; only maximal atoms from the odd polynomial are considered for critical pair computation. The odd-strategy is an extension of the N-strategy [12]. One advantage of the odd-strategy over the N-strategy is that there is no need to convert input formulas into clauses before transforming them into polynomials. If all the polynomials are derived from clauses, then the odd-strategy reduces to the N-strategy because in this case, N-rules are the only polynomials with odd number of monomials.

In general, if $a * p_1 + p_2 = 0$, where a is a maximal atom, p_1 and p_2 are two polynomials not containing a, and $a * p_1$ denotes $a * m_1 + \cdots + a * m_n$ for $p_1 = m_1 + \cdots + m_n$, consists of an odd number of monomials, then the new polynomial generated by the odd-superposition from $a * p_1 + p_2 = 0$ and another polynomial $a * q_1 + q_2 = 0$ is $q_2 * (p_1 + p_2) = 0$.

Example 2.3 Suppose $f(x) > g(x) > h(x)$ for any x and the following three polynomials are given:

$$\begin{array}{lrcl} (1) & f(x) * g(x) + g(x) * h(x) + h(x) * f(x) & = & 0 \\ (2) & g(c) + 1 & = & 0 \\ (3) & h(c) + 1 & = & 0 \end{array}$$

In the Gröbner base method, the rule $f(x) * g(x) \rightarrow g(x) * h(x) + h(x) * f(x)$ is made from (1) and $g(c) \rightarrow 1$ is from (2). The new polynomial $f(c) = g(c) * h(c) + h(c) * f(c)$ is then generated by overlapping the left-hand sides of these two rules (i.e., $f(c) * g(c)$), which gives a contradiction after rules corresponding to polynomials (2) and (3) are applied for rewriting.

In the odd-strategy method, (1) is reformulated as $f(x) * (g(x) + h(x)) + g(x) * h(x) = 0$. Because (1) is the only odd polynomial and f does not appear in (2) and (3), no new polynomials will be generated from (1) and (2) (or from (1) and (3)). However, a new polynomial can be generated from (1) and itself: $g(x) * h(x)(g(x) + h(x) + g(x) * h(x)) = 0$, which is equivalent to $g(x) * h(x) = 0$. From $g(x) * h(x) = 0$, (2) and (3), $1 = 0$ can be easily generated. Note that the N-strategy does not work here because there are no N-rules ((1) is not clausal). $\square$

We have not been able to experiment extensively with the two methods; consequently, these implementations are not fast. There is a need to study efficient data structures for polynomials that minimize duplication of atoms. For instance, for the odd-strategy, it is natural to present a polynomial $a * p + q$ recursively by a triple $\langle a, p, q \rangle$. This data structure is efficient for computing odd-superpositions but may not be good for rewriting. A complete implementation of rewriting requires set matching, an NP-complete step (because subsumption can be implementing by rewriting using polynomials) [25]. It might be better to designate only a small subset of rules for rewriting.

2.4 Clausal Superposition Theorem Proving

The so-called clausal superposition method [70] is implemented in RRL to support the second approach for first-order theorem proving. The input for this method is a set of clauses. Every clause is transformed into a conditional rewrite rule (or a finite set of rewrite rules), with a maximal literal being the head of the rule and the negation of the remaining literals being the condition or the premises of the rule. A well-founded ordering on terms and atoms is used to select maximal literals in a clause. Conditional rules are superposed to generate new rules (or clauses). This definition of superposition between two rules subsumes both resolution and paramodulation on the maximal literals of the corresponding two clauses. Resolution and paramodulation can thus be treated uniformly.

Conditional (contextual) rewriting [70] is used to simplify clauses with the rewrite rules made from other clauses. If a rule (clause) gets reduced, the new rule (clause) is kept and the old is thrown away. This rewriting is more powerful than subsumption and demodulation together.

Below we use two examples to illustrate clausal superposition and contextual rewriting.

Example 2.4 If the input includes a clause $p(a)\ or\ \neg q(b)\ or\ (a = b)$, or equivalently, the conditional equation, $p(a)$ **if** $q(b)\ and\ \neg(a = b)$. For making a rewrite rule from a clause, it is better to think of each literal as an equation and a clause as a disjunction of equations. As in the case of equational theories, RRL uses the lexicographic recursive path ordering to compare terms and atoms. For example, the above clause can be considered as

$$(p(a) \Leftrightarrow true)\ or\ (q(b) \Leftrightarrow false)\ or\ (a = b).$$

A maximal equation is chosen as the head of the rule, and the negation of the remaining equations is included in the condition of the rule. For instance, if an ordering on atoms is chosen in which $q(b) > p(a) > b > a$, then the above clause gives the following rule

$$1. \quad q(b) \rightarrow false \textbf{ if } (p(a) \Leftrightarrow false)\ and\ ((a = b) \Leftrightarrow false).$$

The superposition of rule 1 above with the following rule,

$$2. \quad q(b) \rightarrow true \textbf{ if } p(b),$$

is:

$$(p(a) \Leftrightarrow true)\ or\ (a = b)\ or\ (p(b) \Leftrightarrow false).$$

Superposition of rule 1 with the following rule,

$$3. \quad b \rightarrow c \textbf{ if } (a = b),$$

is:

$$(q(c) \Leftrightarrow false)\ or\ (p(a) \Leftrightarrow true)\ or\ (a = b)\ or\ ((a = b) \Leftrightarrow false),$$

which is trivial after simplification. This superposition is equivalent to a paramodulation. □

Example 2.5 We illustrate contextual rewriting by simplifying $\neg istiger(y)\ or\ \neg isanimal(y)$ using rule

$$isanimal(x) \rightarrow true \textbf{ if } istiger(x).$$

To simplify $\neg\ isanimal(y)$, the remaining literal of the clause, $\{\neg\ istiger(y)\}$, is first negated to give $\{istiger(y) = true\}$, which is called the *context* of the literal $\neg\ isanimal(y)$. The left side of the rule, $isanimal(x)$, is matched against $isanimal(y)$ using the substitution $\sigma = \{x/y\}$. The condition of the rule under σ is $istiger(y)$, which is simplified to $true$ by the context, i.e. by $\{istiger(y) = true\}$. So the rule is applicable, and $isanimal(y)$ is reduced to $true$. The clause is simplified to $\neg istiger(y)$.

