Fundamenta Informaticae 28 (1996) 129–140
IOS Press

Automating Algebraic Proofs in Algebraic Logic

Jieh Hsiang*
Department of Computer Science,
National Taiwan University,
Taipei, Taiwan
hsiang@csie.ntu.edu.tw

Anita Wasilewska†
Department of Computer Science,
State University of New York at Stony Brook,
Stony Brook, NY, USA
anita@sbcs.sunysb.edu

Abstract. We present here an effective proof theory that allows one to reason within algebras of algebraic logic in a purely syntactic, algebraic fashion. We demonstrate the effectiveness of the method by discussing our automated proofs of problems and theorems taken from Professor Helena Rasiowa's book *An Algebraic Approach to Non-Classical Logics*, Studies in Logic and Fundation of Mathematics, Volume 78, North Holland Publishing Company, Amsterdam, London - PWN, Warsaw (1974). We include a detailed proof of a problem presented to us by Professor Helena Rasiowa in June 1993. Most of these proofs are the first *direct* proofs ever discovered, and they are produced by the computer without human assistance.

1. Introduction

The field of algebraic logic studies the relationship between logic and algebra. It starts from purely logical considerations, abstracts from them, places them into a general algebraic context, and makes use of other branches of mathematics such as topology and set theory. It provides a way to investigate symbolic logic from a different perspective, and serves to clarify problems in logic by establishing connections with ordinary mathematics. Indeed, one of the goals of algebraic logic is to treat logic algebraically so that the rich variety of techniques developed in algebra can be put to good use in logic. It allows to study a given logic by investigating its corresponding algebra. The algebra corresponding to classical logic is a Boolean algebra. In 1963 Rasiowa and Sikorski provided, in their classic book *The Mathematics of Metamathematic* ([13]), an uniform, systematic presentaion of the algebraic approach to classical, intuitionistic, modal, and positive logic. In 1974 Rasiowa formulated, in her book *An Algebraic Approach to Non-Classical Logics* ([14]), an algebraic approach to a carefully selected, widest possible class of logics which can be approached from algebraic

*Supported in part by grant NSC 95-2221-E-002-009 of the National Science Council of the Republic of China.

†Supported by Fulbright grant USIA 93-68818 (1993-1994).

point of view. The algebras discussed in the book were the following: positive implication algebras, implication algebras, lattices and distributive lattices, quasi-Boolean algebras, relatively pseudo complemented, contrapositionally complemented and semi-complemented lattices, pseudo-Boolean algebras, quasi-pseudo-Boolean algebras, Boolean and topological Boolean algebras and finally, Post algebras.

It is therefore interesting to note that, despite tremendous progress in algebraic logic over the past half of century, there has not been any effective proof theory that allows one to reason within algebras of algebraic logic in a purely syntactic fashion.

In this paper we address this issue by describing a simple equational proof method called *unfailing Knuth-Bendix completion.* This method was developed by the term rewriting community and is a semi-decision procedure for proving equational theorems of an equational theory. In addition to being one of the most powerful techniques in automated deduction, unfailing completion yields direct proofs using equational replacements which are easy to understand and check by human. We demonstrate the effectiveness of the method by presenting proofs of problems and theorems taken from the book *An Algebraic Approach to Non-Classical Logics* ([14]). We remark that most of these proofs are the first *direct*, syntactic proofs ever discovered, and that they are produced by the computer without human assistance. Our experiments suggest that automated proof procedures such as unfailing completion may be viable candidates for effective algebraic proof methods of algebraic logic.

2. Overview Unfailing Completion

An important goal of automated deduction is to automate proofs in mathematics. Since generality is an important concern, automated deduction methods are usually syntactic in nature. The emergence of resolution and its refinements in the sixties pushed the endeavour of automatically proving theorems in first order predicate to a new horizon. At about the same time, a method called *Knuth-Bendix completion procedure* [10] was developed for deduction in equational logic. The most distinctive characteristic of Knuth-Bendix completion (and other proof systems extended from it) is the incorporation of the concept of well-founded ordering into the inference mechanism. Given a well-founded ordering[1] $>$ on the term algebra, an equation $l = r$ is treated as a rewrite rule $l \rightarrow r$ if $l > r$. Since $>$ is well-founded, $\rightarrow$ is an irreflexive, antisymmetric, transitive congruent binary relation on the term algebra. Furthermore, for every substitution σ and terms s and t, if $s \rightarrow t$, then $s\sigma \rightarrow t\sigma$. Such a relation $\rightarrow$ is called a *reduction relation.* Operationally, let s be a term with a subterm u at position p (which we denote as $s[u]_p$), and $l \rightarrow r$ be a rewrite rule. If there is a subsitution σ such that $u = l\sigma$, then $s[u]_p$ can be *reduced* to $s[r\sigma]_p$. If R is a set of rewrite rules and s cannot be reduced using any rules in R, then s is *R-irreducible.* An irreducible term is also said to be in *normal form.*

We remark that, since $>$ is well-founded, the reduction relation defined using $>$ is also well-founded. In other words, a term cannot be reduced indefinitely.

