J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

EQUATION MANIPULATION

Formal symbol manipulation is ubiquitous. In a broad sense, most of computer science, artificial intelligence, symbolic logic, and even mathematics can be viewed as nothing but symbol manipulation. We focus on a very narrow but useful aspect of symbol manipulation: *reasoning about equations.* Equations arise in all aspects of modeling and computation in many fields of engineering and their applications. We show how equations can be deduced from other equations (using the properties of equality) as well as how equations can be solved, both in a general framework and in some concrete cases.

In the next section, we discuss a rewrite-rule-based approach for inferring equations from other equations. Equational hypotheses are transformed into unidirectional simplification rules, and these rules are then used for rewriting. The concept of a *completion procedure* is introduced for generating a *canonical* rewrite system from a given rewrite system. A canonical rewrite system has the useful property that every object has a *unique normal form* (*canonical form*) using the rewrite system. Objects equivalent by an equality relation have the same canonical form. To infer an equation, it is necessary and sufficient to check whether the two sides of the equation have identical canonical forms.

In the third section, equation solving is reviewed. Given a set of equations in which function symbols are assumed to be uninterpreted (i.e., no special meaning of the symbols is assumed), a method is given for finding substitutions for variables that make the two sides of every equation identical. This unification problem arises in many diverse subfields of computer science and artificial intelligence, including automated reasoning, expert systems, natural language processing, and programming. In the same section, we also discuss how the algorithm is changed to exploit the semantics of function symbols when possible. In particular, we discuss changes to the unification algorithm when some function symbols are commutative, or both associative and commutative, as is often the case for many applications.

The fourth section is the longest. It reviews three different approaches for solving polynomial equations over complex numbers—*resultants*, the *characteristic set* method, and the *Gröbner basis* method. For polynomial equations with parameters, these methods can be used to identify conditions on parameters under which a system of polynomial equations has a solution. This problem comes up in many application domains in engineering, including CAD/CAM, solid modeling, robot kinematics, computer vision, and chemical equilibria. These approaches are compared on a wide variety of examples.

Equational Inference

Consider the following equations:

$$x + 0 = x \tag{1}$$

$$x + -(x) = 0 \tag{2}$$

$$(x + y) + z = x + (y + z) \tag{3}$$

In the above equations, 0 is a constant symbol, standing for the identity for $+$, a binary function symbol, and $-$ is a unary function symbol. Each variable is universally quantified; that is, any expression involving variables, the operations $-$, $+$, 0, and other generators can be substituted for the free variables.

The reader may have noticed that these equations are the defining axioms of the familiar algebraic structure *groups*. It is easy to see that the equations

$$\begin{aligned} u &= u + 0 = u + (-(u) + -(-(u))) \\ &= (u + -(u)) + -(-(u)) = 0 + -(-(u)) \end{aligned}$$

follow from the above defining axioms. By appropriately substituting expressions for variables in the above axioms, this deduction follows by the properties of equality. However, proving other equations routinely given as homework problems in a first course on abstract algebra, such as

$$\begin{aligned} 0 + u &= u \\ -(u) + u &= 0 \\ -(u + v) &= -(v) + -(u) \end{aligned}$$

or showing that $-(u + v) \neq -(u) + -(v)$, is not that easy. It requires some effort. One has to find appropriate substitutions for variables in the above axioms, and then chain them properly to derive these properties.

In a more general setting, a natural question to ask is: given a finite set of equations as the defining axioms, such as the group axioms above, can another equation be deduced from them by substituting for universally quantified variables and using the well-known properties of equality, such as reflexivity, symmetry, transitivity, and replacement of equals by equals? This is a fundamental problem in equational deduction. The answer to this problem is, in the general case, negative, as the problem is undecidable (that is, there can never exist a computer program/algorithm that can solve this problem in general, even though specific instances of it can be solved).

In many cases, however, this question can be answered. Below we discuss a particular heuristic that is often useful in finding the answer. The basic idea is not to use the above equational axioms in both directions; in other words, if an instance of its left side can be replaced by the corresponding instance of its right side, then that instance of its right side cannot be replaced by the instance of its left side, and similarly, the other way around. Instead, using some uniform well-founded measure on expressions, we determine which of the two sides in an equation is more "complex," and view an equation as a unidirectional rewrite (simplification) rule that transforms more complex expressions into less complex. We often employ such a heuristic in solving problems. For instance, the axiom

$$x + 0 = x$$

is viewed as a simplification rule in which $x + 0$ (or any other instance of it) is simplified to x, and not the other way around; that is, x is never replaced by $x + 0$.

Let us precisely define *simplification* or *rewriting*. Given a rewrite rule

$$L \rightarrow R$$

where L and R are terms built from variables and function symbols, a term t can be simplified by the rule (at position p, a sequence of nonzero positive integers,[1] in t) if the subterm t/p of t *matches* L, that is there is a substitution σ for variables in L, written as $\{ x_1 \leftarrow s_1, x_2 \leftarrow s_2, \ldots, x_n \leftarrow s_n \}$, such that $t/p = \sigma(L)$, the result

of applying the substitution σ on L . The result of this simplification is then

$$t[p \leftarrow \sigma(R)]$$

—the term obtained by replacing the subterm at position p in t by $\sigma(R)$. This definition can be extended to consider rewriting (including multistep rewriting) by a system of rewrite rules. A term t is in *normal form* (also called *irreducible*) with respect to a rule $L \rightarrow R$ (respectively, a system of rewrite rules) if no subterm in t matches L (respectively, the left side of any rewrite rule in the system).

It should be noted that for rewriting to be meaningfully defined, the variables appearing in the right side R must also appear in the left side L, as otherwise substitutions for extra variables appearing in R cannot be determined. Henceforth, every rule is assumed to satisfy this property; i.e., all variables appearing in R must appear in L as well.

For example,

$$(u + -(u)) + -(-(u)) = u$$

can be proved easily by simplifying the left side: first using the axiom in Eq. (3), followed by the axiom in Eq. (2), and then by axiom in Eq. (1), using each axiom as a left-to-right rewrite rule.

For certain equations, it is not too difficult to determine which side is more complex (e.g., the first two axioms). But there are cases in which it is not easy.[2]

Consider the case of the associativity axiom. The left side is more complex if left-associative expressions are considered simpler; if right-associative expressions are considered simpler, then the left side is less complex than the right side. In this sense, there is sometimes a choice for certain equations. If all equational axioms can be oriented into terminating rewrite rules (meaning that simplification using such rewrite rules always terminates), then simplification by rewriting itself serves as a useful heuristic for checking whether the given two terms are equal.

Notice that there is another way to simplify the left side of the above equation, which is to first simplify using the axiom in Eq. (2), giving a new equation

$$0 + -(-(u)) = u$$

Clearly, both u and $0 + -(-(u))$ are normal forms of $(u + -(u)) + -(-(u))$ with respect to the above axioms considered as left-to-right rewrite rules. This would imply that $u = 0 + -(-(u))$ is an equation following from the above axioms as well.

For attempting proofs of other equations from equational axioms, the following two properties of equations are helpful. The first property is whether equational axioms can be collectively oriented into terminating rewrite rules. The second property is whether the rewrite rules thus obtained have the so-called *Church–Rosser* or *confluence* property: that is, given an expression, no matter how it is simplified using rewrite rules, if there is a normal form, that normal form is unique. Both of these properties are undecidable in the general case. However, for terminating rewrite rules, the confluence property is decidable.

As discussed earlier, the equational axioms of groups can be oriented as terminating rewrite rules (by going from left to right). These rewrite rules do not have the confluence property, as we saw above that the expression $(u + -(u)) + -(-(u))$ has two different normal forms.

If a set of rewrite rules is terminating and confluent, then every expression has a normal form, and further, that normal form is unique; unique normal forms are also called *canonical* forms. For such rewrite rules, it is easy to decide whether an equational formula follows from the axioms or not: compute the unique normal forms of the expressions on the two sides of the equality; if the normal forms are identical, the equational formula is a theorem; otherwise, it is not a theorem.

For groups, the following set of rewrite rules is terminating and confluent; this system thus serves as a decision procedure for equational formulas involving $+, -, 0$:

$$
\begin{aligned}
x + 0 &\rightarrow x && (1)\\
x + -(x) &\rightarrow 0 && (2)\\
(x + y) + z &\rightarrow x + (y + z) && (3)\\
-(0) &\rightarrow 0 && (4)\\
-(-(x)) &\rightarrow x && (5)\\
-(x) + x &\rightarrow 0 && (6)\\
0 + x &\rightarrow x && (7)\\
x + (-(x) + y) &\rightarrow y && (8)\\
-(x) + (x + y) &\rightarrow y && (9)\\
-(x + y) &\rightarrow -(y) + -(x) && (10)
\end{aligned}
$$

Given a set of equational axioms, it is sometimes possible to generate an *equivalent* terminating and confluent rewriting system from them. The generated system is equivalent to the input system in the sense that the equations provable from the equations corresponding to rewrite systems are precisely the equations provable from the original equational axioms. This can be done using *completion* procedures.

In the next subsection, we discuss heuristics for ensuring/checking termination of rewrite rules. In the following subsection, we discuss the concepts of *superpositions* and *critical pairs* for checking the confluence of terminating rewrite rules. This is followed by a discussion of completion procedures for generating an equivalent confluent and terminating rewrite system from a given terminating rewrite system. Finally, we discuss advanced concepts relating to generalizations of these techniques when semantic information of some function symbols must be exploited. The special focus is on systems in which certain function symbols have the *associative and commutative* (*AC*) properties.

These methods and heuristics can be implemented and tried on a variety of examples. We have built an automated reasoning program called *Rewrite Rule Laboratory* (*RRL*) for checking termination of a large class of rewrite systems, as well as for generating a confluent and terminating rewrite system from a given rewrite system using completion. This program has been used for solving many nontrivial problems in equational inference, and has also been used in a variety of applications, including: automatic verification of hardware arithmetic circuits, such as the SRT division algorithm widely believed to have been implemented in the Pentium chip; software verification; analysis of database integrity constraints; checking consistency; and completeness of behavioral specifications of abstract data types. For details and citations, an interested reader may refer to Refs. 1 and 2.

Termination of Rewriting. Checking for termination of a set of rewrite rules is undecidable, in general. In fact, the termination of a single rewrite rule is undecidable, as a Turing machine can be encoded using just one rewrite rule. However, for a large class of interesting rewrite systems, heuristics have been developed to check their termination.

As stated above, one approach for checking termination is to define a complexity measure on expressions by mapping them to a well-founded set (i.e., a set that does not admit any infinite descending chain) such that for every rule, its right side is of smaller measure than its left side. Since these rewrite rules are used for simplification, such a measure should satisfy some additional properties, namely, in any context (larger expression), whenever the left side of a rule is replaced by the right side, the measure reduces, and similarly,

the measure of every instance of the right side of a rule is always smaller than the measure of the corresponding instance of its left side.

In their seminal paper discussing a completion procedure, Knuth and Bendix 3 introduced a measure by associating *weights* with expressions by assigning weights to function symbols. They required that for an expression $s > t$, every variable in s must have at least as many occurrences as in t . This idea was extended by Lankford to design a complexity measure by associating polynomials with function symbols.