Note that neither subsumption nor demodulation can simplify the above clause to $\neg istiger(y)$ (which can be obtained by resolution). $\square$

The idea of simplifying one literal while using the remaining literals in a clause as its context comes from Boyer and Moore's theorem prover [6]. A formal analysis of contextual rewriting is given in [63], where it is shown that this rewriting is useful not only for inductive theorem proving, but also for deductive theorem proving when each clause is viewed as a (conditional) rewrite rule.

Experimental results about the clausal superposition method were reported for a number of examples including Schubert's Steamroller problem, SAM's lemma, and problems from set theory. The power of the clausal superposition method is evident from the fact that it can be used to easily prove the three isomorphism theorems in group theory (Wos identified the first (the easiest) of these three as a challenge problem in [57, pp. 138, Test Problem 5]); see [62] for more details. Other theorems about automorphisms of groups, products of groups, and homomorphisms of rings can also be proved without much difficulty. Compared with other methods, the clausal superposition method works particularly well if the input involves non-Horn clauses with equality.

3 Automated Induction in *RRL*

RRL is one of the few theorem provers that not only provides extensive capabilities for theorem proving in equational logics and first-order predicate calculus with equality, but also supports methods for proving formulas by induction on recursively defined data structures. In this section, we discuss two different approaches for inductive theorem proving supported in *RRL*.

A data structure (equivalently a data type) can be specified as a set of functions with proper type (or sort) declaration and a set of equations defining these functions. The approach adopted in *RRL* is to explicitly divide all functions into two disjoint subclasses: *constructor* and *non-constructors*, with the interpretation that the constructors and the equations on constructors define a standard model of the data structure. Every nonconstructor function is assumed to be completely defined over the constructors (called *sufficient completeness*); *RRL* provides algorithms to perform this check in most cases. A (conditional) equation is said to be an *inductive theorem* of the data type if every instance of the equation, which is obtained by substituting the variables in the equation by a ground term containing only constructors, can be proved to be true by equational reasoning. In other words, an inductive property is an "abstraction" of many theorems provable by equational reasoning.

RRL supports two different approaches to proving inductive properties:

- *proof by consistency* and

- *proof by induction based on well-founded orderings.*

In the next two subsections, the implementations of these approaches in *RRL* are briefly discussed.

3.1 Proof by Consistency

As the name implies, the proof-by-consistency approach is the opposite of the proof-by-contradiction approach. The conjecture being proved is assumed to be true, and it is checked whether the conjecture with the hypotheses is consistent; absence of an inconsistency implies consistency. The inconsistency in the proof-by-consistency approach is detected by checking whether certain constructor values declared to be *unequal* have been made equal as a result of a conjecture being assumed. So it is important that the hypotheses are consistent.

Inconsistency check can be performed in many different ways. One method is to use completion and incrementally detect whether any of the new rules generated as a result of possible interactions between the conjecture and the hypotheses implies unequal constructor ground terms being made equal. In case an inconsistency is generated, *RRL* backtracks and notifies that the conjecture being proved is not an inductive theorem. Otherwise, if no inconsistency is detected and the completion procedure terminates, then the conjecture is proved. This approach, which is also popularly known as the *inductionless induction method*, is radically different from the conventional induction method in the sense that in the conventional induction methods, induction is done explicitly on a well-founded ordering by considering the basis step and the inductive step. Even though a well-founded ordering used to orient rules into equations is being implicitly used, basis and inductive steps are being generated indirectly through the process of critical pair generation.

Two inductionless induction methods, *ground-reducibility* and *test set*, have been implemented in *RRL*. The ground-reducibility (also called *quasi-reducibility* or *inductive reducibility*) method was proposed in [18]. A term is *ground-reducible* if and only if every ground instance of this term is reducible. The consistency check can be shown to be equivalent to the property that the left side of rewrite rules generated during completion is ground-reducible. This method has an advantage over earlier methods proposed by Musser [46], Goguen [10], and Huet and Hullot [17] as constructors need not be free. An efficient method to decide the ground reducibility of a term in left-linear systems (the occurrence of each variable in the left side of each rule is unique) is implemented in *RRL* [28].

In [30], we discussed a more efficient method for checking consistency based on the concept of *test set*. Test sets were developed in [28] to check whether a nonconstructor function is completely specified by a rewrite system. A *test set* is a finite set of terms that describes the equivalence classes of constructor ground terms defined by equations defining constructor functions. For an equational theory with an associated canonical rewriting system, each equivalence class can be represented by a canonical element that is the normal form of every element in the equivalence class with respect to the rewriting system. During completion, whenever a new rule is generated, inconsistency check is performed by analyzing the effect of the new rule on the test set. A new test set is computed, and its equivalence with the old test set is checked. If equivalence test fails, an inconsistency is reported leading to the conclusion that the conjecture is not true.

Example 3.1 The test set method is illustrated by using a simple example over integers. Integers are generated by nonfree constructors 0, *suc* and *pre* together with the following rewrite rules:

$$\begin{array}{lrcl} 1. & suc(pre(x)) & \rightarrow & x, \\ 2. & pre(suc(x)) & \rightarrow & x. \end{array}$$

The set of irreducible constructor ground terms is $\{0, suc^i(0), pre^j(0) \mid i > 0, j > 0\}$. That is, each integers is represented by 0, or $suc^i(0)$, $i > 0$, for positive integers, or $pre^j(0)$, $j > 0$, for negative integers. The test set for this example is $\{0, suc(0), suc(suc(x)), pre(0), pre(pre(y))\}$. Inconsistency in this case would mean two distinct integers being made equivalent.