The basic difference between applying a rule to a term and applying an equation is that in reduction, a term is *replaced* by the term that it reduces to. Although logically they may look the same, this difference is very important for automated deduction since, by keeping only one term of a class of equivalent terms, the search space can be reduced significantly.

Knuth-Bendix completion was originally designed for transforming a set of equations E into a canonical set R of rewrite rules, under which the reduction relation yields a unique normal form for every term. Thus, proving the equivalence of two terms s and t under the

[1]This ordering is usually a simplification ordering [4]. A *simplification ordering* on a set of terms is a binary irreflexive partial order with the following properties: Let s and t be terms and σ be a substitution, if $s > t$ then $s\sigma > t\sigma$ and $u[s]_p > u[t]_p$ for every term u. Furthermore, $u[s] > s$ if s is not a trivial subterm of u. By $u[s]_p$ we mean a term u, whose subterm at position p is s. The term $u[t]_p$ is the same term u except that the subterm at position p is t.

theory E becomes simply reducing s and t to their respective R-normal forms and check if they are identical.

The first critical barrier that prohibited extensive use of Knuth-Bendix completion was that there are important equations which cannot be oriented under any well-founded ordering. One typical example is the axiom of commutativity, $x * y = y * x$, which is commonplace in algebraic theories. This difficulty was partially overcome with the utilization of E-unification. Informally speaking, E-unification incorporates some of the axioms (especially those un-orientable ones) into the unification process. The problem with commutivity was solved in [12] by building both commutivity and associativity (AC) into the unification process. AC-Knuth-Bendix completion was extended considerably to allow building any axiom set into the unification process as long as the axioms yield a finite and complete unification algorithm [9].

A complete solution to this problem was presented in [7]. We called this method the *unfailing Knuth-Bendix completion procedure*, or *unfailing completion* for short. It forms the basis of the proof theory used in deriving the proofs in this paper. In the following we describe the method in detail.

A simplification ordering is *complete* if it is a total order on ground terms. The unfailing completion method assumes that there is a pre-defined complete simplification ordering $>$ on the set of terms. Given an equation $s = t$, an *orientable instance* of $s = t$ is an instance $s' = t'$ such that $s' > t'$. If $s = t$ itself is orientable, that is, $s > t$, then we may emphathize this fact by representing $s = t$ as a rewrite rule $s \rightarrow t$. Since rules are just orientable equations, in the rest of the paper we do not distinguish between equations and rules.

The first inference rule is the inference rule of *simplification*. Let $u[w] = v$ and $s = t$ be two equations. If there is a substitution σ such that $w = s\sigma$ and $s\sigma > t\sigma$, then $s[w] = v$ can be *simplified* into $s[t\sigma] = v$. Note that even if an equation cannot be oriented, its orientable instances can still be used to perform simplifications. A slightly more general, but more complicated, simplification rule can be found in [2].

As an example, let $(a * b) * c$ be a term and $x * y = y * x$ be an (un-orientable) equation. Assume that $* > a > b > c$ and the left-to-right lexicographic ordering is used, then in our ordering $a * b > b * a$ and $(b * a) * c > c * (b * a)$. Thus, the term $(a * b) * c$ can be reduced first to $(b * a) * c$, then to $c * (b * a)$. We remark, once again, that this reduction relation using orientable instances of $x * y = y * x$ is well-founded.

The second inference rule is the rule of *superposition*. Let $u[w] = v$ and $s = t$ be two equations in which w is not a single variable. If there is a most general unifier σ between w and s such that $u\sigma \not\leq v\sigma$ and $s\sigma \not\leq t\sigma$, then a *critical pair* $u[t]\sigma = v\sigma$, in which both terms are equivalent to the superposition $u[w]\sigma$, is produced.

We present unfailing completion as a refutational proof method. It works as follows: Given a set of equations E and a theorem $\forall x(s = t)$ to be proved. We first negate and skolemize the equation into $\hat{s} \neq \hat{t}$ where $\hat{s}$ and $\hat{t}$ are s and t with variables replaced by skolem constants. We then repeatedly apply simplification and superposition inference rules to $E \cup \{\hat{s} \neq \hat{t}\}$. (Note that the inequality does not play any role in the superposition process.) The process terminates if either no more inference rule can be applied, or if the inequality is simplified to one of the form $\hat{r} \neq \hat{r}$ for some r. In the latter case, we have established that $s = t$ is a theorem of E. It has been shown that unfailing completion is a semi-decision procedure for proving universally quantified equations of any equational theory [7].