A more commonly used measure is *syntactic*, based on a precedence relation among function symbols. Assuming that function symbols can be compared, terms built using these function symbols are compared by recursively comparing their subterms. These *path* orderings are quite powerful in the sense that termination of all primitive recursive function definitions, as well as other definitions such as of Ackermann's function, which grows faster than any primitive recursive function, can be established using these orderings. For a function symbol that takes more than one argument, if two terms have that function symbol as their root, then the arguments can be compared without considering their order, left-to-right order, right-to-left order, or any other permutation. A commonly used path ordering based on these ideas is the *lexicographic recursive path ordering* (4); this is the ordering implemented in our theorem prover *RRL*.

Let $\succ_{\mathrm{f}}$ be a well-founded precedence relation on a set of function symbols F ; function symbols can also have equivalent precedence, written $f \sim_{\mathrm{f}} g$. The lexicographic recursive path ordering (with status) $\succ_{\mathrm{rpo}}$ extends $\succ_{\mathrm{f}}$ to a well-founded ordering on terms:

Definition 1. $s = f(s_1, \ldots, s_m) \succ_{\mathrm{rpo}} g(t_1, \ldots, t_n) = t$ iff one of the following holds.

(1) $f \succ_{\mathrm{f}} g$, and $s \succ_{rpo} t_j$ for all j $(1 \leq j \leq n)$.

(2) For some i $(1 \leq i \leq n)$, either $s_{\mathrm{i}} \sim_{\mathrm{rpo}} t$ or $s_i \succ_{rpo} t$

(3) $f \sim_{\mathrm{f}} g, f$ and g have multiset status, and $\{\{s_1, \ldots, s_m\}\} \succ_{mul} \{\{t_1, \ldots, t_n\}\}$

(4) $f \sim_{\mathrm{f}} g, f$ and g have left-to-right status, then $(s_1, \ldots, s_m) \succ_{lex} (t_1, \ldots, t_{\mathrm{n}})$, and $s \succ_{rpo} t_{\mathrm{j}}$ for all j $(1 \leq j \leq n)$; if f and g have right-to-left status, then $(s_m, \ldots, s_1) \succ_{lex} (t_n, \ldots, t_1)$ and $s \succ_{rpo} t_j$ for all j $(1 \leq j \leq n)$.

Of course, $s \succ_{rpo} x$ if s is nonvariable, and the variable x occurs in s . Term $s \sim_{\mathrm{rpo}} t$ if and only if $f \sim_{\mathrm{f}} g$ and

(1) f and g have multiset status, and $\{\{s_1, \ldots, s_{\mathrm{m}}\}\} = \{\{t_1, \ldots, t_{\mathrm{n}}\}\}$, or

(2) $f \sim_{\mathrm{f}} g, f \sim$ and g have left-to-right (similarly, right-to-left) status, and for each $1 \leq i \leq m$, $s_{\mathrm{i}} \succ_{rpo} t_{\mathrm{i}}$.

In the above definitions, $\succ_{rpo}$ is recursively defined using its multiset and sequence extensions. The multiset $M_1 \succ_{mul} M_2$ if and only if for every $t_i \in M_2 - M_1$, where $-$ is the multiset difference, there is an $s_{\mathrm{j}} \in M_1 - M_2$, $s_{\mathrm{j}} \succ_{rpo} t_i$. The sequence $(s_1, \ldots, s_m) \succ_{\mathrm{lex}} (t_1, \ldots, t_n)$ if and only if there is a $1 \leq i \leq m$ such that for all $1 \leq j < i$ one has $s_{\mathrm{j}} \succ_{rpo} t_j$ and $s_i \succ_{rpo} t_i$.

It can be shown that $\succ_{rpo}$ has the desired properties of a measure needed to ensure that rewrite rules used for simplification are indeed terminating. So if the left side of an equation is $\succ_{rpo}$ its right side, it can be oriented from left to right as a terminating rewrite rule. The properties are as follows:

- $\succ_{rpo}$ is well founded,
- $\succ_{rpo}$ is *stable* under substitutions, that is, if $s \succ_{rpo} t$, then for any substitution σ of variables, $\sigma(s) \succ_{rpo} \sigma(t)$, and
- $\succ_{rpo}$ is *preserved under contexts*, that is, $s \succ_{rpo} t$, then for any term c that has a position p, one has $c[p \leftarrow s] \succ_{rpo} c[p \leftarrow t]$ as well.

Local Confluence of Rewriting: Superposition and Critical Pairs. Since bidirectional equations are used as unidirectional rewrite rules for efficiently exploring search space, additional rewrite rules are needed to compensate for the lack of symmetric use of equations. For instance, we saw above an example of an expression, $(u + -(u)) + -(-(u))$, which could be simplified in two different ways—using the axiom in Eq. (3) or using the axiom in Eq. (2). Depending upon the axiom used, two different normal forms can be computed from the expression, thus showing that the rewrite system obtained from the axioms is not *confluent*. However, $0 + -(-(u)) = u$ can be inferred from the three axioms.

A rewrite system $\mathscr{R}$ is called *confluent* if for every term t, if t is simplified by $\mathscr{R}$ in *many* steps in two different ways to two different results, it is always possible to simplify the results to the same expression. Checking for confluence is, in general, undecidable. However, for a terminating rewrite system, the confluence check is equivalent to *local confluence*, which can be easily decided based on the concepts of *superpositions* and *critical pairs* generated by overlapping the left sides of rewrite rules. A rewrite system $\mathscr{R}$ is called *locally confluent* if for every term t, if t is simplified in a *single* step in two different ways to two different results, it is always possible to simplify the results to the same expression.

Theorem 2. If a terminating rewrite system $\mathscr{R}$ is locally confluent, then $\mathscr{R}$ is confluent.

Definition 3. Give two rules $L_1 \to R_1$ and $L_2 \to R_2$, not necessarily distinct, such that a nonvariable subterm of L_1 at position p unifies with L_2 with a most general substitution (unifier) σ, $\sigma(L_1)$ is called a superposition of $L_2 \to R_2$ into $L_1 \to R_1$ and

$$\langle \sigma(R_1), \sigma(L_1)[p \leftarrow \sigma(R_2)] \rangle$$

is called the associated critical pair.

(Unification of terms is defined in the next section.)

Theorem 4. Give a rewrite system $\mathscr{R}$ if for every critical pair $\langle c_1, c_2 \rangle$ generated from every pair of $L_1 \to R_1$, $L_2 \to R_2$ in $\mathscr{R}$, c_1 and c_2 have the same normal form, then $\mathscr{R}$ is locally confluent.

As an illustration, consider the axioms in Eqs. (2) and (3) above, viewed as left-to-right unidirectional rules. Using *unification* (see the next section), the superposition $(x + -(x)) + z$ [of rules 2 and 3 corresponding to the axioms in Eqs. (2) and (3), respectively] can be generated from the overlap of the two left sides; when axiom 2 is applied on it, the result is $0 + z$, but if the axiom in Eq. (3) is applied on the same expression, the result is $x + (-(x) + z)$. The pair $\langle x + (-(x) + z), 0 + z \rangle$ is the critical pair obtained from the superposition. Such pairs are critical in determining the confluence property; hence the name. In this example, the expressions in the above critical pair have different normal forms (both $0 + z$ and $x + (-(x) + z)$ are already in normal form). So the rewrite rules corresponding to the three group axioms are not confluent, even though they are terminating.

In contrast, the system of 10 rewrite rules given above is confluent; it can be shown that every superposition generated from possible overlaps between the left sides of these rewrite rules generates critical pairs in which the two terms have the same normal form. For instance, terms in the pair $\langle x + (-(x) + z), 0 + z \rangle$ reduce to the same normal form using the ten rewrite rules above.

Identifying which superpositions are essential and which are redundant for checking local confluence as well as in completion procedures has been a fruitful area of research in rewrite-rule-based automated deduction. Implementation of such results speeds up generation of canonical rewrite systems using the completion procedure as discussed below. As the reader might have observed, superpositions and critical pairs are defined by unifying a nonvariable subterm of the left side of a rule with the left side of another rule, since variable subterms always result in pairs that rewrite to the same expression. This research of identifying and discarding unnecessary inferences has recently been found useful in other approaches to automated deduction as well, including resolution-based calculi.

Completion Procedure: Making Rewrite Systems Canonical. As the reader might have guessed, if a terminating rewrite system is not confluent, it may be possible to make it confluent by augmenting it with additional rewrite rules obtained from normal forms of critical pairs. An equation between the two different normal forms of the terms in a critical pair obtained from a superposition is an equational consequence of the original rules. Including it in the original set of equations does not, in any way, change the equational theory of the original system. An equational theory associated with a set of equations is defined as the set of all equations that can be derived from the original set of equations using the rules of equality and instantiation of variables.

This process of adding new equational consequences generated from superpositions and critical pairs from rewrite rules is called *completion*. Every nontrivial new equation generated must be oriented into a terminating rewrite rule. If this process terminates successfully, then the result is a terminating and confluent rewrite system (also called a *canonical* or *complete* rewrite system), which serves as a decision procedure for the equational theory of the original set of equations. An equation $s = t$ is an equational consequence of the original system if and only if the normal forms of s and t with respect to the canonical rewrite system are the same. A canonical rewrite system thus associates canonical forms with congruence classes induced by a set of equations. For example, from the equational axioms defining groups, the above set of ten rewrite rules can be obtained by completion from the original set of three rewrite rules; this set can be generated by our theorem prover *RRL* in less than a second.

A completion procedure can be viewed as an implementation of an inference system consisting of the following inference steps (5). Each inference step transforms a pair consisting of a set $\mathscr{E}$ of equations and a set $\mathscr{R}$ of rules. The initial state is $\mathscr{E}_0$ and $\mathscr{R}_0$, with $\mathscr{R}_0$ usually being $\{\ \}$, the empty set. An inference step transforms $\langle \mathscr{E}_\mathrm{i}, \mathscr{R}_\mathrm{i} \rangle$ to $\langle \mathscr{E}_\mathrm{i} + 1, \mathscr{R}_{i+1} \rangle$ using a termination ordering $\succ$ on terms, as follows:

(1) *Process an Equation.* Given an $e_1 = e_2$ in $\mathscr{E}_\mathrm{i}$, let n_1 and n_2 be, respectively, normal forms of e_1 and e_2 using $\mathscr{R}_\mathrm{i}$.

(1) *Delete a Redundant Equation.* If $n_1 \equiv n_2$, then $\mathscr{E}_{i+1} = \mathscr{E}_\mathrm{i} - \{e_1 = e_2\}$ and $\mathscr{R}_{i+1} = \mathscr{R}_\mathrm{i}$ ($n_1 \equiv n_2$ stands for n_1 and n_2 being identical.)
(2) *Add a New Rule.*

(1) If $n_1 \succ n_2$, then $\mathscr{E}_{i+1} = \mathscr{E}_\mathrm{i} - \{e_1 = e_2\}$ and $\mathscr{R}_{i+1} = \mathscr{R}_\mathrm{i} \cup \{n_1 \to n_2\}$.
(2) If $n_2 \succ n_1$, then $\mathscr{E}_{i+1} = \mathscr{E}_i - \{e_1 = e_2\}$ and $\mathscr{R}_{i+1} = \mathscr{R}_i \cup \{n_2 \to n_1\}$.

Introduce a New Function. Let f be a new function symbol not in the theory, and let $x_1, \ldots, x_j$ be the common variables appearing in n_1 and n_2 .