A unary function *neg* is defined on integers, giving the negative of its argument.

$$\begin{array}{lrcl} 3. & neg(0) & \rightarrow & 0, \\ 4. & neg(suc(x)) & \rightarrow & pre(neg(x)). \end{array}$$

One may get the impression that the above definition of *neg* is not complete because the $pre(x)$ case is not being explicitly considered, even though it is sufficiently complete. Completion can be performed to check the sufficient completeness of *neg*. Using rules 1 and 4, a new rule is generated:

$$5. \quad pre(neg(pre(x))) \quad \rightarrow \quad neg(x).$$

From rules 2 and 5, another rule is generated:

$$6. \quad neg(pre(x)) \quad \rightarrow \quad suc(neg(x)),$$

which deletes rule 5. Rules $\{1, 2, 3, 4, 6\}$ constitute a canonical rewriting system.

To check whether $neg(neg(x)) = x$ is an inductive theorem, we add the conjecture as another rule:

$$7. \quad neg(neg(x)) \quad \rightarrow \quad x.$$

It turns out that the rule set $\{1, 2, 3, 4, 6, 7\}$ is a canonical rewriting system. Since its test set is the same as the original test set, the conjecture is indeed a theorem by induction. Critical pair computation in this case essentially performs basis step and two induction steps for the induction variable x, and verifies that each of these three subgoals follow from the definition of *neg* and the induction hypothesis.

Now consider another conjecture that is obviously not a theorem,

$$neg(neg(neg(x))) = x,$$

since the equality does not hold if x is different from 0 (we are not assuming $neg(neg(x)) = x$ is in the system). If we add the conjecture as a rule,

$$8. \quad neg(neg(neg(x))) \quad \rightarrow \quad x,$$

we get new rules from rules 4 and 8:

$$9. \quad pre(x) \quad \rightarrow \quad suc(x).$$

Rule 9 reduces the term $pre(pre(y))$ in the test set, which means that at least two distinct irreducible constructor ground terms were made equivalent by the conjecture. The new test set, $\{0, suc(0), suc(suc(x))\}$, is not equivalent to the original test set. This implies that the conjecture is not a theorem. By instantiating the term reduced in the test set, we also get a counterexample, $pre(pre(0))$, to the conjecture. In fact, we obtain an infinite family of counterexamples. $\square$

One of the major advantages of the proof by consistency approach is that it is a semi-decision procedure for determining whether a conjecture is **not** an inductive theorem. For a false conjecture, a counterexample can also be determined using this approach. The approach works quite well on simple examples, especially when functions are defined using constructors in primitive recursion style. For such definitions, it is often possible to generate a canonical rewriting system. The approach, if it works, does not need much user interaction.

The proof by consistency approach is, however, not widely applicable. When functions are not defined directly using constructors and are instead defined using other already defined functions (see the *gcd* example below), two difficulties may arise in its applicability. First, it may not be possible to obtain a finite canonical rewriting system (or even a ground confluent rewriting system) from such function definitions, as one simply may not exist. Second, even if a canonical rewriting system for the definitions exists, it may not be possible to generate a finite canonical rewriting system (or even a ground confluent rewriting system) when the conjecture being proved is also included.

3.2 Cover Set Induction Method

RRL supports an approach for automating proofs by induction based on explicit induction techniques. The method, called the *cover set* method, is reported in [72]. It is closely related to Boyer and Moore's approach. Using the cover set method and contextual rewriting, *RRL* has been successfully used to prove many nontrivial theorems including the unique prime factorization theorem of numbers, Chinese remainder theorem, and Ramsey's theorem.

Example 3.2 Below, we illustrate the cover set method using an example over natural numbers, Nat, whose constructors are 0 and suc. The function $+$ is defined on Nat in a primitive recursive style:

$$\begin{array}{lll} 1. & x + 0 & = & x, \\ 2. & x + suc(y) & = & suc(y). \end{array}$$

It is easy to see that because of the primitive recursive definition, $+$ is completely defined. Another function gcd can be defined on Nat as follows. Notice that this definition is not given using the constructors; instead, it makes use of the function $+$.

$$\begin{array}{llll} 3. & gcd(x, 0) & = & x, \\ 4. & gcd(0, x) & = & x, \\ 5. & gcd(x, x + y) & = & gcd(x, y), \\ 6. & gcd(x + y, y) & = & gcd(x, y). \end{array}$$

The gcd definition can be proved to be complete.

In order to prove theorems about gcd using induction, it turns out that structural induction rule does not help, because the proof subgoals produced by these theorems may not be provable by equational inference. Let us check, for instance, whether

$$gcd(x, y) = gcd(y, x)$$

using structural induction. Without any loss of generality, we can induct on x. One subgoal, $gcd(0, y) = gcd(y, 0)$ follows easily by equational inference from the above definitions. The second subgoal is

$$gcd(x, y) = gcd(y, x) \Rightarrow gcd(suc(x), y) = gcd(y, suc(x)),$$

which is not provable by equational reasoning from the above definitions. □

The test set method does not work either for the gcd example, because the above equational theory constituted from the definitions of $+$ and gcd does not have an equivalent finite canonical (or ground-confluent) rewriting system.

In the cover set induction method, the induction scheme is developed from the definitions of the functions appearing in a conjecture being proved. In contrast to the structural induction scheme, there is no fixed inductive inference rule for a data structure. Different types of definitions of functions can lead to different inductive inference rules for the same data structure. The cover set method is similar to Boyer and Moore's approach in which inductive inference rules are designed from recursive definitions of functions. Because the induction scheme is based on the structure of the defining axioms, it often yields subgoals that are provable by rewriting.

For the above example, for properties about $+$, the cover set method proposes an inductive inference rule that is the same as the structural induction rule using constructors 0 and suc, but for properties in which gcd appears, the inductive inference rule could be stated using the function symbol $+$. For example, to prove $gcd(x, y) = gcd(y, x)$, a cover set for gcd is obtained from its definition:

$$\langle (0, x), \emptyset, true \rangle, \langle (x, 0), \emptyset, true \rangle, \langle (x + y, y), (x, y), true \rangle, \langle (x, x + y), (x, y), true \rangle.$$

From the above cover set, RRL generates the following inductive inference rule for natural numbers:

$$\frac{\begin{array}{ccc} & P(x,0) & \\ & P(0,y) & \\ P(x,y) & \Rightarrow & P(x+y,y) \\ P(x,y) & \Rightarrow & P(x,x+y) \end{array}}{P(x,y)}$$

where P is a formula with two free variables ranging over the natural numbers. With this inductive inference rule, the conjecture $gcd(x, y) = gcd(y, x)$ can be easily proved using the rewrite rules corresponding the definitions of $+$ and gcd given above as Equations 1 to 6.