Several efficiency measures are often employed when performing unfailing completion. Simplification is usually applied as much as possible before any superposition step is performed. By the same token, once a critical pair is generated, it is also simplified until both terms are irreducible with respect to the current set of equations. This way the data base can be kept as compact as possible. Another common optimization involves special-purpose unification. If an operator is both associative and commutative (AC), one may incorporate both axioms into the unification process by declaring the operator to be AC. Then

AC-unification instead of regular unification is used. Although AC-unification, as a single inference step, may be much slower than regular unification, it often provides great savings as far as search space is concerned. When an operator f has the cancellation axiom, that is, $y = z$ if $f(x, y) = f(x, z)$, then this axiom can also be incorporated into part of the inference mechanism to produce smaller critical pairs. For complete sets of inference rules for the cancellation axioms, see [8].

2.1. Automated Deduction using Unfailing Completion

In addition to being highly effective in producing proofs, there are several other advantages with choosing unfailing completion and its extensions as the automated proof theory for algebraic logics.

First of all, unfailing completion is easy to implement. Indeed, there are many implementations of unfailing completion existing in languages ranging from Prolog to C++.

Second, unfailing completion is easy to use, since the inference system is fully automated and does not need human intervention when being executed. Take *Reveal* [1] as an example. To prove an equational theorem in *Reveal*, one only needs to input the equational theory (as a set of equations), the equations to be proved, and an ordering on the operators. The ordering will be extended automatically to a term ordering (ordering on the set of terms) that the user chooses. *Reveal* incorporates two most commonly used orderings on terms, the *recursive path ordering* and the *lexicographic path ordering* [5].

We remark that the choice of an appropriate ordering is not as difficult as it may sound at first glance. The user usually wants some equations to be oriented in a specific way. For instance, the distributivity law $x*(y+z) = x*y+x*z$ is usually ordered from left to right. Therefore one may declare that $* > +$, which will orient the equation accordingly in both path orderings. Unlike the original Knuth-Bendix completion, which may fail if an equation cannot be oriented, unfailing completion proceeds regardless of how many equations cannot be oriented. This flexibility relieves the user from the need to have a deep knowledge of the inner working of the orderings.

Since *Reveal* implements C-unification, AC-unification, and the cancellation inference rules, the user may also wish to declare some operators to be commutative, commutative and associative, or cancellable. The following is a typical input to *Reveal*.

```
term

i(i(x,x),y)=y ;
i(x,i(y,z))=i(i(x,y),i(x,z)) ;
i(i(x,y),i(i(y,x),y))=i(i(y,x),i(i(x,y),x)) ;
i(i(x,y),x)=x;
o(x,y)=i(i(x,y),y);
i(x,x)=d;

i(a,c)=d;
i(b,c)=d;
i(o(a,b),c)!=d;
;

prec d a b c i o ;
sta i l ;
sta o l ;

ukb
```

The first input `term` indicates that *Reveal* is taking input from the terminal. The input equations end with a single ";". The line "`prec d a b c i o ;`" gives the ordering on the operators, in increasing order, and the line "`sta i 1 ;`" indicates that the lexicographic ordering from left to right is used when two terms with the same outermost operator `i` are compared.

The meaning of the above problem is that in an implication algebra $(A, \mathtt{d}, \Rightarrow)$ (as defined by the first group of equations, in which `i` stands for $\Rightarrow$ and `o` stands for $\cup$), for every s, y and z, if $x \Rightarrow z = \mathtt{d}$ and $y \Rightarrow z = \mathtt{d}$, then $(x \cup y) \Rightarrow z = \mathtt{d}$.

The theorem to be proved, after being negated and skolemized, becomes the second group of the two equations and one inequality. The variables in the theorem, after skolemization, becomes constants `a`,`b`, and `c`. Incidentally, it took *Reveal* less tham 1.5 seconds to prove this problem on a Sparcstation 10/51 with 64MB of memory.

The third advantage we mention here is that the proofs produced by a prover based on unfailing completion are usually easy for human to read and to check. The first reason is because unfailing completion works directly on the set of equations without changing them into a different form[2]. Secondly, the inference rules of unfailing completion are essentially highly restricted versions of equational replacement. The superposition inference rule corresponds to finding an equational lemma from the current set of equations, and simplification is plain equational replacement using existing equations. These inference rules coincide naturally with human reasoning. It is fairly easy to reconstruct a proof produced by the prover into a typical algebraic proof that one sees in a logic or algebra textbook. Indeed, it is even fairly easy to write a program which does this transformation automatically [6]. In Section 3.1., we will show such a proof in detail.