(1) If $n_1 \succ f(x_1, \ldots, x_j)$, then $\mathscr{E}_{i+1} = \mathscr{E}_\mathrm{i} - \{e_1 = e_2\} \cup \{n_2 = f(x_1, \ldots, x_j)\}$ and $\mathscr{R}_{i+1} = \mathscr{R}_\mathrm{i} \cup \{n_1 \to f \sim (x_1, \ldots, x_j)\}$.
(2) If $f(x_1, \ldots, x_j) \succ n_1$, then $\mathscr{E}_{i+1} = \mathscr{E}_\mathrm{i} - \{e_1 = e_2\} \cup \{n_2 = f(x_1, \ldots, x_j)\}$ and $\mathscr{R}_{i+1} = \mathscr{R}_i \cup \{f(x_1, \ldots, x_j) \to n_1\}$. This choice should not be taken. Almost always, $f(x_1, \ldots, x_j)$ should be made the right side of introducing rules.

Add Critical Pairs. Given two (not necessarily distinct) rules $l_1 \to r_1$ and $l_2 \to r_2$ in $\mathscr{R}_i$,

$$\mathscr{R}_{i+1} = \mathscr{R}_i$$

and

$$\mathscr{E}_{i+1} = \mathscr{E}_i \cup \{c_1 = c_2 \mid \langle c_1, c_2 \rangle \text{ is a critical pair of } l_1 \to r_1 \text{ and } l_2 \to r_2\}$$

Normalize Rules. Given two distinct rules $l_1 \to r_1$ and $l_2 \to r_2$ in $\mathscr{R}_i$:

(1) If $l_1 \to r_1$ rewrites l_2, then $\mathscr{E}_{i+1} = \mathscr{E}_i \cup \{l_2 = r_2\}$ and $\mathscr{R}_{i+1} = \mathscr{R}_i - \{l_2 \to r_2\}$. The rule $l_2 \to r_2$ is thus deleted from $\mathscr{R}_i$ and inserted as an equation into $\mathscr{E}_i$.
(2) If $l_1 \to r_1$ rewrites r_2, then $\mathscr{E}_{i+1} = \mathscr{E}_i$ and $\mathscr{R}_{i+1} = \mathscr{R}_i - \{l_2 \to r_2\} \cup \{l_2 \to r'_2\}$, where r'_2 is a normal form of r_2 using $\mathscr{R}_i$.

The above steps can be combined in many different ways to generate a complete system when all critical pairs among rules have been considered. The resulting algorithm must be *fair* in the sense that critical pairs among all pairs of rules must eventually be considered. Many useful heuristics have been explored and developed to make completion faster. The order in which rules are considered for computing superpositions, the order in which critical pairs are processed, how new rules are used to simplify already generated rules, and so on, appear to affect the performance of completion considerably. For some early work on this, the reader may consult Ref. 6.

There are at least two ways a completion procedure can fail to generate a canonical rewrite system:

(1) An equational consequence is generated that cannot be oriented into a terminating rewrite rule.
(2) Even if all equational consequences generated from critical pairs during completion are orientable, the process of adding new rules does not appear to terminate.

The first condition could arise for many reasons. The new equation, when oriented either way, might result, in conjunction with other rules, in nontermination of rewriting. Or the ordering heuristics might not be powerful enough to establish the termination of the rewrite system when augmented with the new rewrite rule. It is also possible that the generated equation could not be made into a rewrite rule because either side has extra variables that the other side does not have.

In some cases, it is helpful to split such an equation by introducing a new function symbol (to stand for the operation corresponding to one of the sides of the equation); this is done in the "Introduce a New Function" step above. For instance, the following single axiom can be shown to characterize free groups using completion:

$$(x/((((x/x)/y)/z)/(((x/x)/x)/z))) = y$$

where/corresponds to the right division operation. During completion of the above axiom, an equation

$$x/x = y/y$$

is generated, which suggests that a new constant can be introduced. That constant turns out to satisfy the properties of the identity. Similarly, an inverse function symbol is introduced, finally leading to the following

canonical system:

$$y/y \rightarrow f2 \quad (11)$$
$$f1(y,x) \rightarrow f3(y) \quad (12)$$
$$x/f2 \rightarrow x \quad (13)$$
$$f3(f2) \rightarrow f2 \quad (14)$$
$$f3(f3(x)) \rightarrow x \quad (15)$$
$$f2/y \rightarrow f3(y) \quad (16)$$
$$((y/z)/f3(z)) \rightarrow y \quad (17)$$
$$((y/f3(y1))/y1) \rightarrow y \quad (18)$$
$$f3((y/x)) \rightarrow (x/y) \quad (19)$$
$$(y/(x/z)) \rightarrow ((y/f3(z))/x) \quad (20)$$

Above, $f2$ behaves as the identity 0; $f3$ behaves as the inverse $-$; $f1$ is an extra function symbol introduced during completion; $x + y$ can be defined as $x/f3(y)$. All the rewrite rules corresponding to the canonical system of free groups presented earlier are equational consequences of the above canonical system.

Another approach for handling nonorientable equations is discussed below in which such equations are processed semantically. Finally, an equation could be kept as is in $\mathscr{E}_i$, and its instances could be oriented and used for simplification as needed by including them in $\mathscr{R}_i$; this is the approach taken in *unfailing* completion (7).

The second condition above (non-termination of completion) cannot, in general, be avoided, since the decision problem of equational theories is unsolvable. In some cases, however, introducing a new function symbol to stand for an expression is helpful even though intermediate equations can be oriented as terminating rewrite rules. With the help of this new symbol, it may be possible to generate a finite canonical system though no such system can be generated without it (see Ref. 8 for such an example of a finitely presented semigroup).

Forward versus Backwards Reasoning. A completion procedure employs *forward reasoning* and *saturation*. That is, additional consequences are derived from the original set of axioms, without considering conjectures/goal(s) being attempted. And this process is continued until the resulting set of derivations is completely saturated.

A completion procedure can also be used in a goal-directed way. A conjecture to be attempted is negated, and a proof by contradiction is attempted. Axioms interact with each other by forward reasoning, generating new consequences. They can also interact with the negation of the goal, and attempt to generate a contradiction using *backward reasoning*. This approach based on the completion procedure can be shown to be semidecidable for determining membership in an equational theory. In other words, if a conjecture is provable, then barring any difficulties arising due to nonorientability of new equations,[3]

a proof by contradiction can always be generated by the completion procedure.

To illustrate using the group example again, one way to attempt a proof of $0 + x = x$ is to

(1) first complete the above axioms using a completion procedure that generates a complete set of rewrite rules, and then
(2) simplify both sides of the conjecture, and check for identity.

If a completion procedure does not terminate, then the second step will never be performed. To circumvent this problem, an obvious heuristic is to do step 2 whenever a new rule is generated, to see whether with the existing set of rules, the conjecture can be proved by rewriting. The same approach can be tried in a uniform

way by adding the negation of the (Skolemized) conjecture (for the example, $0 + a \neq a$), and then running completion on the axioms and the negated conjecture, and looking for a contradiction.

Unorientable Equations. So far, all equations are assumed to be oriented as terminating rewrite rules. In many applications, one often has to consider function symbols that are commutative and/or associative. Commutative axioms (and associative axioms in conjunction with commutative axioms) are nonterminating when used for simplification. For example, consider the axioms defining abelian groups, which, in addition to the three axioms in Eqs. (1, ,), include the commutative axiom as well:

$$x + y = y + x$$

One approach for handling axioms such as commutativity and associativity is to integrate them into the definitions of rewriting (simplification) and superpositions. In the definition of rewriting, instead of looking for an exact match of the left side of a rewrite rule $L \rightarrow R$, matching is done modulo such axioms. In the discussion below, we assume that certain function symbols have the associative and commutative properties.

A term t rewrites at position p using a rewrite rule $L \rightarrow R$ (where t and L could have occurrences of function symbols with associative and commutative properties) if there is a $t' =_{AC} t$ such that $t'/p =_{AC} \sigma(L)$, where $=_{AC}$ is the congruence relation defined by the associative–commutative properties of those function symbols on terms; t is then said to rewrite to $t[p \leftarrow \sigma(R)]$.[4]

For instance, $-(u) + (u + w)$ can be simplified by associative–commutative simplification using the axiom $x + -(x) = 0$ to $0 + w$, which simplifies further using the axiom $x + 0 = x$ to w . Notice that the term $-(u) + (u + w)$ is first rearranged using the associativity property of $+$ to the equivalent term $(-(u) + u) + w$. The subterm $-(u) + u$ then matches modulo commutativity to $x + -(x)$. Similarly, the result $0 + w$ matches modulo commutativity to $x + 0$.

Similarly, superpositions among the left sides of rewrite rules are defined using unification modulo axioms. Further, to ensure confluence of terminating rewrite rules, additional checks must be made (which can be viewed as considering specialized superpositions between each rule and the axioms to account for the semantics) (9,10). In the next section, we discuss unification modulo associative–commutative properties as well. Termination heuristics must be generalized appropriately also; termination of rewrite rules modulo axioms, such as associative–commutative, means defining a well-founded ordering on equivalence classes defined by the axioms (11).

Below is a list of rewrite rules of abelian groups which is canonical modulo AC.

$$\begin{aligned} x + 0 &\rightarrow x && (1)\\ x + -(x) &\rightarrow 0 && (2)\\ -(0) &\rightarrow 0 && (4)\\ -(-(x)) &\rightarrow x && (5)\\ -(x + y) &\rightarrow -(y) + -(x) && (10) \end{aligned}$$

This list is much smaller than the list for (general, abelian as well as nonabelian) groups, since one rewrite rule here, namely $x + 0 \rightarrow x$, is equivalent to both $x + 0 \rightarrow x$ and $0 + x \rightarrow x$ because of the commutativity property of $+$. Using the above rewrite rules, it can proved that $-(x + y) = -(x) + -(y)$, a property true for abelian groups but not for nonabelian groups.

Below is another list of canonical rewrite rules of abelian groups. Notice that the rule in Eq. (10) above is oriented in the opposite direction below:

$$x + 0 \rightarrow x \qquad (1)$$
$$x + -(x) \rightarrow 0 \qquad (2)$$
$$-(0) \rightarrow 0 \qquad (4)$$
$$-(-(x)) \rightarrow x \qquad (5)$$
$$x + -(x + y) \rightarrow -(y) \qquad (21)$$
$$-(y) + -(x) \rightarrow -(x + y) \qquad (22)$$
$$-(x + -(y)) \rightarrow y + -(x) \qquad (23)$$

The issue of redundant superpositions becomes very critical when confluence check and completion procedure modulo a set of axioms are considered. In general, there are a lot more superpositions to consider, because unlike the "ordinary" case, there can be many most general unifiers of two terms. Further, rewriting modulo axioms is a lot more expensive than "ordinary" rewriting. It was control over this redundancy which led us to easily prove ring commutative problems using *RRL* (12). In his EQP program, McCune extensively exploited heuristics for discarding redundant superpositions to establish that Robbins algebras are Boolean (13), thus solving a long-standing open problem in algebra and logic.[5]

Without these optimizations, it is unclear whether EQP would have been able to settle this long-standing conjecture.

The concept of a completion procedure and related properties generalize as well. For instance, from the first two axioms of groups discussed above (since the associativity axiom in addition to the commutativity axiom is semantically built-in), the canonical system consisting of the five rules in Eqs. (1), (2), (4), (5), (10) above can be generated using the completion procedure for associative–commutative function symbols. Our theorem prover *RRL* can generate this canonical system in a few seconds. For details about an associative–commutative completion procedure as well as for an $\mathscr{E}$completion procedure, where $\mathscr{E}$ is a first-order theory for which $\mathscr{E}$-unification and $\mathscr{E}$-rewriting are decidable, an interested reader can consult 9,10. In a later section, we discuss the Gröbner basis algorithm, which is a specialized completion procedure for finitely generated polynomial rings in which coefficients are from a field. This completion procedure has the nice property that it always terminates.