A cover set is, in general, a finite set of 3-tuples: The first component of a 3-tuple, which could also be a n-tuple, $n \geq 1$, depending upon the number of variables involved in the induction, corresponds to the substitution on induction variables to generate a subgoal; the second component, which is a set of n-tuples, corresponds to the substitution used to generate induction hypotheses for that subgoal; and the third component corresponds to the condition under which the subgoal needs to be proved. If the second component of a 3-tuple is empty, that case corresponds to a basis step of the induction scheme. Often, the condition is true. For a well-founded ordering, a cover set must have the following properties:

- (**soundness**) For each instance of a 3-tuple in the cover set obtained by substituting each variable of the 3-tuple by a constructor ground term, if the condition of the instantiated 3-tuple is satisfied, the second component of the instantiated 3-tuple should be smaller than the first component in a well-founded order.
- (**completeness**) Every ground term (or n-tuple of ground terms) representing the values (or n-tuples of values) of the data type is equal to an instance of the first component of some 3-tuple in the cover set for which the condition instantiates to true.

RRL generates cover sets from the rewrite rules defining functions, or alternatively, the user can specify a cover set to be used for a particular occurrence of a function. The soundness property of a cover set follows if it is generated from terminating rewrite rules associated with definitions; the completeness property follows if it is generated from rewrite rules associated with a complete definition.

RRL supports an algorithm for checking the sufficient completeness property of functions definitions based on test sets. For instance, the completeness of $+$ can be easily checked by *RRL*, thus the cover set generated from the definition of $+$ is guaranteed to be complete. However, the completeness of gcd requires proving that the function $+$ is onto which cannot be done using the sufficient completeness algorithm. It might be possible to prove the onto-ness property of $+$ by induction.

Although the cover set method is quite powerful, it is not easy to use by an inexperienced user for obtaining proofs. The method is primarily user-driven, as it requires considerable user assistance in the form of key lemmas needed to push a proof of a theorem through *RRL*. This is despite the fact that some of the lemmas are generated automatically using generalization and other heuristics implemented in *RRL*. Further, the order in which lemmas are proved may sometimes affect the proof generation process. User interaction is also needed in converting the problem formulation and lemmas into conditional rewrite rules with the termination property. For an inexperienced user, a serious weakness of the cover set method is that it does not provide much useful information when it fails. In particular, it is not clear from the generated output whether the conjecture being proved is false or a proof of the conjecture is likely to need additional lemmas, and if the latter, how those lemmas can be arrived at. Extracting useful relevant information from failed proof attempts to generate candidates for lemmas requires a lot of expertise, and is an important research topic in automating theorem proving by induction.

3.3 Combining Proof-by-Consistency and Cover-Set Induction

The automation of proofs by induction relies on an appropriate induction scheme that constitutes a good selection of variables to perform induction on, subgoals to be considered, and for each subgoal an appropriate choice of induction hypotheses. In the implementation of the cover set method in *RRL*, corresponding to each element in a cover set, a subgoal, which is a conditional equation, is generated. For an induction step, induction hypotheses are made part of the condition in the conditional equation. In an induction hypothesis, no distinction is made between a free variable or an induction variable. The main reason for this decision is simplicity and ease of automation; otherwise, if free variables are assumed to be universally quantified in an induction hypothesis, one has to find appropriate instantiations for these existentially quantified variables to satisfy the condition.

In contrast, in the proof by consistency method, the subcases and induction hypotheses are not explicitly given, but are instead obtained from an interaction of the conjecture with the definitions of functions involved in the conjecture. In that approach, free variables get instantiated as the need arises.

Example 3.3 Suppose, for a simple example, $E = \{0 + y = y,\ suc(x) + y = suc(x + y)\}$, consisting of a primitive recursive definition of $+$. A goal is to prove the associativity of $+$, that is, $(x + y) + z = x + (y + z)$. The subgoals due to the cover set are (assuming that x is the variable selected for induction)

1. $(0+y)+z = 0+(y+z)$,
2. $(x+y)+z = x+(y+z) \Rightarrow (suc(x)+y)+z = suc(x)+(y+z)$.

Note that y and z are the same in the induction hypothesis as well as the subgoal. All the variables in the two subgoals are assumed to be universally quantified.

This way of using the induction hypothesis is different from a typical hand proof. For the induction step, a hand proof consists of proving

$$\forall x \forall y \forall z[(\forall y' \forall z'(x+y')+z' = x+(y'+z')) \Rightarrow (suc(x)+y)+z = suc(x)+(y+z)], \quad (1)$$

It should be noted that this equation is different from the second subgoal generated in the cover set induction method because the hypothesis above has universally quantified variables y' and z' different from y and z of the subgoal.

Now let us illustrate how the induction step is set up in the proof by consistency approach and what induction hypothesis is used. Three rewrite rules are made from the definition of $+$ and the conjecture, that is,

$$\begin{aligned} 0+y &\rightarrow y && (2)\\ suc(x)+y &\rightarrow suc(x+y) && (3)\\ (x+y)+z &\rightarrow x+(y+z). && (4) \end{aligned}$$

The superposition between the left sides of (2) and (4) (i.e., unifying $x+y$ of (4) with the left side of (2) generates $y_1+z_1 = 0+(y_1+z_1)$, which corresponds to the basis case of the proof (the first goal in the cover set method). The superposition between the left sides of (3) and (4) generates $suc(x_2+y_2)+z_2 = suc(x_2)+(y_2+z_2)$, which corresponds to the inductive case of the proof. The proof is completed by showing that

$$E \cup \{(x+y)+z = x+(y+z)\} \vdash suc(x_2+y_2)+z_2 = suc(x_2)+(y_2+z_2). \quad (5)$$

In the above formula, distinct variable names denote distinct variables, and every variable is assumed to be universally quantified. The induction hypothesis $((x+y)+z = x+(y+z))$ does not share any variable with the subgoal $suc(x_2+y_2)+z_2 = suc(x_2)+(y_2+z_2)$.[1] □

The induction hypotheses used in the cover set method are weaker than the corresponding induction hypotheses in the proof by consistency method. Because of this, we call the cover set method as using the *instantiated induction hypothesis* technique, and the proof by consistency as using the *quantified induction hypothesis* technique.