3. Proofs in Peirce Algebras

To demonstrate that unfailing completion is indeed a feasible proof theory for algebraic logic, we conducted a number of experiments using two of the most popular implementations, *Reveal* [1] and *Otter* [11]. Both provers are written in C. While both of them are very easy to use, *Reveal* is implemented for Sparcstations and has the advantage of featuring AC-unfailing completion as well as inference rules for cancellation. On the other hand, *Otter* runs on PC and Sparcstations, and is slightly faster than *Reveal*. *Otter* has the additional advantage of being a first-order theorem prover as well, although its first-order part cannot yet utilize the concept of ordering to its full benefit.

The problems we tested are all taken from the book "*An Algebraic Approach to Non-Classical Logics*" [14]. They include various properties of implication algebras, positive implication algebras and quasi-Boolean algebras. There has not been any known direct syntactic proof for most of the problems. In the following we give a proof of a problem in Peirce algebras in full detail. This proof is a typical example of what unfailing completion is capable of doing.

In order to fully appreciate the proof, we provide, below, some information concerning Peirce algebras. Peirce algebras[3] were first introduced in [14] in the exercise section of page 48. We should remark that many of the exercises in the book are either published results or open problems, despite their unassuming position in the book. There were four problems asked about Peirce algebras. They are:

Problem 1 Prove that every Peirce algebra is a distributive lattice with the unit element V.

[2]Such a pre-processing transformation is quite common in automated deduction. For example, resolution-based methods typically require the input to be transformed into clausal form.

[3]Another class of algebras bearing the same name appeared in [3]. But these two algebras do not seem to relate to each other despite their identical name.

Problem 2 Prove that the notion of an implicative filter Δ in Peirce algebra algebra coincides with the notion of a filter.

Problem 3 Let $\mathcal{A}$ be a Peirce algebra, prove that the quotient algebra $\mathcal{A}/\nabla$ is a Peirce algebra.

Problem 4 Prove the Representation Theorem for Peirce algebras.

Since the problems were given without any references, we contacted Professor Rasiowa in order to obtain more detail. She gave us, in June 1993, the following information:

1. Peirce Algebras were defined to provide an algebraic model for a negation-free fragment of classical logic. The name was derived from the *Peirce Law* $(((a \Rightarrow b) \Rightarrow a) \Rightarrow a)$ which holds in the corresponding logic. Let us denote this logic as PL. One can then prove (using **Problem 3** as the essential step) the following Separation Theorem for Peirce logic.

   Let A be a negation free propositional formula, then A is provable in classical logic if and only if A is provable in Peirce logic.

2. One can then use the implicative filter of **Problem 2** to prove the Representation Theorem mentioned in **Problem 4**.
3. The solution to the **Problem 1** becomes a corollary to the Representation Theorem.
4. A direct proof, not via the Representation Theorem, of the **Problem 1** is not known.

Since a conventional solution for **Problem 1** depends on successfully solving the other three problems, it is curious why Rasiowa decided to give it as the first problem as opposed to be the last one. Unfortunately, we did not obtain an answer for this question.

The approach to **Problem 1** suggested by Rasiowa reflects more or less how problems concerning the expressive power of a logic are solved in algebraic logic; one essentially needs to prove a representation theorem for each of the algebras in question.

Definition 3.1. A *Peirce algebra* is an algebra $(A, \Rightarrow, \cup, \cap)$ satisfying

I1 $(a \Rightarrow a) \Rightarrow b = b$,
I2 $a \Rightarrow (b \Rightarrow c) = (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$,
I3 $(a \Rightarrow b) \Rightarrow ((b \Rightarrow a) \Rightarrow b) = (b \Rightarrow a) \Rightarrow ((a \Rightarrow b) \Rightarrow a)$,
P $(a \Rightarrow b) \Rightarrow a = a$,
P1 $a \cup b = (a \Rightarrow b) \Rightarrow b$,
P2 $(a \cap b) \Rightarrow a = V$,
P3 $(a \cap b) \Rightarrow b = V$,
P4 $(a \Rightarrow b) \Rightarrow ((a \Rightarrow c) \Rightarrow (a \Rightarrow (b \cap c))) = V$.

We remark that any algebra that satisfies axioms *I1* through *I3* is called a *positive implication algebra*, and that any positive implication algebra with the addition axiom *P* is called an *implication algebra.*

Definition 3.2. An algebra $(A, \cup, \cap)$ is a *distributed lattice* if it satisfies the following axioms:

1. $a \cup b = b \cup a$,
2. $a \cap b = b \cap a$,
3. $a \cup (b \cup c) = (a \cup b) \cup c$,
4. $a \cap (b \cap c) = (a \cap b) \cap c$,
5. $(a \cap b) \cup b = b$,
6. $a \cap (a \cup b) = a$,

7. $a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$,
8. $a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$.