Extensions. We have discussed proofs in equational theories, in which only the properties of equality are used. When function symbols are defined by recursive equations on inductively (recursively) defined data structures, equations must be proved by induction as well. For example, if addition on numbers is defined recursively as

$$x + 0 = x, \qquad x + s(y) = s(x + y)$$

where s is the successor function on numbers, then $0 + x = x$ cannot be proved equationally from the definition. But $0 + x = x$ can be proved by induction; that is, no matter what ground term built using 0 and s is substituted for x in the above equation, the resulting equation is an equational inference of the two equations defining $+$. Proofs by induction have been found useful in verification and specification analysis.

There are many approaches for mechanizing proofs of equational formulae by induction, in which some of the variables are ranging over inductively (recursively) defined data structures such as natural numbers, integers, lists, sequences, or trees (14,15). In the following sub-subsection, we will briefly review the *cover-set* approach implemented in *RRL* (15), which is closely related to the approach in 14 in that the induction

scheme used in attempting a proof of a conjecture is derived from the well-founded ordering used to show the termination of the definitions of function symbols appearing in the conjecture.

We will not be able to discuss full first-order inference, in which properties of other logical connectives are used for deduction. This is a very active area of research, with many conferences and workshops. First-order theorem proving can be mechanized equationally as well. In fact, this was proposed by the author in collaboration with Paliath Narendran (16), in which quantifier-free first-order formulae are written as polynomials over a Boolean ring, using $+$ to represent exclusive or and $*$ to represent conjunction. Our theorem prover *RRL* includes an implementation of this approach to first-order theorem proving.

Inductive Inference: Cover-Set Method. The cover-set method for mechanizing induction is based on analysis of the definitions of function symbols appearing in a conjecture. The definition of a function symbol as a finite set of terminating rewrite rules is used to design an induction scheme based on the well-founded ordering used to show the termination of the function definition. The induction hypotheses are generated from the recursive calls in the definition, which are lower in the well-founded order. In the case of competing induction variables and induction schemes, heuristics are employed to pick the variable and the associated induction scheme most likely to succeed. Most induction theorem provers use backtracking so that if one particular choice fails, another choice can be attempted.

Let us start with a simple example. Assume that functions $+$, $*$, and x^y are defined on natural numbers, generated by 0 and s, where s is the successor operation to denote incrementing its argument by 1, as follows:

$$\begin{aligned} x + 0 &= x \\ x + s(y) &= s(x + y) \\ x * 0 &= 0 \\ x * s(y) &= x + (x * y) \\ x^0 &= s(0) \\ x^{s(y)} &= x * x^y \end{aligned}$$

Without assuming any additional properties of $+$, $*$, x^y, we wish to prove from the above definitions that

$$x^{(y+z)} = x^y * x^z$$

It is easy to see that the above defining equations, when oriented from left to right as terminating rewrite rules, are confluent as well. Further, since the two sides of the above conjecture are already in normal form, it is clear that the conjecture is not provable by equational reasoning; that is, the conjecture is not in the equational theory of the definitions.

The conjecture can, however, be proved by induction, using the property that every natural number can be generated *freely* by 0 and s . Our theorem prover *RRL*, for instance, can prove the above conjecture automatically without any guidance or help; it generates the needed intermediate lemmas, including the associativity and commutativity of $*$, $+$, and so on.

While attempting a proof by induction of a conjecture (by hand or automatically), a number of issues need to be considered:

(1) Which variable in a conjecture should be selected for performing induction? Associated with this choice is determining an induction scheme to be used.

(2) What mechanisms can be attempted for generalizing intermediate subgoals so as to have stronger lemmas, which are likely to be more useful?
(3) How does one ensure whether progress is being made while trying subgoals generated from following a particular approach, and when should an alternate approach be attempted instead?
(4) When should one give up?

In the above example, there are three candidates for an induction variable. By performing definition analysis, it can be easily determined that if x or y is chosen as a induction variable, a proof attempt will get stuck, since the definitions above are given using recursion on the second argument of $+$, $*$, and x^y . Thus the most promising variable to be used for doing induction is z .

The induction scheme to be used is suggested by the definitions of the function symbols $+$ and x^y, in which z appears as an argument. Since each of these definitions can be proved to be terminating, a well-founded ordering used to show termination of the definition can be used to design an induction scheme. For this example, the induction schemes suggested by the definitions of $+$ and x^y are precisely the principle of mathematical induction, namely, that to prove the above conjecture, it suffices to show that the conjecture can be proved for the case when

(1) $z = 0$—the *basis subgoal*, and
(2) $z = s(z')$, using the induction hypothesis obtained by making $z = z'$—the *induction step subgoal*.

In general, for each rewrite rule in the function definition used for designing the induction scheme, a subgoal is generated. A rewrite rule whose right side does not have a recursive call to the function gives a basis subgoal. A rewrite rule whose right side invokes recursive calls to the function gives an induction step, in which the changing arguments in the recursive calls generate substitutions for producing potentially useful induction hypotheses. In general, there can be many basis subgoals, and in an induction subgoal there can be many induction hypotheses.

For the above example, the basis subgoal can be easily proved by normalizing it using the rewrite rules. The induction subgoal after normalization produces

$$x * x^{(y+z')} = x^y * (x * x^{z'})$$

assuming the induction hypothesis

$$x^{(y+z')} = x^y * x^{z'}$$

Using the induction hypothesis, the conclusion in the subgoal simplifies to

$$x * (x^y * x^{z'}) = x^y * (x * x^{z'})$$

Most induction theorem provers, including *RRL*, would attempt to generalize this intermediate conjecture using a simple heuristic of abstracting common subexpressions on the two sides by new variables. For instance, the above intermediate conjecture would be generalized to

$$x * (u * v) = u * (x * v)$$

This conjecture cannot be proved equationally either, but can be proved by induction. The variable v is chosen as the induction variable by recursion analysis. The basis subgoal can be easily proved. The induction step subgoal leads to the following intermediate subgoal to be attempted:

$$x * (u + (u * v')) = u * (x + (x * v'))$$

assuming

$$x * (u * v') = u * (x * v')$$

In the conclusion above, it is not even clear how to use the induction hypothesis.

Another induction on the variable v' (which is an obvious choice based on recursion analysis) leads to a basis subgoal when $v' = 0$:

$$x * u = u * x$$

the commutativity of $*$. The induction step, after generalization, leads to an intermediate conjecture:

$$x + y = y + x$$

which can be easily established by induction. Using these intermediate conjectures, the original intermediate conjecture

$$x * (u + (u * v')) = u * (x + (x * v'))$$

is proved, from which the proof of the main goal follows.

As the reader might have guessed, the major challenge during proof attempts by induction is to generate/speculate lemmas likely to be found useful for the proof attempt to make progress.

More details about the cover-set induction method can be found in Ref. 15; see Ref. 2 for the use of the cover-set method for mechanically verifying parametrized generic arithmetic circuits of arbitrary data width. For mechanizing inductive inference about Lisp-like functions, an interested reader may consult Ref. 14.

Equation Solving Over Terms: Unification

In the last section, we discussed whether an equation can be deduced from a set of equations using the properties of equality, with the assumption that any expression can be substituted for a variable, that is, variables in hypothesis equations are assumed to be universally quantified. In equation solving, one is often interested in a particular instance of such variables; that is, variables in equations are assumed to be existentially quantified. In this section, we focus on this aspect of symbolic computation.

Given a set of equations over terms involving variables and function symbols, we are interested in solving these equations, that is, finding a substitution for variables appearing in the equations so that after the substitution is made, both sides of every equation become identical. At first, nothing is assumed about the meaning of the function symbols appearing in the equations (such symbols are called *uninterpreted*). In a later subsection, we assume that as in the case of $+$ in abelian groups, some of the function symbols have

the associative–commutative properties. In a subsequent section, we discuss solving polynomial equations, and assume even more about the operators, namely that $+$, $*$ are, respectively, addition and multiplication on numbers.

For example, consider an equation

$$f(a, g(x, y)) = f(x, z)$$

This equation can be solved, and it has many solutions. One solution is

$$\{x \leftarrow a,\ \ z \leftarrow g(a, y)\}$$

In fact, this solution is the most general solution, and any other solution can be obtained by instantiating the variables in it.

Another equation,

$$f(a, g(x, y)) = f(x, g(b, z))$$

does not have any solution, since x cannot be made equal to both a and b (recall that the function symbols are assumed to be uninterpreted, so it cannot be assumed that a could possibly be equal to b) .

In the previous section, we saw an application of unification for considering overlaps among the left sides of rewrite rules to compute superpositions and critical pairs. Later, we will summarize other applications of unification as well.

Simple Unification. Consider a finite set of equations

$$\{s_1 = t_1, \ldots,\ \ s_k = t_k\}$$

The goal is to find whether they have a common *solution*; i.e., whether there is a substitution σ for variables in $\{s_i, t_i \mid 1 \leq i \leq k\}$ such that for each i, $\sigma(s_i)$ and $\sigma(t_i)$ are the same. If so, what is a most general solution; that is, what is a solution from which all other solutions can be found by instantiating the variables? A solution of these equations is called a *unifier*, and a most general solution is called a *most general unifier* (mgu).

Just as in the case of solving linear equations (in linear algebra), it is necessary to be clear about the *simplest* equations for which there is a solution as well as for equations that do not have any solution.

Similarly to $x = 3$ in linear algebra, the equation $x = t$, where x does not appear in t, has a solution, and in fact, the most general solution is $\{x \leftarrow t\}$. Such an equation is said to be in *solved form.* Similarly to $3 = 0$ having no solution, the equation $f(\ldots) = g(\ldots)$, where f , g are different function symbols, has no solution. No substitution for variables can make the two sides equal, as no assumption can be made about the properties of distinct function symbols.

Also, an equation $x = t$, where t is a nonvariable term with an occurrence of x, has no solution, since no substitution for x can make the two sides of the equation equal. After the substitution, the size of the left side is not equal to the size of the right side.

The general problem of solving a finite set of equations can be transformed into those of the above simple equations by a sequence of transformation steps. During the transformation, if a simple equation $x = t$, with t having no occurrence of x, is identified, then the partial solution obtained so far can be extended by including

$\{x \leftarrow t\}$ (*solution extension* step). In addition, t is substituted for x in the remaining equations yet to be solved. Equations of the form $t = t$ can be deleted, as solutions are not affected (*deletion* step).

If a simple equation $x = t$, where t is nonvariable and includes an occurrence of x, is generated, there is no solution to the system of equations under consideration. Similarly, an equation $f \sim(\ldots) = g(\ldots)$, in which $f \sim$, g are different function symbols, has no solution either. These two cases are the *no-solution* steps.

An equation of the form $f(u_1, \ldots, u_i) = f(v_1, \ldots, v_i)$ is replaced by the set of equations $\{u_1 = v_1, \ldots, u_i = v_i\}$ (*decomposition* step), since every solution to the original equation is also a solution to the new set of equations and vice versa.