The instantiated hypothesis technique has an advantage that it releases the theorem prover from the burden of handling quantifiers in the condition of a conditional equation. However, it limits the use of the induction hypotheses by fixing the values of the free variables in them. In some cases, some special or several different instances of an induction hypothesis are needed to complete a proof. With the instantiated hypothesis technique, these instantiations will have to be given explicitly through some hint mechanism or a user-defined induction scheme; this approach is adopted, for instance, in the Boyer-Moore theorem prover.

[1] The soundness of this proof is ensured by the fact that $(x+y)+z = x+(y+z)$ can apply only to (an equation simplified from) $suc(x_2+y_2)+z_2 = suc(x_2)+(y_2+z_2)$, not to $(suc(x_2)+y_2)+z_2 = suc(x_2)+(y_2+z_2)$.

The quantified hypothesis technique is used nicely in the proof by consistency approach, in which proper instantiations for free variables in an induction hypothesis are found by matching the rewrite rule corresponding to the induction hypothesis against a subgoal. One, however, loses flexibility in the use of an induction hypothesis. An induction hypothesis can be used only from left to right. Using the instantiated hypothesis technique, in contrast, offers considerable flexibility in the sense that induction hypotheses as well as lemmas can be used in either direction.

We implemented in *RRL* the quantified hypothesis technique in conjunction with the cover set method. We compared the performance of the two implementations of the cover set method—one using the instantiated hypothesis technique and the other using the quantified hypothesis technique, on the proofs of some theorems including the pigeonhole principle in set theory.

One of the major advantages of the quantified hypothesis technique over the instantiated hypothesis technique is that proofs needing special or several different instances of an induction hypothesis can be automated using the quantified hypothesis technique. In contrast, the instantiated hypothesis technique invariably require considerable user assistance. Further, the number of additional lemmas needed with the instantiated hypothesis technique are much more than in the case when the quantified hypothesis technique is used. With the instantiated hypothesis technique, more subgoals may be generated if many instances of an induction hypothesis are used. In a proof of the pigeonhole theorem done with *RRL* using the cover set method, the instantiated hypothesis technique generated six subgoals (excluding all the lemmas), while the quantified hypothesis technique generates only two. Often, more subgoals are generated, more computer time *RRL* takes to finish a proof. There are, however, examples in which it is an advantage to have more subgoals, as doing case analysis is helpful when proofs need additional lemmas.

An interested reader can find a more detailed comparison of the two techniques in [39]. Further detailed investigations are necessary to examine the use of the quantified hypothesis technique in the cover set method.

4 Applications

One of the main ideas behind the development of *RRL* has been to have access to an environment in which one can experiment with ideas and algorithms in automated reasoning based on rewriting. *RRL* has been very useful for this application; a strong evidence is the fact that we and our colleagues have written over thirty research papers related to *RRL*.

Another important use of *RRL* has been as a teaching aid. Since its development in 1984, *RRL* has also been used in a course on theorem proving based on rewrite techniques taught by us as well as other colleagues. Having access to *RRL* provides students with an opportunity to try methods and approaches discussed in the course.

Below, we discuss other applications of *RRL*, in particular its use as a theorem prover in specification and verification analysis of hardware and software, as well as for attacking mathematically challenging problems.

4.1 Hardware Verification

In 1986, *RRL* was successfully used at GECRD for hardware verification. *RRL* was integrated into a VHSIC Hardware Description Language (VHDL) workstation environment. For small combinational and sequential circuits, as well as leaf cells of a bit-serial compiler, *RRL* was demonstrated to be effective for automatically proving that the behavioral specification written as a first-order formula was realized by a structural specification of a circuit written also as a first-order formula and generated from the circuit layout. More details can be found in [47].

In 1987, *RRL* was used to analyze the input-output behavior of a SOBEL image-processing chip being designed at Research Triangle Institute, Raleigh, North Carolina. Our colleague Paliath Narendran was provided with a VHDL description of the chip, which had about 10,000 transistors, implementing Sobel's edge detection algorithm for image processing. The VHDL description of the chip was used to manually generate its first-order structural specification. The behavioral specification was generated from a description of the algorithm in a book on image processing. Using first-order theorem-proving capabilities of *RRL*, Narendran and Stillman detected two bugs in the chip design, one of which was not known to the chip designers despite extensive testing and simulation of the chip. See [48] for more details.

Recently, Hua and Zhang enriched VHDL to provide a common language for specifying, designing, and verifying hardware circuits. Based on a denotational semantics of VHDL proposed by Hua in his thesis [13], a prototype translator was implemented that automatically translates VHDL specifications into algebraic specifications accepted by *RRL*. Hua and Zhang have successfully verified using *RRL* the correctness of various circuit designs, including an ALU, a generic decoder, a traffic light controller and a systolic array multiplier [15].

4.2 Software Verification and the Tecton System

Since 1985, *RRL* has been used to prove properties of algorithms and data structures in which specifications are written as equational axioms. For example, alternating bit-protocol was analyzed, and stacks and other data structures were established.

In [67], Zhang, Guha, and Hua suggested a technique that combines the Floyd-Hoare axiomatic approach for program verification with the cover set induction method. Several sorting algorithms are used to demonstrate that the technique can help in verifying software mechanically.

Since 1990, Kapur, in collaboration with David Musser, has been developing the *Tecton* proof system. Tecton (Greek for "builder") is a methodology and tool set for formal specification and verification of computational systems (both hardware designs and software) [1, 23, 24], in which *RRL* is the main inference engine. In formulating the goals of Tecton and designing its tools, we are seeking to combine many of the key advances in specification and proof technology. We also seek to simplify the use of formal methods, making them more accessible to nonexperts and more easily applicable to nontrivial computational systems.