Both provers *Reveal* and *Otter* proved, automatically, the eight properties of distributed lattices from the axioms of Peirce algebra. Otter is the faster one, and took 591.57 cpu seconds on a Sparcstation 10/51. The entire proof generated 160 useful lemmas, and the most difficult among the proofs is the property is the associativity of $\cup$. The number of lemmas necessary to prove the theorem indicates that it is highly unlikely that such a direct, algebraic proof can be easily performed by human.

3.1. Commutativity of $\cap$ in Peirce Algebra – An Example

In the following, we present the proof generated by Otter of

Theorem 3.1. *In Peirce algebra, $x \cap y = y \cap x$ for every x and y.*

We choose this theorem as an example because the symbol $\cap$ is not explicitly defined in Peirce algebra. It is therefore interesting to see how unfailing completion "pulls out" the symbol via superposition and simplification.

The following is an output script produced by *Otter*. The proof took 7.63 cpu seconds on a Sparcstation 10/51 with 64MB RAM.

```
----> UNIT CONFLICT at   7.63 sec ---->

Level of proof is 14, length is 22.

---------------- PROOF ----------------

2 [] (i(i(x,x),y) = y).
5,4 [] (i(i(x,y),i(x,z)) = i(x,i(y,z))).
6 [] (i(i(x,y),i(i(y,x),y)) = i(i(y,x),i(i(x,y),x))).
9,8 [] (i(i(x,y),x) = x).
12 [] (i(and(x,y),x) = d).
15,14 [] (i(and(x,y),y) = d).
16 [demod,5,5] (i(x,i(y,i(z,and(y,z)))) = d).
19,18 [] (i(x,x) = d).
20 [] (and(b,a) != and(a,b)).
21 [back_demod,6,demod,9,9] (i(i(x,y),y) = i(i(y,x),x)).
24,23 [back_demod,2,demod,19] (i(d,y) = y).
26,25 [para_into,8,23] (i(x,d) = d).
27 [para_into,8,8] (i(x,i(x,y)) = i(x,y)).
35 [para_into,16,27] (i(x,i(y,and(x,y))) = d).
55 [para_into,21,14,demod,24] (i(i(x,and(y,x)),and(y,x)) = x).
57 [para_into,21,12,demod,24] (i(i(x,and(x,y)),and(x,y)) = x).
71 [para_into,4,35,demod,24] (i(x,i(i(z,and(x,z)),y)) = i(x,y)).
77 [para_into,4,14,demod,24] (i(and(x,y),i(y,z)) = i(and(x,y),z)).
93 [para_into,4,18,demod,26] (i(x,i(y,x)) = d).
97 [para_into,4,12,demod,26] (i(and(x,y),i(z,x)) = d).
120,119 [para_from,93,4,demod,24] (i(x,i(i(z,x),y)) = i(x,y)).
139 [para_into,97,21] (i(and(x,y),i(i(x,z),z)) = d).
1838,1837 [para_into,71,55] (i(x,and(x,y)) = i(x,y)).
1857 [back_demod,57,demod,1838] (i(i(x,y),and(x,y)) = x).
2017 [para_from,1857,119] (i(x,and(y,x)) = i(x,y)).
```

```
2032,2031 [para_from,2017,119,demod,120] (i(x,and(z,i(y,x))) = i(x,z)).
2483 [para_into,77,139] (i(and(x,i(x,y)),y) = d).
2603 [para_from,2483,21,demod,24,2032] (i(i(x,y),and(y,i(y,x))) = x).
2908,2907 [para_into,2603,12,demod,1838,24] (and(x,i(x,y)) = and(x,y)).
2913 [para_into,2603,2017,demod,15,2908,24] (and(y,x) = and(x,y)).
2915 [binary,2913,20] .

------------ end of proof -------------
```

The proof can be converted easily into a series of lemmas which are easy to read and check.
Given

1. $(x \Rightarrow x) \Rightarrow y = y$
2. $(x \Rightarrow y) \Rightarrow (x \Rightarrow z) = x \Rightarrow (y \Rightarrow z)$
3. $(x \Rightarrow y) \Rightarrow ((y \Rightarrow x) \Rightarrow y) = (y \Rightarrow x) \Rightarrow ((x \Rightarrow y) \Rightarrow x)$
4. $(x \Rightarrow y) \Rightarrow x = x$
5. $(x \cap y) \Rightarrow x = d$
6. $(x \cap y) \Rightarrow y = d$
7. $(x \Rightarrow y) \Rightarrow ((x \Rightarrow z) \Rightarrow (x \Rightarrow (y \cap z))) = d$
8. $x \Rightarrow x = d$
9. $b \cap a \neq a \cap b$

Lemma 3.1. $x \Rightarrow (y \Rightarrow (z \Rightarrow (y \cap z))) = d$

Proof:
This equation is obtained by simplifying the left hand side (LHS) of equation (7) by equation (2) twice. □

We remark that, since Lemma 3.1. is obtained directly from equation (7) through simplification by other axioms, it is probably better to use the equation of Lemma 3.1. as one of the axioms for Peirce algebra instead of the original one – equation (7) – given by Rasiowa.