The above transformation steps (decomposition, solution extension, deletion, and no solution) can be repeated in any order (nondeterministically) until either the no-solution condition is detected or a *solved* (*triangular*) form $\{x_1 \leftarrow w_1, \ldots, x_j \leftarrow w_j\}$ is obtained, in which for each $1 \leq i \leq j$, x_i does not appear in w_h, $h \geq i$.

The termination of these steps follows from the following observations:

- If the system of equations has no solution, then during the transformation, an unsolvable equation is generated, which would eventually be recognized by the no-solution step,
- whenever a simple equation $x = t$, where x does not occur in t, is included in a solved form, the number of variables under consideration goes down (even though the size of the problem may increase because of the substitution, unless proper data structures with bookkeeping are chosen), and
- a decomposition step reduces the problem size.

A measure that lexicographically combines the number of unsolved variables and the problem size reduces with every transformation step.

The order in which these transformation steps are performed determines the complexity of the algorithm. An algorithm of linear time complexity is discussed in Ref. 17; it is rarely used, due to the considerable overhead in implementing it. A typical implementation is of n^2 (quadratic) or $n \log n$ complexity, where n is the sum of the sizes of all the terms in the original problem. The main trick is to keep track of variables that have the same substitution; this can be done using a union-find data structure.

It can be easily shown that a set of equations either does not have a solution, or has a solution, in which case, then the most general solution (mgu) is unique up to the renaming of the (independent) variables (variables not being substituted for).

Simple unification is fundamental in automated reasoning and other areas in artificial intelligence. For example, superposition and critical-pair construction used in checking confluence, as well as in the completion procedure discussed in the section on equational inference, use a unification algorithm for identifying terms that can be simplified in either of two different ways. Resolution theorem provers as well as provers based on other approaches also use unification as the main primitive. Unification is the main computation mechanism in logic programming languages, including Prolog. Unification is also used for type inference and type checking in programming languages such as ML.

Associative–Commutative (*AC*) Unification. The above algorithm for simple unification assumed no semantics for the function symbols appearing in equations. That is why simple equations such as $x = t$, where x appears in t, as well as $f(u_1, \ldots, u_i) = g(v_1, \ldots, v_j)$, where f , g are distinct symbols, cannot be solved. If we assume some properties of function symbols and constants, then some of these equations may be solvable. For example, since for any x we have $x = x + 0$ over the integers, any substitution for x is a solution to this equation over the integers. Similarly, $x * y = x + 1$ has a solution over natural numbers: $\{ x \leftarrow 1, y \leftarrow 2 \}$, even though the top function symbols of the terms on the two sides of the equation are different.

Unification algorithms have been developed for solving equations in which some of the function symbols have specific interpretations whereas other symbols may be uninterpreted. Procedures have been proposed for

solving the general E-unification problem in which equations are solved in the presence of *interpreted symbols*, whose semantics are given by an equational theory generated by a set E of equations.

Below, we briefly review a particular but very useful case of solving equations over algebraic structures when some of the function symbols have the associative–commutative (AC) properties. The use of an associative–commutative unification/matching algorithm was key in McCune's EQP theorem prover settling the Robbins algebra conjecture (13).

Consider a finite set of equations

$$\{s_1 = t_1, \ldots, s_k = t_k\}$$

in which some of the function symbols are known to be AC. Except for the decomposition step, all transformation steps discussed above for the simple unification problem are still applicable.

For a commutative function symbol f, an equation of the form $f(u_1, u_2) = f(v_1, v_2)$ has two possible most general solutions—one in which it is attempted to make u_1, u_2 equal to v_1, v_2, respectively, and the other in which it is attempted to make u_1, u_2 equal to v_2, v_1, respectively. In general, there are many solutions possible. Each of these possibilities can lead to a different most general unifier. So in the presence of commutative function symbols, there can be exponentially many most general unifiers of a set of equations. In fact, it is easy to construct examples of equations with commutative function symbols for which the number of most general solutions is exponential in the size of the input.

If f is associative as well, then the problem gets even more interesting. As a simple example, consider

$$x + y + z = u + u + u$$

where $+$ is an AC function symbol. There are many most general solutions of the above equation. The problem is related to the partitioning problem. In one most general solution x, y, z, and u all have the same substitution, say w ; in another, the substitution for u can be $u_1 + u_2$, whereas x gets u_1, y gets $u_2 + u_2$, and z gets $u_1 + u_1 + u_2$; of course, there are many other possibilities as well.

As the reader must have observed, for an AC symbol f, its occurrences appearing as arguments to f (i.e., an argument of f has f itself as the outermost symbol) can be *flattened*. An AC f can be viewed as an n-ary function symbol instead of a binary function symbol.

Consider an equation of the form $f(u_1, \ldots, u_i) = f(v_1, \ldots, v_j)$ where f is AC and no u_i or v_j has f as its outermost symbol. This equation cannot be decomposed as before, since the order of arguments is irrelevant; also notice that i need not be equal to j . Many different decomposition may be possible.

As shown in Ref. 18, such decompositions can be done by building a *decision tree* that records all possible different choices made. For every nonvariable argument u_k, it must be determined whether u_k will be unified with (made equal to) some nonvariable argument v_l with the same outermost symbol, or will be part of a solution for some variable v_l . Such choices can be enumerated systematically with some decision paths leading to a solution, whereas others may not give any solution. Similarly, for every variable u_k, it must be determined whether it will unify with another variable and/or what its top level symbol will be in a solution. Every possible choice must be pursued for generating a *complete* set of unifiers (from which every unifier can be generated by instantiating variables in some element in the set) unless it can be determined that a particular choice will not lead to a solution.

Partial solutions are built this way until equations of the form $f(u_1, \ldots, u_i) = f(v_1, \ldots, v_j)$ in which every argument is either a variable or a constant are generated. These equations are transformed to linear diophantine equations, for which nonzero nonnegative solutions are sought (18). Since constants stand for nonvariable subterms, a solution for a variable x should not include a constant that stands for a subterm in which x occurs.

The decision tree is so constructed that a complete set of *AC* unifiers of s and t is the union of complete sets of *AC* unifiers of the unification problem corresponding to the leaf node resulting from each path. The termination of the algorithm is obvious. Further, there is considerable flexibility and the possibility of using heuristics to speed up the computation as well as to discard *a priori* paths leading to leaf nodes not resulting in any solutions.

Computing a complete basis of nonnegative solutions of simultaneous linear diophantine equations can be done in exponentially many steps. *AC* unifiers can be constructed by considering every possible subset of such a basis that assigns a nonzero value to every variable. In the worst case, double-exponentially many steps must be performed. Since there are exponentially many leaf nodes in a decision tree in the worst case, the complexity of the algorithm has an upper bound of double-exponentially many steps [i.e., there is a polynomial $p(n)$, where n is the input size, such that the number of steps is $O(2^{2^{p(n)}})$]. This also gives a double-exponential bound on the size of a complete set of *AC* unifiers. In fact, there exist simple equations (generalizations of the equation describing the partitioning problem given above) for which this bound on the number of most general *AC* unifiers, as well as the number of steps for computing them, is optimal.

To illustrate the main steps of the above algorithm, consider an equation $s = t$, where

$$s = h(*(+(x, a), +(y, a), +(z, a)), x)$$
$$t = h(*(+(w, w, w), z, z), x)$$

and $+$ and $*$ are the only *AC* function symbols. Since h is assumed to be uninterpreted, the decomposition step applies and we get the equation $*(+(x, a), +(y, a), +(z, a)) = *(+(w, w, w), z, z)$ as well as $x = x$. The second equation is trivial (i.e., it is always true no matter what substitution is made) and is discarded by the deletion step.

Consider now

$$*(+(x, a), +(y, a), +(z, a)) = *(+(w, w, w), z, z) \qquad (24)$$

A decision tree can be built based on what arguments of $*$ on both sides are made equal. One possibility is to make $+(x, a) = +(w, w, w)$. Under this assumption, the above equation simplifies to

$$*(+(y, a), +(z, a)) = *(z, z)$$

The latter equation does not have any solution, as any possible solution would have to satisfy the equation $z = +(z, a)$, which has no solution.

Similarly, making $+(y, a) = +(w, w, w)$ also does not lead to any solution.

The next possibility is to make $+(z, a) = +(w, w, w)$, which simplifies Eq. (24) to

$$*(+(x, a), +(y, a)) = *(z, z)$$

From this equation, we have $z = +(x, a) = +(y, a)$, giving a solution $\{ x \leftarrow y \}$. Substituting for z in $+(z, a) = +(w, w, w)$ gives $+(y, a, a) = +(w, w, w)$. This path thus leads to the following set of equations:

$$\{z = +(x, a),\ \ z = +(y, a),\ \ +(y, a, a) = +(w, w, w)\}$$

The equation

$$+(y, a, a) = +(w, w, w)$$

has two most general solutions:

(1) $\{ y \leftarrow w \}$, which also makes $\{ w \leftarrow a \}$, thus producing $\{ x = y = w \leftarrow a \}$
(2) $\{ w \leftarrow +(v_1, a) \}$, which also makes $\{ x = y \leftarrow +(v_1, v_1, v_1, a) \}$

For this example, there are two most general unifiers:

$$\{w \leftarrow a,\ x \leftarrow a,\ y \leftarrow a,\ z \leftarrow +(a, a)\},$$
$$\{w \leftarrow +(v_1, a),\ x \leftarrow +(v_1, v_1, v_1, a),$$
$$y \leftarrow +(v_1, v_1, v_1, a),\ z \leftarrow +(v_1, v_1, v_1, a, a)\}$$

An algorithm for computing a complete set of *AC* unifiers is discussed in detail in Ref. 18, where its complexity analysis is also given. For function symbols that, in addition to being *AC*, have properties such as identity and idempotency, unification algorithms are discussed in Ref. 19, where their complexity is also given.

Other Aspects of Equation Solving. The discussion thus far has focused on solving equations over *first-order* terms, that is, equations in which variables range over terms, and function symbols cannot be variables. However, methods have been developed for solving equations over *higher-order* terms, that is, terms that are built with two types of variables: variables that range over terms, also called *first-order* variables, and variables that range over functions and functionals, called *higher-order* variables, An allowable substitution for a first-order variable is a term, whereas a function expression (also called a λ-expression) is substituted for a higher-order variable.

For illustration, consider a very simple equation in which f and x are variables, and 0 is a constant symbol; in contrast to x, $f\sim$ is a function variable:

$$f(x) = 0$$

This equation has many most general solutions, including the following two: $f = (\lambda v.0)$ to stand for the constant function 0, together with $f\sim = (\lambda u.u)$ to stand for the identity function; and $x = 0$.

Higher-order unification and equation solving have been useful in many applications, including program synthesis, program transformation, mechanization of proofs by induction (particularly for generation of intermediate lemmas), generic and higher-order programming, and mechanization of different logics. Theorem provers such as HOL, Isabelle, and NuPRL have been designed that use higher-order unification and matching as primitive inference steps. A logic-based programming language, λ-Prolog, has been designed for facilitating some of these applications. For more details about different approaches for solving equations over higher-order terms as well as their applications, the reader may consult Ref. 20.

Narrowing is a particular approach for solving equations; a by-product of the narrowing method is that it can also be used for solving unification problems with respect to a set of equations from which a canonical rewrite system can be generated.