Tecton is an experimental tool for constructing proofs of first-order logic formulas and of program specifications expressed using formulas in Hoare's axiomatic proof formalism. Tecton represents and manages proofs internally in flexible structures called proof forests, allowing records of multiple complete or incomplete proof attempts to be retained (an extension of the proof forest notion of [9]). Tecton uses a hypertext system, KMS (Knowledge Management

System), for the structured external display format to present proofs to the user, using tables, graphics, and hypertext links.

The inference engine of Tecton has been extended to support in the reduction mechanism of *RRL*, a decision procedure for a subclass of *Presburger arithmetic* with uninterpreted function symbols. Integers and operations and relations on them are used extensively in the mathematics of computer programming. The built-in procedure eliminates the need to explicitly state some axioms for integers such as transitivity axioms for inequality relations. The proofs of verification conditions obtained using the linear arithmetic procedure are shorter and more compact than those obtained without the use of the linear arithmetic procedure.

Tecton has been used to prove properties of simple (but efficient) programs on integers, equational programs for sorting and searching, and simple programs using abstract data types. More details about the Tecton system can be found in [24, 23].

4.3 Attacking Challenge Problems

In 1988, *RRL* was successfully used to attack the so-called ring commutativity problems, considered a challenge for theorem provers [56]. For the theorem that an associative ring is commutative if every element x satisfies $x^3 = x$, an automatic proof was obtained using *RRL* in less than two minutes on a Symbolics machine in 1988. Previous attempts to solve this problem using the completion-procedure-based approach required over ten hours of computing time [52]; see [53] for another proof using resolution paramodulation and demodulation on an earlier Argonne theorem prover. With some special heuristics developed for the more general family of problems for any exponent $n > 1$, namely, that an associative ring is commutative if every element x satisfies $x^n = x$, *RRL* takes 5 seconds for the case when $n = 3$, 70 seconds for $n = 4$, and 270 seconds for $n = 6$; see also [54]. To our knowledge, *RRL* was used to generate the first computer proof of this theorem for $n = 6$ [71]. Using algebraic techniques, Zhang was able to prove this theorem for many even values of n [59].

The key idea that helped in obtaining fast proofs of ring commutativity problems was that very many inferences in completion-based-approaches were unnecessary and redundant in the presence of associative-commutative theories. We emphasize the importance not only of implementing primitive steps efficiently, but also of identifying redundant inferences inexpensively so they can be skipped without much penalty.

In 1987–88, *RRL* was also used to prove the equivalence of different nonclassical axiomatizations of free groups developed by Higman and Neumann to the classical three-law characterization [37].

Herky was recently used by Zhang to win a competition on equality problems announced in 1990 by Ross Overbeek of Argonne National Laboratory. Herky was used to solve 9 out of 10 equality problems proposed in the competition (the tenth problem is an open question in algebra). We are not aware of any theorem prover that has been successful in solving these 9 problems. Details about the competition as well as Herky's solution to the problems can be found in [65]. The special completion procedure in *RRL* has also been used to prove over 30 difficult theorems in the theory of alternative rings, a special kind of nonassociative rings [66].

The cover set induction method was developed and implemented in *RRL* in 1987 as a part of Zhang's dissertation work. Since then, many interesting challenging problems proposed in theorem proving by induction have also been successfully attempted in *RRL*. The problems include the unique prime factorization theorem for numbers, Ramsey's theorem, the Chinese remainder theorem, and Gilbreath's card trick. Preliminary comparison of proofs obtained on *RRL* with proofs of these problems on Boyer-Moore's theorem prover suggest that *RRL* proofs require less user interaction and fewer user-supplied intermediate lemmas.

Acknowledgments: A number of people have contributed to this project in many different ways. We particularly acknowledge G. Sivakumar, D. Musser, P. Narendran, D. Cyrluk, X. Nie, G.H. Hua, and R. Harris for their contributions. Thanks also to R. Overbeek, E. Lusk, W. McCune, L. Wos, W. Pase, R. Boyer, J. Rushby, D. Lankford, M. Stickel, and N. Dershowitz for encouragement and support of the *RRL* project.

References

[1] Agarwal, R., Kapur, D., Musser, D. R., and Nie, X., Tecton proof system. In: Proc. *Fourth International Conference on Rewriting Techniques and Applications,* Milan, Italy, 1991.

[2] Anantharaman, S., and Hsiang, J., Automated proofs of the Moufang identities in alternative rings, *J. Automated Reasoning* 6 (1990): 79–109.

[3] Basin, D., and Kaufmann, M., The Boyer-Moore prover and Nuprl: An experimental comparison. In: Proc. *BRA Logical Frameworks Workshop*, 1990.

[4] Benanav, D., Kapur, D., and Narendran, P., Complexity of matching problems, *J. Symbolic Computation* 3 (1987) 203–216.

[5] Boyer, R. S., and Moore, J S., *A computational logic.* New York: Academic Press, 1979.

[6] Boyer, R. S., and Moore, J S., *A computational logic handbook.* New York: Academic Press, 1988.

[7] Christian, J., Fast Knuth-Bendix completion: Summary. In: Proc. *Third International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 355, Springer-Verlag, 1989, pp. 548–510.

[8] Dershowitz, N., Termination of rewriting. *J. Symbolic Computation* 3 (1987) 69–116.

[9] Erickson, R. W., and Musser, D. R., The AFFIRM theorem prover: Proof forests and management of large proofs. In: Proc. *7th International Conference on Automated Deduction,* 1980.

[10] Goguen, J. A., How to prove algebraic inductive hypotheses without induction. In: Proc. *Fifth Conference on Automated Deduction*, W. Bibel and R. Kowalski (eds.), Lecture Notes in Computer Science, Vol. 87, Springer-Verlag, 1980, pp. 356–373.

[11] Guttag, J. V., Kapur, D., and Musser, D. R., eds., In: Proc. *NSF Workshop on the Rewrite Rule Laboratory*, Sept. 6–9, 1983.

[12] Hsiang, J., Refutational theorem proving using term-rewriting systems. *Artificial Intelligence* 25 (1985) 255–300.