Lemma 3.2. $(x \Rightarrow y) \Rightarrow y = (y \Rightarrow x) \Rightarrow x$

Proof:
Obtained by simplifying both sides of equation (3) by equation (4) □

Lemma 3.3. $d \Rightarrow y = y$

Proof:
This lemma is the result of simplifying equation (1) using equation (8) □

Lemma 3.4. $x \Rightarrow d = d$

Proof:
LHS: $(d \Rightarrow y) \Rightarrow d = y \Rightarrow d$, by equation (4)
RHS: $(d \Rightarrow y) \Rightarrow d = d$, by Lemma 3.3. □

Lemma 3.5. $x \Rightarrow (x \Rightarrow y) = x \Rightarrow y$

Proof:
LHS: $((y \Rightarrow w) \Rightarrow y) \Rightarrow (y \Rightarrow w) = y \Rightarrow (y \Rightarrow w)$ by equation (4)
RHS: $((y \Rightarrow w) \Rightarrow y) \Rightarrow (y \Rightarrow w) = y \Rightarrow w$, also by equation (4) □

Lemma 3.6. $x \Rightarrow (y \Rightarrow (x \cap y)) = d$

Proof:
LHS: $u \Rightarrow (y \Rightarrow (z \Rightarrow (y \cap z))) = u \Rightarrow (z \Rightarrow (u \cap z))$ by Lemma 3.5.
RHS: $u \Rightarrow (y \Rightarrow (z \Rightarrow (y \cap z))) = d$ by Lemma 3.1. □

Lemma 3.7. $(x \Rightarrow (y \cap x)) \Rightarrow (y \cap x) = x$

Proof:
LHS: $((u \cap v) \Rightarrow v) \Rightarrow v = (v \Rightarrow (u \cap v)) \Rightarrow (u \cap v)$ by Lemma (3.2.)
RHS: $((u \cap v) \Rightarrow v) \Rightarrow v = d \Rightarrow v$ by equation (6)
$= v$ by Lemma 3.3. □

Lemma 3.8. $(x \Rightarrow (x \cap y)) \Rightarrow (x \cap y) = x$

Proof:
LHS: $((u \cap v) \Rightarrow u) \Rightarrow u = (u \Rightarrow (u \cap v)) \Rightarrow (u \cap v)$ by Lemma (3.2.)
RHS: $((u \cap v) \Rightarrow u) \Rightarrow u = d \Rightarrow u$ by equation (5)
$= u$ by Lemma 3.3. □

Lemma 3.9. $x \Rightarrow ((z \Rightarrow (x \cap z)) \Rightarrow y) = x \Rightarrow y$

Proof:
LHS: $(x \Rightarrow (v \Rightarrow (x \cap v))) \Rightarrow (x \Rightarrow z) = x \Rightarrow ((v \Rightarrow (x \cap v)) \Rightarrow z)$ by equation (2)
RHS: $(x \Rightarrow (v \Rightarrow (x \cap v))) \Rightarrow (x \Rightarrow z) = d \Rightarrow (x \Rightarrow z)$ by Lemma 3.6.
$= x \Rightarrow z$ by Lemma 3.3. □

Lemma 3.10. $(x \cap y) \Rightarrow (y \Rightarrow z) = (x \cap y) \Rightarrow z)$

Proof:
LHS: $((u \cap v) \Rightarrow v) \Rightarrow ((u \cap v) \Rightarrow z) = (u \cap v) \Rightarrow (v \Rightarrow z)$ by equation (2)
RHS: $((u \cap v) \Rightarrow v) \Rightarrow ((u \cap v) \Rightarrow z) = d \Rightarrow ((u \cap v) \Rightarrow z)$ by equation (6)
$= (u \cap v) \Rightarrow z$ by Lemma 3.3. □

Lemma 3.11. $x \Rightarrow (y \Rightarrow x) = d$

Proof:
LHS: $(x \Rightarrow y) \Rightarrow (x \Rightarrow x) = x \Rightarrow (y \Rightarrow x)$ by equation (2)
RHS: $(x \Rightarrow y) \Rightarrow (x \Rightarrow x) = (x \Rightarrow y) \Rightarrow d$ by equation (8)
$= d$ by Lemma 3.4. □

Lemma 3.12. $(x \cap y) \Rightarrow (z \Rightarrow x) = d$

Proof:
LHS: $((u \cap v) \Rightarrow y) \Rightarrow (u \cap v) \Rightarrow u) = (u \cap v) \Rightarrow (y \Rightarrow u)$ by equation (2)
RHS: $((u \cap v) \Rightarrow y) \Rightarrow (u \cap v) \Rightarrow u) = ((u \cap v) \Rightarrow y) \Rightarrow d$ by equation (5)
$= d$ by Lemma 3.4. □