Assume that E is a finite set of equations, from which a finite canonical rewrite system R can be generated. A term s is said to *narrow* to a term s' with respect to R if there is a rewrite rule $l \rightarrow r$ in R, and a nonvariable

subterm s/p at position p in s, such that s/p and l unify by the most general unifier σ, and $s' = \sigma(s[p \leftarrow r])$. To check whether $s = t$ can be solved [i.e., a substitution σ for variables in s and t can be found such that $\sigma(s)$ and $\sigma(t)$ are equivalent with respect to E], both s and t are narrowed using R so that narrowing sequences from s and t converge. In that case, a substitution solving $s = t$ can be generated from the narrowing sequence.

Equation solving using narrowing is discussed in Ref. 21; see also Ref. 22, where basic narrowing is used to derive complexity results on equation solving. Narrowing can be viewed as a generalization of pseudodivision of a polynomial by a polynomial; pseudodivision is discussed in a later subsection on the characteristic-set approach for polynomial equation solving.

Polynomial Equation Solving

So far, we have discussed equational reasoning and equation solving in a general and abstract framework. In this section, we focus on equation solving in a concrete setting. All function symbols appearing in equations are interpreted; that is, the meaning of the symbols is known. This additional information is exploited in developing algorithms for solving equations.

Consider what we learn in high school about solving a system of linear equations. Functions $+$, $-$, and multiplication by a constant, as well as numbers, have the usual meaning. We learn methods for determining whether a system of linear equations has a solution or not. For solving equations, Gauss's method involves selecting a variable to *eliminate*, determining its value (in terms of other variables), eliminating the variable from the equations, and so on. With a little more effort, it is also possible to determine whether a system of equations has a single solution or infinitely many solutions. In the latter case, it is possible to study the structure of the solution space by classifying the variables into independent and dependent subsets, and specifying the solutions in terms of independent variables.

In this section, we discuss how nonlinear polynomial equations can be solved in a similar fashion, though not as easily. Below we briefly review three different approaches for symbolically solving polynomial equations—*resultants, characteristic sets*, and *Gröbner bases*. The last two approaches are related to each other, as well as to the equational inference approach based on completion discussed in the second section. The treatment of resultants is the most detailed in this section, since the material on multivariate resultants is not easily accessible. In contrast, there are books written on the Gröbner-basis and characteristic-set approaches (23 24,25).

Nonlinear polynomials are used to model many physical phenomena in engineering applications. Often there is a need to solve nonlinear polynomial equations, of if solutions do not have to be explicitly computed, it is necessary to study the nature of solutions. Many engineering problems can be easily formulated using polynomial with the help of extra variables. It then becomes necessary to eliminate some of those extra variables. Examples include implicitization of curves and surfaces, geometric reasoning, formula derivation, invariants with respect to transformation groups, robotics, and kinematics. To get an idea about the use of polynomials for modeling in different application domains, the reader may consult books by Morgan (26,27) Kapur and Mundy 28, and Donald et al. 29.

Resultant Methods. Resultant means the *result of elimination*. It is also the projection of intersection. Let us start with a simple example. Given two univariate polynomials $f(x), g(x) \in Q\ [x]$ of degrees m and n respectively, where Q; is the field of rational numbers—that is,

$$f(x) = f_n x^n + f_{n-1} x^{n-1} + \ldots f_1 x + f_0$$

and

$$g(x) = g_m x^m + g_{m-1} x^{m-1} + \ldots + g_1 x + g_0$$

—do f and g have common roots over the complex numbers, the algebraically closed extension of Q; ? Equivalently, do $\{ f(x) = 0, g(x) = 0 \}$ have a common solution? If the coefficients of f and g, instead of being rational numbers, are themselves polynomials in parameters, one is then interested in finding conditions, if any, on the parameters so that a common solution exists.

The above problem can be generalized to the elimination of many variables.

Resultant methods were developed in the eighteenth century by Newton, Euler, and Bezout, in the nineteenth century by Sylvester and Cayley, and the early parts of the twentieth century by Dixon and Macaulay. Recently, these methods have generated considerable interest because of many applications using computers. Sparse resultant methods have been discussed in Ref. 30. In Refs. 31 and 32, we have extended and generalized Dixon's construction for simultaneously eliminating many variables.

Most of the earlier work on resultants can be viewed as an attempt to extend simple linear algebra techniques developed for linear equations to nonlinear polynomial equations. A number of books on the theory of equations were written that are now out of print. Some very interesting sections were omitted by the authors in revised editions of some of these books because abstract methods in algebraic geometry and invariant theory gained dominance, and constructive methods began to be viewed as too concrete to provide any useful structural insights. A good example is the 1970 edition of van der Waerden's book on algebra (33), which does not include a beautiful chapter on elimination theory that appeared as the first chapter in the second volume of its 1940 edition. For an excellent discussion of the history of constructive methods in algebra, the reader is referred to an article by Abhyankar (34).

Resultant techniques can be broadly classified into two categories: (1) dialytic methods, in which a square system of equations is generated by multiplying polynomials by terms so that the number of equations equals the number of distinct terms appearing, and (2) differential methods, in which suitable linear combinations of polynomials are constructed, again resulting in a square system. Methods of Euler, Sylvester, Macaulay, and (more recently) sparse resultants fall in the first category, whereas methods of Bezout, Cayley, and Dixon and their generalization, along with other hybrid methods, fall into the second category.

Euler and Sylvester's Univariate Resultants. In 1764, Euler proposed a method for determining whether two univariate polynomial equations in x have a common solution. Sylvester popularized this method by giving it a matrix form, and since then, it has been widely known as Sylvester's method. Most computer algebra systems implement this method for eliminating a variable from two polynomials. It is based on the observation that for polynomials $f(x)$, $g(x)$ to have a common root, it is necessary and sufficient that there exist factors ϕ, ψ respectively of f, g such that $\deg(\phi) < n$ and $\deg(\psi) < m$ and

$$\frac{f}{\phi} = \frac{g}{\psi}$$

which is equivalent to $f * \psi - g * \phi = 0$. Since the above relation is true for arbitrary $f(x)$, $g(x)$, the coefficient of each power of x in the left side $f * \psi - g * \phi$ must be identically 0. This gives rise to $m + n$ equations in $m +$

n unknowns, which are the coefficients of terms in ϕ, ψ. This linear system gives rise to the Sylvester matrix:

$$R = \begin{pmatrix} f_0 & 0 & \dots & 0 & g_0 & 0 & \dots & 0 \\ f_1 & f_0 & \dots & 0 & g_1 & g_0 & \dots & \vdots \\ f_2 & f_1 & \dots & & g_2 & g_1 & \dots & \vdots \\ \vdots & \vdots & & & \vdots & \vdots & & \\ & & & & g_m & g_{m-1} & \dots & 0 \\ & & & & 0 & g_m & \dots & 0 \\ f_{n-1} & f_{n-2} & \dots & f_{n-m-1} & \vdots & \vdots & \dots & g_0 \\ f_n & f_{n-1} & \dots & f_{n-m} & \vdots & & & g_1 \\ 0 & f_n & \dots & f_{n-m+1} & \vdots & \vdots & \dots & g_2 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \dots & \vdots \\ & & & & \vdots & \vdots & & \vdots \\ 0 & 0 & & f_n & 0 & 0 & \dots & g_m \end{pmatrix}$$

The existence of a nonzero solution implies that the determinant of the matrix R is zero.

The Sylvester resultant can be used successively to eliminate several variables, one at a time. One soon discovers that the method suffers from an explosive growth in the degrees of the polynomials generated in the successive elimination steps. If one starts out with n or more polynomials in $Q;[x_1, x_2, \ldots, x_n]$, whose degrees are bounded by d, polynomials of degree double-exponential in n (i.e., $d^2\ n$) can get generated in the successive elimination process. The technique is impractical for eliminating more than three variables.

Macaulay's Multivariate Resultants. Macaulay generalized the resultant construction for eliminating one variable to eliminating several variables simultaneously from a system of *homogeneous* polynomial equations (35).

In a homogeneous polynomial, every term is of the same degree. A term or power product of the variables $x_1, x_2, \ldots, x_n$ is $x^{\alpha 1}{}_1\, x^{\alpha}{}_2, \ldots, x^{\alpha}{}_{\mathrm{n}}$ with $\alpha_j \geq 0$, and its degree is $\alpha_1 + \alpha_2 + \ldots + \alpha_n$, denoted by $\deg(t)$, where $t = x\ \alpha_1\ x\ \alpha_{22} \ldots x\ \alpha^{\mathrm{n}}{}_n$.

Macaulay's construction is also a generalization of the determinant of a system of homogeneous linear equations. For solving nonhomogeneous polynomial equations, the polynomials must be homogenized first. This can be easily done by introducing an extra variable; every term in the polynomial is multiplied by the appropriate power of the extra variable to make the degree of every term equal to the degree of the highest term in the polynomial being homogenized. Methods based on Macaulay's matrix give out projective zeros of the homogenized system of equations, and they can include zeros at infinity.

The key idea is to show which power products are sufficient to be used as multipliers for the polynomials so as to produce a square system of linear equations in as many unknowns as the equations. The construction below discusses that.

Let $f_1, f_2, \ldots, f_n$ be n *homogeneous* polynomials in $x_1, x_2, \ldots, x_n$. Let $d_i = \deg(f_i)$, $1 \leq i \leq n$, and $d_{\mathrm{M}} = 1 + \Sigma_1{}^{\mathrm{n}}\ (d_i - 1)$. Let T denote the set of all terms of degree d_{M} in the n variables $x_1, x_2, \ldots, x_n$:

$$T = \{x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n} | \alpha_1 + \alpha_2 + \ldots + \alpha_n = d_M\}$$

and let

$$|T| = \binom{d_M + n - 1}{n - 1}$$

The polynomial f_1 is viewed as introducing the variable x_1 ; similarly, f_2 is viewed as introducing x_2, and so on. For f_1, the power products used to multiply f_1 to generate equations are all terms of degree $d_{\mathrm{M}} - d_1$, where d_1 is the degree of f_1 . For f_2, the power products used to multiply f_2 to generate equations are all terms of degree $d_{\mathrm{M}} - d_2$ that are not multiples of $x_1\ \mathrm{d}^1$; power products that are multiples of $x_1\ \mathrm{d}_1$ are considered to be taken care of while generating equations from f_1 . That is why the polynomial f_1, for instance, is viewed as introducing the variable x_1 .) Similarly, for f_3, the power products used to multiply f_3 to generate equations are all terms of degree $d_{\mathrm{M}} - d_3$ that are not multiples of $x_1\ \mathrm{d}^1$ or $x_2\ \mathrm{d}^2$, and so on. The order in which polynomials are considered for selecting multipliers results in different systems of linear equations.

Macaulay showed that the above construction results in $|T|$ equations in which power products in x_i 's of degree d_M (these are the terms in T) are unknowns, thus resulting in a square matrix A . This matrix is quite sparse, most entries being zero; nonzero entries are coefficients of the terms in the polynomials.

Let det(A) denote the determinants of A, which is a polynomial in the coefficients of the f_i 's. It is easy to see that det(A) contains the resultant, denoted as R, as a factor. This polynomial det(A) is homogeneous in the coefficients of each f_i ; for instance, its degree in the coefficients of f_n is $d_1\ d_2\ \ldots\ d_{n-1}$.

Macaulay discussed two ways to extract a resultant from the determinants of matrices A . The resultant R can be computed by taking the gcd of all possible determinants of different matrices A that can be constructed by ordering f_i's in different ways (i.e., viewing them as introducing different variables). However, this is quite an expensive way of computing a resultant.