[13] Hua, G. X., Formal Verification of Circuit Designs in VHDL. Ph.D thesis, Department of Computer Science, The University of Iowa, 1992.

[14] Hua, G. X., and Zhang, H., Axiomatic semantics of a hardware specification language. In Proc. *Second IEEE Great Lakes Symposium on VLSI Design*, Kalamazoo, Michigan, 1992, pp. 183–190.

[15] Hua, G. X., and Zhang, H., Formal semantics of VHDL for verification of circuit designs. In: Proc. *IEEE International Conference on Circuit Designs*, 1993.

[16] Hua, G. X., and Zhang, H., FRI: Failure Resistant Induction in RRL. In: Proc. *1992 International Conference of Automated Deduction*, D. Kapur (ed.), Lecture Notes in Artificial Intelligence, Vol. 607, Springer-Verlag, 1992, pp. 691–695.

[17] Huet, G., and Hullot, J. M., Proofs by induction in equational theories with constructors. In: *21st IEEE Symposium on Foundations of Computer Science*, Syracuse, N.Y., 1980, pp. 96–107.

[18] Jouannaud, J., and Kounalis, E., Automatic proofs by induction in equational theories without constructors. In: Proc. *Symposium on Logic in Computer Science*, 1986, pp. 358–366.

[19] Knuth, D., and Bendix, P., Simple word problems in universal algebras. In: Computational Problems in Abstract Algebra, Leech (ed.), New York: Pergamon Press, 1970, pp. 263–297.

[20] Kapur, D., and Narendran, P., An equational approach to theorem proving in first-order predicate calculus. In: Proc. *9th IJCAI*, 1985.

[21] Kapur, D., and Musser, D. R., Proof by consistency. *Artificial Intelligence* 31 (1984) 125–157.

[22] Kapur, D., Musser, D. R., and Narendran, P., Only prime superpositions need be considered for the Knuth-Bendix completion procedure. *J. Symbolic Computation* 4 (1988).

[23] Kapur, D., and Musser, D. R. (1992): Tecton: A Framework for Specifying and Verifying Generic System Components. Invited talk at *TPCD Conf. 1992 (Theorem Provers in Circuit Design)*, University of Nijmegen, The Netherlands, June 22–24, 1992.

[24] Kapur, D., Musser, D. R., and Nie, X., Tecton proof system. In: In: Proc. *Workshop on Formal Methods in Databases and Software Engineering,* Alagar, Lakshmanan, and Sadri (eds.), Workshop in Computing Series, Springer-Verlag, 1992, pp. 54–79.

[25] Kapur, D., and Narendran, P., Matching, unification and complexity, *SIGSAM Bulletin*, 1987.

[26] Kapur, D., and Narendran, P., Complexity of unification problems with associative-commutative operators, *J. Automated Reasoning* 9 (1992) 261–288.

[27] Kapur, D., and Narendran, P., Double-exponential complexity of computing a complete set of AC-unifiers. In: Proc. *Logic in Computer Science*, Santa Cruz, June 1992.

[28] Kapur, D., Narendran, P., Rosenkrantz, D., and Zhang, H., Sufficient-completeness, quasi-reducibility and their complexity, *Acta Informatica,* 28 (1991) 311–350.

[29] Kapur, D., Narendran, P., and Zhang, H., On sufficient completeness and related properties of term rewriting systems, *Acta Informatica* 24, Fasc. 4 (August 1987) 395–416.

[30] Kapur, D., Narendran, P., and Zhang, H., Proof by induction using test sets. In: Proc. *Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 230, Springer-Verlag, New York, 1986, pp. 99–117.

[31] Kapur, D., Narendran, P., and Zhang, H., Automating inductionless induction by test sets, *J. Symbolic Computation* 11 (1991) 83–111.

[32] Kapur, D., and Sivakumar, G., Architecture of and experiments with RRL, a Rewrite Rule Laboratory. In: Reference [11], pp. 33–56.

[33] Kapur, D., Sivakumar, G., and Zhang, H., RRL: A Rewrite Rule Laboratory. In: Proc. *Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 230, Springer-Verlag, Berlin, 1986, pp. 692–693.

[34] Kapur, D., Sivakumar, G., and Zhang, H., A new method for proving termination of AC-rewrite systems. In: Proc. *Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, 1990.

[35] Kapur, D., and Zhang, H., *RRL: A Rewrite Rule Laboratory – User's Manual.* GE Corporate Research and Development Report, Schenectady, N.Y., April 1987. A revised version appears as Report 89-03, Dept. of Computer Science, The University of Iowa.

[36] Kapur, D., and Zhang, H., RRL: a Rewrite Rule Laboratory. In: Proc. *Ninth International Conference on Automated Deduction*, E. Lusk and R. Overbeek (eds.), Lecture Notes in Computer Science, Vol. 310, Springer-Verlag, Berlin, 1988, pp. 768–769.

[37] Kapur, D., and Zhang, H., Proving equivalence of different axiomatizations of free groups, *J. Automated Reasoning* 4 (1988) 331–352.

[38] Kapur, D., and Zhang, H., An overview of RRL: Rewrite Rule Laboratory. In: Proc. *Third International Conference on Rewriting Techniques and Its Applications*, Lecture Notes in Computer Science, Vol. 355, Springer-Verlag, Berlin, 1989, pp. 513–529.

[39] Kapur, D., and Zhang, H., Automating induction: Explicit vs. inductionless. In: Proc. *Third International Symposium on Artificial Intelligence and Mathematics,* Fort Lauderdale, Florida, Jan. 2–5, 1994.

[40] Kaufmann, M., An interactive enhancement to the Boyer-Moore theorem prover. In: Proc. *Ninth International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 310, Springer-Verlag, Berlin, 1988, pp. 735–736.

[41] Knuth, D. E., and Bendix, P. B., Simple word problems in universal algebras. In: *Computational Problems in Abstract Algebras,* J. Leech (ed.), Pergamon Press, 1970, pp. 263–297.

[42] Küchlin, W., Inductive completion by ground proof transformation. In: *Rewriting Techniques: Resolution of Equations in Algebraic Structures*, II, H. Aït-Kaci and M. Nivat (eds.), Academic Press, New York, 1989, pp. 211–244.