Lemma 3.13. $x \Rightarrow (z \Rightarrow x) \Rightarrow y) = x \Rightarrow y$

Proof:
LHS: $(u \Rightarrow (v \Rightarrow u)) \Rightarrow (u \Rightarrow z) = u \Rightarrow ((v \Rightarrow u) \Rightarrow z)$ by equation (2)
RHS: $(u \Rightarrow (v \Rightarrow u)) \Rightarrow (u \Rightarrow z) = d \Rightarrow (u \Rightarrow z)$ by Lemma 3.11.
$= u \Rightarrow z$ by Lemma 3.3. □

Lemma 3.14. $(x \cap y) \Rightarrow ((x \Rightarrow z) \Rightarrow z) = d$

Proof:
LHS: $(x \cap y) \Rightarrow ((z \Rightarrow x) \Rightarrow x) = (x \cap y) \Rightarrow ((x \Rightarrow z) \Rightarrow z)$ by Lemma 3.2.
RHS: $(x \cap y) \Rightarrow ((z \Rightarrow x) \Rightarrow x) = d$ by Lemma 3.12. □

Lemma 3.15. $x \Rightarrow (x \cap y) = x \Rightarrow y$

Proof:
LHS: $x \Rightarrow ((z \Rightarrow (x \cap z)) \Rightarrow (x \cap z)) = x \Rightarrow (x \cap z)$ by Lemma 3.9.
RHS: $x \Rightarrow ((z \Rightarrow (x \cap z)) \Rightarrow (x \cap z)) = dx \Rightarrow z$ by Lemma 3.7. □

Lemma 3.16. $(x \Rightarrow y) \Rightarrow (x \cap y) = x$

Proof:
The lemma is obtained from simplifying the LHS of Lemma 3.8. using Lemma 3.15. □

Lemma 3.17. $x \Rightarrow (y \cap x) = x \Rightarrow y$

Proof:
LHS: $x \Rightarrow (z \Rightarrow x) \Rightarrow (z \cap x) = x \Rightarrow (z \cap x)$ by Lemma 3.13.
RHS: $x \Rightarrow (z \Rightarrow x) \Rightarrow (z \cap x) = x \Rightarrow z$ by Lemma 3.16. □

Lemma 3.18. $x \Rightarrow (z \cap (y \Rightarrow x)) = x \Rightarrow z$

Proof:
LHS: $u \Rightarrow ((w \Rightarrow u) \Rightarrow (y \cap (w \Rightarrow u))) = u \Rightarrow (y \cap (w \Rightarrow u))$ by Lemma 3.13.
RHS: $u \Rightarrow ((w \Rightarrow u) \Rightarrow (y \cap (w \Rightarrow u))) = u \Rightarrow ((w \Rightarrow u) \Rightarrow y)$ by Lemma 3.17.
$= u \Rightarrow y$ by Lemma 3.13. □

Lemma 3.19. $(x \cap (x \Rightarrow y)) \Rightarrow y = d$

Proof:
LHS: $(x \cap (x \Rightarrow z)) \Rightarrow ((x \Rightarrow z) \Rightarrow z) = (x \cap (x \Rightarrow z)) \Rightarrow z)$ by Lemma 3.10.
RHS: $(x \cap (x \Rightarrow z)) \Rightarrow ((x \Rightarrow z) \Rightarrow z) = d$ by Lemma 3.14. □

Lemma 3.20. $(x \Rightarrow y) \Rightarrow (y \cap (y \Rightarrow x)) = x$

Proof:
LHS: $((x \cap (x \Rightarrow y)) \Rightarrow y) \Rightarrow y = (y \Rightarrow (x \cap (x \Rightarrow y)) \Rightarrow (x \cap (x \Rightarrow y))$ by Lemma 3.2.
$= (y \Rightarrow x) \Rightarrow (x \cap (x \Rightarrow y))$ by Lemma 3.18.
RHS: $((x \cap (x \Rightarrow y)) \Rightarrow y) \Rightarrow y = d \Rightarrow y$ by Lemma 3.19.
$= y$ by Lemma 3.3. □

Lemma 3.21. $x \cap (x \Rightarrow y) = x \cap y$

Proof:
LHS: $((u \cap v) \Rightarrow u) \Rightarrow (u \cap (u \Rightarrow (u \cap v))) = d \Rightarrow (u \cap (u \Rightarrow (u \cap v)))$ by equation (5)
$= d \Rightarrow (u \cap (u \Rightarrow v))$ by Lemma 3.15.
$= u \cap (u \Rightarrow v)$ by Lemma 3.3.
RHS: $((u \cap v) \Rightarrow u) \Rightarrow (u \cap (u \Rightarrow (u \cap v)))\, u \cap v$ by Lemma 3.20. □