Macaulay also constructed the following formula relating R and det(A) :

$$R\det(B) = \det(A)$$

where det(B) is the determinant of a submatrix B of A; B is obtained from A by deleting those columns labeled by terms not divisible by any $n - 1$ of the power products $\{\ x\ 1\mathrm{d}_1, x\ 2\mathrm{d}_2, \ldots, x_{\mathrm{nd}}\ n\ \}$, and deleting those rows that contain at least one nonzero entry in the deleted columns. See Ref. 35 for more details.

u -Resultants The above construction is helpful for determining whether a system of polynomials has a common projective zero as well as for identifying a condition on parameters leading to a common projective zero. If the goal is to extract common projective zeros of a nonhomogeneous system $S = \{f_1(x_1, x_2, \ldots, x_n), \ldots, f_n(x_1, x_2, \ldots, x_n)\}$ of n polynomials in n variables, this can be done using the construction of u-resultants discussed in Ref. 33.

Homogenize the polynomials using a new homogenizing variable x_0 . Let R_u denote the Macaulay resultant of the $n + 1$ homogeneous polynomials ${}^hf_1, {}^hf_2, \ldots, {}^hf_n, {}^hf_u$ in $n + 1$ variables $x_0, x_1, \ldots, x_n$ where ${}^hf_i$ is a homogenization of f_i, f_u is the linear form

$$f_u = x_0u_0 + x_1u_1 + \cdots + x_nu_n$$

and $u_0, u_1, \ldots, u_n$ are *new unknowns*. R_u is a polynomial in $u_0, u_1, \ldots, u_n$, homogeneous in u_i 's of degree $B = \Pi^{\mathrm{n}}{}_{\mathrm{i}=1}\ d_i$. R_u is known as the u-resultant of $\mathscr{F}$. It can be shown that R_u factors into *linear factors* over $\mathbb{C}$; that

is,

$$R_u = \prod_{j=1}^{B} (u_0\alpha_{0,j} + u_1\alpha_{1,j} + \cdots + u_n\alpha_{n,j})$$

and if $u_0\alpha_{0,j} + u_1\alpha_{1,j} + \ldots + u_{n\alpha n,j}$ is a factor of R_u, then $(\alpha_{0,j}, \alpha_{1,j}, \ldots, \alpha_{n,j})$ is a common zero of ${}^hf_1, {}^hf_2, \ldots, {}^hf_n$. The converse can also be proved: if $(\beta_{0,j}, \beta_{1,j}, \ldots, \beta_{n,j})$ is a common zero of ${}^hf_1, {}^hf_2, \ldots, {}^hf_n$, then $u_0\beta_{0,j} + u_1\beta_{1,j} + \ldots + u_n\beta_{n,j}$ divides R_u. This gives an algorithm for finding all the common zeros of ${}^hf_1, {}^hf_2, \ldots, {}^hf_n$.

Recall that B is precisely the Bezout bound on the number of common zeros of n polynomials of degree $d_1, \ldots, d_n$ when zeros at infinity are included and the multiplicity of common zeros is also taken into consideration.

Nongeneric Polynomial Systems. The above methods based on Macaulay's formulation do not always work, however. For a specific polynomial system in which the coefficients of terms are specialized, the matrix A may be singular or the matrix B may be singular. If $\mathcal{F}$ has infinitely many common projective zeros, then its u-resultant R_u is identically zero, since for every zero, R_u has a factor. Even if we assume that the f_i's have only finitely many affine common zeros, that is not sufficient, since the u-resultant vanishes whenever there are infinitely many common zeros of the homogenized polynomials ${}^hf_i$'s—finitely many affine zeros, but infinitely many zeros at infinity. Often, one or both conditions arise.

Grigoriev and Chistov 36 and Canny 37 suggested a perturbation of the above algorithm that will give all the affine zeros of the original system (as long as they are finite in number) even in the presence of infinitely many zeros at infinity. Let $g_i = {}^hf_i + \lambda x^{\mathrm{d}}{}_i$ for $i = 1, \ldots, n$, and $g_u = (u_0 + \lambda)x_0 + u_1x_1 + \ldots + u_nx_n$, where λ is a new unknown. Let $R_u(\lambda, u_0, \ldots, u_n)$ be the Macaulay resultant of $g_1, g_2, \ldots, g_n$ and g_u, regarded as homogeneous polynomials in $x_0, x_1, \ldots, x_n$. The polynomial $R_u(\lambda, u_0, \ldots, u_n)$ is called the *generalized characteristic polynomial*. Suppose R_u is considered as a polynomial in λ whose coefficients are polynomials in $u_0, u_1, \ldots, u_n$:

$$R_u(\lambda, u_0, \ldots, u_n) = \lambda^{\delta} + R_{\delta-1}\lambda^{\delta-1} + \cdots + R_k\lambda^k$$

where $k \geq 0$ and the R_i's are polynomials in $u_0, u_1, \ldots, u_n$. Then the trailing coefficient R_k of R_u has all the information about the affine zeros of the original polynomial system. If $k = 0$, then R_0 is the same as the u-resultant of the original polynomial system when there are finitely many projective zeros. However, if there are infinitely many zeros at infinity, then $k > 0$. As in the case of the u-resultant, R_k can be factored, and the affine common zeros can be extracted from these factors; R_k may also include extraneous factors.

This perturbation technique is inefficient in practice because of the extra variable λ introduced. As shown in Table 1 later, the method is unable to compute a result in practice even on small examples (it typically runs out of memory). We discuss another linear algebra technique in a subsequent subsection, called the *rank submatrix* construction, for computing resultants in the context of Dixon resultant formulation. This construction can be used for singular Macaulay matrices also, as shown on a number of examples discussed in Table 1.

Sparse Resultants. As stated above, the Macaulay matrix is sparse (most entries are 0) since the size T of the matrix is larger than the number of terms in a polynomial f_i. Further, the number of affine roots of a polynomial system does not directly depend upon the degree of the polynomials, and this number decreases as certain terms are deleted from the polynomials. This observation has led to the development of *sparse elimination* theory and *sparse resultants*. The theoretical basis of this approach is the so-called *BKK bound* (30) on the number of zeros of a polynomial system in a toric variety (in which no variable can be 0); this bound depends only on the *support* of polynomials (the structure of terms appearing in them) irrespective of their degrees. The BKK bound is much tighter than the Bezout bound used in the Macaulay's formulation.

The main idea is to refine the subset of multipliers needed using *Newton polytopes*. Let $\hat{f}$ be the square matrix constructed by multiplying polynomials in $F\hat{}$ so that the number of multipliers is precisely the number

Table 1. Empirical Data

Example	Application	No. of Poly-nomials	Dixon			Macaulay			Sparse			Gröbner
			Matrix Size	Cnstr. Time	*RSC* Time	Matrix Size	*GCP* Time	*RSC* Time	Matrix Size	Cnstr. Time	*RSC* Time	Macaulay Time
1	Geom. reason.	3	13	0.18	342	55	*	4608	34	0.3	566	65836
2	Formula deriv.	5	14	0.36	76	330	*	*	86	9.8	4094	585
3	Formula deriv.	3	5	0.08	*	15	N/R	*	14	0.2	*	*
4	Geometry	3	4	0.08	1021	15	*	*	14	0.2	*	15429
5	Random	4	6	0.06	*	84	N/R	*	27	0.7	*	2207
6	Implicitization	3	10	0.1	0.58	45	3741	13.15	16	0.2	0.71	1
7	Implicitization	3	18	0.45	47	66	N/R	604	55	2	519	12530
8	Implicitization	3	18	9.33	126	153	*	94697	54	1.9	1369	70485
9	Random	4	7	0.21	220	56	*	9972	24	1.2	412	13508
10	Random	5	36	1.1	8130	1365	*	*	151	40	201558	*
11	Implicitization	3	11	0.06	3.01	36	N/R	32.58	33	0.7	31.04	57
12	Equilibrium	4	5	0.05	0.24	20	N/R	1.53	20	0.7	2.36	5
13	Vision	6	6	3.3	0.05	792	*	*	60	248	10.53	6
14	Comput. Biol.	3	8	0.067	0.27	28	11.18	0.68	16	0.2	0.33	1

of distinct power products in the resulting set of polynomials. Given a polynomial f_i, the exponents vector corresponding to every term in f_i defines a point in n -dimensional Euclidean space $\mathbb{R}^n$. The Newton polytope of f_i is the convex hull of the set of points corresponding to exponent vectors of all terms in f_i . The *Minkowski sum* of two Newton polytopes Q and S is the set of all vector sums, $q + s$, $q \in Q$, $s \in S$.

Let $Q_i \subset \mathbb{R}^n$ be the Newton polytope (wrt X) of the polynomial p_i in $\mathscr{F}$. Let $Q = Q_1 + \ldots + Q_{n+1} \subset \mathbb{R}^n$ be the Minkowski sum of Newton polytopes of all the polynomials in $\mathscr{F}$. Let $\mathscr{E}$ be the set of exponents (lattice points of $\mathbb{Z}^n$) in Q (obtained after applying a small perturbation to Q to move as many boundary lattice points as possible outside Q).

Construction of $F^{\wedge}$ is similar to the Macaulay formulation—each polynomial p_i in $\mathscr{F}$ is multiplied by certain terms to generate $F^{\wedge}$ with $|\mathscr{E}|$ equations in $|\mathscr{E}|$ unknowns (which are terms in $\mathscr{E}$), and its coefficient matrix is the *sparse resultant matrix*. Columns of the sparse resultant matrix are labeled by the terms in $\mathscr{E}$ in some order, and each row corresponds to a polynomial in $\mathscr{F}$ multiplied by a certain term. A *projection operator* (a nontrivial multiple of the resultant, as there can be extraneous factors) for $\mathscr{F}$ is simply the determinant of this matrix; see discussion on extraneous factors below.

In contrast to the size of the Macaulay matrix ($|T|$), the size of the sparse resultant matrix is $|\mathscr{E}|$, and it is typically smaller than $|T|$, especially for polynomial systems for which the BKK bound is tighter than the Bezout bound. Canny, Emiris, and others have developed algorithms to construct matrices, using greedy heuristics, that may result in smaller matrices, but in the worst case, the size can still be $|\mathscr{E}|$.

Much like Macaulay matrices, for specialization of coefficients, the sparse resultant matrix can be singular as well—even though this happens less frequently than in the case of Macaulay's formulation.

Theoretically as well as empirically, sparse resultants can be used wherever Macaulay resultants are needed (unless one is interested in projective zeros that are not affine). Further, sparse resultants are much more efficient to compute than Macaulay resultants.

So far, we have used the dialytic method for setting up the resultant matrix and computing a projection operator. To summarize, the main idea in this method is to identify enough power products that can be used as multipliers for polynomials to generate a square system with as many equations generated from the polynomials as the power products in the resulting equations.

In the next few subsections, we discuss a related but different approach for setting up the resultant matrix, based on the methods of Bezout and Cayley.

Bezout and Cayley's Method for Univariate Resultants. In 1764, around the same time as Euler, Bezout developed a method for computing the resultant that is quite different from Euler's method discussed in the previous sub-subsection. Instead of generating $m + n$ equations as in Euler's method, Bezout constructed n equations (assuming $n \geq m$) as follows:

(1) First multiply $g(x)$ by x^{n-m} to make the result a polynomial of degree n .
(2) From $f(x)$ and $g(x)x^{n-m}$, construct equations in which the degree of x is $n - 1$, by multiplying:

(1) $f(x)$ by g_m and $g(x)x^{n-m}$ by f_n and subtracting,
(2) $f(x)$ by $g_m x + g_{m-1}$, and $g(x)x^{n-m}$ by $f_n x + f_{n-1}$ and subtracting,
(3) $f(x)$ by $g_m x^2 + g_{m-1} x + g_{m-2}$, and $g(x)x^{n-m}$ by $f_n x^2 + f_{n-1} x + f_{n-2}$ and subtracting, and so on.