[43] Lankford, D. S., and Ballantyne, A. M., Decision Procedures for Simple Equational Theories with Commutative-Associative Axioms: Complete Sets of Commutative-Associative Reductions. Report ATP-39, Dept. of Math. and Computer Science, University of Texas, Austin, Texas, 1977.

[44] Lankford, D. S., and Ballantyne, A. M., The refutational completeness of blocked permutative narrowing and resolution. In: Proc. *Fourth Conference on Automated Deduction*, 1979.

[45] McCune, W., OTTER 2.0 Users Guide. ANL-90/9, Argonne National Laboratory, Argonne, Ill., 1990.

[46] Musser, D. R., On proving inductive properties of abstract data types. In: Proc. *Seventh Principles of Programming Languages*, 1980.

[47] Narendran, P., and Stillman, J., Hardware verification in the Interactive VHDL Workstation. In: *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam (eds.), Kluwer Academic Publishers, Dordrecht, 1988, pp. 217–235.

[48] Narendran, P., and Stillman, J., Formal verification of the Sobel image processing chip. In: Proc. *Design Automation Conference*, 1988.

[49] Overbeek, R., A proposal for a competition. Argonne National Laboratory, Argonne, Ill., 1990.

[50] Peterson, G. L., Complete sets of reductions with constraints. In: Proc. *Tenth International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 449, Springer-Verlag, Berlin, 1990, pp. 381–395

[51] Peterson, G. L., and Stickel, M. E., Complete sets of reductions for some equational theories, *J. ACM* 28 (1981) 233–264.

[52] Stickel, M. E., A case study of theorem proving by the Knuth-Bendix method: Discovering that $x^3 = x$ implies ring commutativity. In: Proc. *Seventh Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 170, Springer-Verlag, Berlin, 1984, pp. 248–258.

[53] eroff, R. L., Canonicalization and Demodulation. ANL-81-6, Argonne National Laboratory, Argonne, Ill., 1981.

[54] Wang, T. C., Case studies of Z-module reasoning: Proving benchmark theorems for ring theory. *J. Automated Reasoning* 3 (1987) 437–451.

[55] Winkler, F., and Buchberger, B., A criterion for eliminating unnecessary reductions in the Knuth-Bendix algorithm. In: Proc. *Colloquium on Algebra, Combinatorics and Logic in Computer Science*, Györ, Hungary, 1983.

[56] Wos, L., Automated reasoning: 33 basic research problems. Prentice Hall, Englewood Cliffs, N.J., 1988.

[57] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., Automated reasoning: Introduction and applications. 2nd ed. McGraw-Hill, New York, 1992.

[58] Zhang, H., Reduction, Superposition and Induction: Automated Reasoning in an Equational Logic. Ph.D. thesis, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, N.Y., 1988.

[59] Zhang, H., Automated proof of ring commutativity problems by algebraic methods, *J. Symbolic Computation* 9 (1990) 423–427.

[60] Zhang, H., Criteria of critical pair criteria: A practical approach and a comparative study, *J. Automated Reasoning*, to appear.

[61] Zhang, H., Herky: High-performance rewriting techniques in RRL. In: Proc. *International Conference of Automated Deduction*, D. Kapur (ed.), Lecture Notes in Artificial Intelligence, Vol. 607, Springer-Verlag, Belrin, 1992, pp. 696–700.

[62] Zhang, H., Implementing contextual rewriting. In: Proc. *Third International Workshop on Conditional Term Rewriting Systems,* J. L. Remy and M. Rusinowitch (eds.), Lecture Notes in Computer Science, Vol. 656, Springer-Verlag, Berlin, 1992, pp. 363–377.

[63] Zhang, H., Proving group isomorphism theorems: A case study of conditional completion. In: Proc. *Third International Workshop on Conditional Term Rewriting Systems*, J. L. Remy and M. Rusinowitch (eds.), Lecture Notes in Computer Science, Vol. 656, Springer-Verlag, Berlin, 1992, pp. 302–306.

[64] Zhang, H., A new strategy for the Boolean ring based approach to first order theorem proving, *J. Symbolic Computation*, to appear.

[65] Zhang H., Automated Proofs of Equality Problems in Overbeek's Competition. Report 92-06, Department of Computer Science, The University of Iowa, 1993.

[66] Zhang, H., A case study of completion modulo distributivity and Abelian groups. In: Proc. *International Conference on Rewrite Techniques and Applications*, C. Kirchner (ed.), Lecture Notes in Computer Science, Vol. 690, Springer-Verlag, 1993, pp. 32–46.

[67] Zhang, H., Guha, A., and Hua, G. X., Using algebraic specifications in Floyd-Hoare assertions. In: Proc. *Second International Conference on Algebraic Methodology and Software Technology,* T. Rus (ed.), Iowa City, Iowa, 1991.

[68] Zhang, H., and Hua, G. X., Proving Ramsey theorem by the cover set induction: A case and comparison study. In: *The Annals of Mathematics and Artificial Intelligence*, Vol. 8, No. 3–4, 1992.

[69] Zhang, H., and Hua, G. X., Proving the Chinese remainder theorem by the cover ste induction. In: Proc. *International Conference of Automated Deduction*, D. Kapur (ed.) Lecture Notes in Artificial Intelligence, Vol. 607, Springer-Verlag, Berlin, 1992, pp. 431–445.

[70] Zhang, H., and Kapur, D., First-order logic theorem proving using conditional rewrite rules. In: E. Lusk and R. Overbeek (eds.), Proc. *Ninth International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 310, Springer-Verlag, Berlin, 1988, pp. 1–20.

[71] Zhang, H., and Kapur, D., Unnecessary inferences in associative-commutative completion procedures, *J. Mathematical System Theory* 23 (1990) 175–206.

[72] Zhang, H., Kapur, D., and Krishnamoorthy, M. S., A mechanizable induction principle for equational specifications. In: Proc. *Ninth International Conference on Automated Deduction*, E. Lusk and R. Overbeek (eds.), Lecture Notes in Computer Science, Vol. 310, Springer-Verlag, Berlin, 1988, pp. 250–265.