Lemma 3.22. $x \cap y = y \cap x$

Proof:
LHS: $((w \cap y) \Rightarrow y) \Rightarrow (y \cap (y \Rightarrow (w \cap y))) = w \cap y$ by Lemma 3.20.
RHS: $((w \cap y) \Rightarrow y) \Rightarrow (y \cap (y \Rightarrow (w \cap y))) = ((w \cap y) \Rightarrow y) \Rightarrow (y \cap (y \Rightarrow w))$ by Lemma 3.17.
$= d \Rightarrow (y \cap (y \Rightarrow w))$ by equation (6)
$= d \Rightarrow (y \cap w)$ by Lemma 3.21.
$= y \cap w$ by Lemma 3.3. □

The last lemma implies Theorem 3.1.

4. Discussion

Proofs in algebraic logic usually involve the discovery, and proof, of a Representation Theorem for a given algebra. Proving the Representation Theorem and associated lemmas involves the use of mathematical techniques which are not purely algebraic in nature.

In this paper we presented a proof method which, we argue, can be used to automate proofs in algebraic logic. The method, called *unfailing completion*, is essentially a highly effective refinement of Birkhoff's inference rules for equational logic. It presumes a well-founded ordering on the set of terms, which forms the foundation of all inferences. Through the rule of *superposition*, it produces new equations (lemmas). Through the rule of *simplification*, it keeps the terms (and thus the search space) minimal.

In addition to being effective, unfailing completion is easy to implement, its implementations are easy to use, and the proofs that it produces can be easily reformulated into lemmas which are, in turn, easy to read and understand.

We have used several different impelementations of unfailing completion to proof-check a number of theorens and exercises in the book *An Algebraic Approach to Non-Classical Logics* of Rasiowa. They include many problems concerning implication algebra, positive implication algebra, quasi-Boolean algebra, and Peirce algebra. For most of these problems there had not been any *direct* algebraic proofs. Since these proofs are often quite long, in this paper we only included one detailed proof in Peirce algebra. Our experiments show that unfailing completion is indeed a viable (automated) proof theory for algebraic logic.

References

[1] Anantharaman, S., Hsiang, J., and Mzali, J., SbReve2: A term rewriting laboratory with (AC)-unfailing completion, in *3rd RTA*, Springer-Verlag Lecture Notes in Computer Science, Vol 355, pp 533–537, 1989.

[2] Bachmair, L., Dershowitz, N., and Plaisted, D., Completion without failure. In *Resolution of equations in algebraic structures 2*, Ait-Kaci and Nivat, eds., Academic Press, pp 1–30, 19899

[3] Brink, C., On the application of relations, *South African Journal of Philosophy*, 7(2):105–112, 1988.

[4] Dershowitz, N., Orderings for term-rewriting systems, *Theoretical Computer Science* vol 17, pp279–301, 1982.

[5] Dershowitz, N., Termination of rewriting, *Journal of Symbolic Computation*, Vol 3, pp 69–116, 1987.

[6] Hsiang, J., On the reconstruction of equational proofs, in preparation.

[7] Hsiang, J. and Rusinowitch, M., On word problems in equational theories, in Th.Ottman (ed.), *Proceedings of the Fourteenth International Conference on Automata, Languages and Programming*, Karlsruhe, West Germany, July 1987, Springer Verlag, Lecture Notes in Computer Science 267, 54–71, 1987.

[8] Hsiang, J.,Rusinowitch, M. and Sakai, K., Complete set of inference rules for the cancellation laws, IJCAI 87, Milan, Italy, 1987.

[9] Jouannaud, J.-P. and Kirchner, H., Completion of a set of rules modulo a set of equations, *SIAM J. of Computing*, 15(4), pp 1155–1194, 1986.

[10] Knuth, D.E., and Bendix, P., Simple Word Problems in Universal Algebras, in J.Leech (ed.), *Proceedings of the Conference on Computational Problems in Abstract Algebras*, Oxford, England, 1967, Pergamon Press, Oxford, 263–298, 1970.

[11] McCune, W., OTTER 2.0 Users Guide, Technical report, ANL-90/9, Argonne National Lab, 1990.

[12] Peterson, G. and Stickel, M, Complete sets of reductions for some equational theories, *J.ACM* 28,2, pp 233-264, 1981.

[13] Rasiowa, H., Sikorski, R., *The Mathematics of Metamathematics*, PWN, Warszawa, 1963.

[14] Rasiowa, H., *An Algebraic Approach to Non-Classical Logics*, Studies in Logic and the Fundations of Mathematics, Volume 78, North-Holland Publishing Company, Amsterdam, London - PWN, Warsaw, 1974.