This construction yields m equations. An additional $n - m$ equations are obtained by multiplying $g(x)$ by $1, x, \ldots, x^{n-m-1}$, respectively. There are n equations and n unknowns—the power products, $1, x, \ldots, x^{n-1}$.

In contrast to Euler's construction, in which the coefficients of the terms in equations are the coefficients of the terms in f and g, the coefficients in this construction are sums of 2×2 determinants of the form $f_i g_j - f_i g_j$.

Cayley reformulated Bezout's method, and proposed viewing the resultant of $f(x)$ and $g(x)$ as follows: If we replace x by α in both $f(x)$ and $g(x)$, we get polynomials $f(\alpha)$ and $g(\alpha)$. The determinant

$$\Delta(x, \alpha) = \begin{vmatrix} f(x) & g(x) \\ f(\alpha) & g(\alpha) \end{vmatrix}$$

is a polynomial in x and α, and is obviously equal to zero if $x = \alpha$, meaning that $x - \alpha$ is a factor of $\Delta(x, \alpha)$. The polynomial

$$\delta(x, \alpha) = \frac{\Delta(x, \alpha)}{x - \alpha}$$

is a degree $n - 1$ polynomial in α and is symmetric in x and α . It vanishes at every common zero x_0 of $f(x)$ and $g(x)$, no matter what values α has. So, at $x = x_0$, the coefficient of every power product of α in $\delta(x, \alpha)$ is 0. This gives n equations which are polynomials in x, and the maximum degree of these polynomilas is $n - 1$. Any common zero of $f(x)$ and $g(x)$ is a solution of these polynomial equations, and they have a common solution if the determinant of their coefficients is 0. It is because of the above formulation of $\delta(x, \alpha)$ that we have called these techniques differential methods.

Extraneous Factor. There is a price to be paid in using this formulation instead of the Euler–Sylvester formulation. The result computed using Cayley's method has an additional extraneous factor of $f_n{}^{n-m}$. This factor arises because Cayley's formulation is set up assuming both polynomials are of the same degree.

Except for a system of generic polynomials, almost all multivariate elimination methods rarely compute the exact resultant. Instead, they produce a *projection operator*, which is a nontrivial nonzero multiple of the resultant. The resultant, on the other hand, is the principal generator of the ideal of the projection operators. Sometimes it is possible to predict these extraneous factors from the structure of a polynomial system, as in the case of two polynomials from which a single variable is eliminated. In general, however, it is a major challenge to determine the extraneous factors. For some results on this topic, the reader may consult Ref. 38.

Dixon's Formulation for Elimination of Two Variables. Dixon showed how to extend Cayley's formulation to three polynomials in two variables. Consider the following three generic bidegree polynomials, which have

all the power products of the type $x^i y^j$ where $0 \le i \le m$, $0 \le j \le n$:

$$f(x,y) = \sum_{i,j} a_{ij} x^i y^j, \qquad g(x,y) = \sum_{i,j} b_{ij} x^i y^j$$
$$h(x,y) = \sum_{i,j} c_{ij} x^i y^j$$

Just as in the single-variable case, Dixon 39 observed that the determinant

$$\Delta(x,y,\alpha,\beta) = \begin{vmatrix} f(x,y) & g(x,y) & h(x,y) \\ f(\alpha,y) & g(\alpha,y) & h(\alpha,y) \\ f(\alpha,\beta) & g(\alpha,\beta) & h(\alpha,\beta) \end{vmatrix}$$

vanishes when α is substituted for x or β is substituted for y, implying that $(x-\alpha)(y-\beta)$ is a factor of the above determinant. The expression

$$\delta(x,y,\alpha,\beta) = \frac{\Delta(x,y,\alpha,\beta)}{(x-\alpha)(y-\beta)}$$

is a polynomial of degree $2m-1$ in α, $n-1$ in β, $m-1$ in x, and $2n-1$ in y . Since the above determinant vanishes when we substitute $x = x_0, y = y_0$ where (x_0, y_0) is a common zero of $f(x,y), g(x,y), h(x,y)$ into the above matrix, $\delta(x, y, \alpha, \beta)$ must vanish no matter what α and β are. The coefficients of each power product $\alpha^i \beta^j$, $0 \le i \le 2m-1$, $0 \le j \le n-1$, have common zeros that include the common zeros of $f(x,y), g(x,y), h(x,y)$. This gives $2mn$ equations in power products of x, y, and the number of power products $x^i y^j$, $0 \le i \le m-1$, $0 \le j \le 2n-1$, is also $2mn$. Using a simple geometric argument, Dixon proved that the determinant is in fact the resultant up to a constant factor.

If the polynomials $f(x,y), g(x,y), h(x,y)$ are not bidegree, it can be the case that the resulting Dixon matrix is not square or, even if square, is singular.

Kapur, Saxena, and Yang's Formulation: Generalizing Dixon's Formulation. Cayley's construction for two polynomials generalizes for eliminating $n-1$ variables from a system of n nonhomogeneous polynomials $f_1, \ldots, f_n$. A matrix similar to the above can be set up by introducing new variables $\alpha_1, \ldots, \alpha_{n-1}$, and its determinant vanishes whenever $x_i = \alpha_i$, $1 \le i < n$, implying that $(x_1 - \alpha_1) \ldots (x_{n-1} - \alpha_{n-1})$ is a factor. The polynomial δ, henceforth called the *Dixon polynomial*, can be expressed directly as the determinant

$$\delta(X,\Gamma) = \begin{vmatrix} f_{1,1} & \cdots & f_{1,n} \\ f_{2,1} & \cdots & f_{2,n} \\ \vdots & & \vdots \\ f_{n-1,1} & \cdots & f_{n-1,n} \\ f_1(\alpha_1,\alpha_2,\ldots,\alpha_{n-1}) & \cdots & f_n(\alpha_1,\alpha_2,\ldots,\alpha_{n-1}) \end{vmatrix}$$

where $\Gamma = \{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\}$, where for $1 \leq j \leq n-1$ and $1 \leq i \leq n$ we have

$$f_{j,i} = \frac{f_i(\alpha_1, \ldots, \alpha_{j-1}, x_j, x_{j+1}, \ldots, x_n) - f_i(\alpha_1, \ldots, \alpha_{j-1}, \alpha_j, x_{j+1}, \ldots, x_n)}{x_j - \alpha_j}$$

and where $f_i(\alpha_1, \ldots, \alpha_k, x_{k+1}, \ldots, x_n)$ stands for uniformly replacing x_j by α_j for $1 \leq j \leq k$ in f_i.

Let

$$\begin{aligned} u &= u + 0 = u + (-(u) + -(-(u))) \\ &= (u + -(u)) + -(-(u)) = 0 + -(-(u)) \end{aligned}$$

$\mathcal{F}$ be the set of all coefficients (which are polynomials in X) of terms in δ, when viewed as a polynomial in Γ. Terms in Γ can be used to identify polynomials in

$$\begin{aligned} u &= u + 0 = u + (-(u) + -(-(u))) \\ &= (u + -(u)) + -(-(u)) = 0 + -(-(u)) \end{aligned}$$

$\mathcal{F}$ based on their coefficients. A matrix D is constructed from F in which rows are labeled by terms in Γ and columns are labeled by the terms in polynomials in

$$\begin{aligned} u &= u + 0 = u + (-(u) + -(-(u))) \\ &= (u + -(u)) + -(-(u)) = 0 + -(-(u)) \end{aligned}$$

$\mathcal{F}$. An entry $d_{i,j}$ is the coefficient of the jth term in the ith polynomial, where polynomials in

$$\begin{aligned} u &= u + 0 = u + (-(u) + -(-(u))) \\ &= (u + -(u)) + -(-(u)) = 0 + -(-(u)) \end{aligned}$$

$\mathcal{F}$ and terms appearing in the polynomials can be ordered in any manner. We have called D the *Dixon matrix*.

$n+1$ nonhomogeneous polynomials $f_0, f_1, \ldots, f_n$ in $x_1, \ldots, x_n$ are said to be *generic n-degree* if there exist nonnegative integers $m_1, \ldots, m_n$ such that each $f_j = \Sigma_{i_1}^{m_1} \mathrm{m}_1 = 0 \ldots \Sigma\, i_n \mathrm{m}_\mathrm{n} = 0\, a_{j,i1}, \ldots, \mathrm{i}_n\, x_1{}^{\mathrm{i}}{}_1 \ldots x_n\, \mathrm{i}^{\mathrm{n}}$ *for* $1 \leq j \leq n$, where $a_j, i_{i1}, \ldots, i_n$ is a distinct parameter. For generic n-degree polynomials, it can be shown that the determinant of the Dixon matrix so obtained is a resultant (up to a predetermined constant factor).

If a polynomial system is not generic n-degree, and/or the coefficients of different power products are related (specialized), then its Dixon matrix is (almost always) singular, much as in the case of Macaulay matrices. This phenomenon of resultant matrices being singular is observed in all the resultant methods when more than one variable is eliminated. In the next sub-subsection, we discuss a linear algebra construction, the *rank submatrix*, which has been found very useful for extracting projection operators from singular resultant matrices.

As shown in Table 1 below comparing different methods, this construction seems to outperform other techniques based on perturbation—for example, the generalized characteristic polynomial construction for singular Macaulay matrices discussed above. Further, empirical results suggest that the extraneous factors in projection operators computed by this method are fewer and of lower degrees.

Rank Submatrix (RSC) Construction. If a resultant matrix is singular (or rectangular), a projection operator can be extracted by computing the determinants of its largest nonsingular submatrices. This method is applicable to Macaulay matrices and sparse matrices as well as Dixon matrices. The general construction (31) is as follows:

(1) Set up the resultant matrix R of $\mathscr{F}$.
(2) Compute the rank of R and return the determinant of any *rank submatrix*, a maximal nonsingular submatrix of R. The determinant computation of a rank submatrix can be done along with the rank computation.

It was shown in Refs. 31 and 32 that

Theorem 5. If a Dixon matrix of a polynomial system f has a column that is linearly independent of the others, the determinant of any rank submatrix is a *nontrivial* multiple of the resultant of F.

The above theorem holds in general, including for nongeneric polynomial systems whose coefficients are specialized in any way. If the coefficients are assumed to be generic, then any rank submatrix is a projection operator, since one of the columns in the Dixon matrix is linearly independent of others.

Comparison of Different Resultant Methods. Interestingly, even though the Dixon formulation is classical, it does exploit the sparsity of the polynomial system, as illustrated by the following results about the size of the Dixon matrix (40).

Let $N(P)$ denote the Newton polytope of any polynomial P in F. A system F of polynomials is called *unmixed* if every polynomial in F has the same Newton polytope. Let $\pi_i(A)$ be the *projection* of an n-dimensional set A to $n - i$ dimensions obtained by substituting 0 for all the first i dimensions. Let $mvol(F)$ [which stands for $mvol(N(P), \ldots, N(P))$] be the n-fold *mixed volume* of the Newton polytopes of polynomials in F in the unmixed case, $mvol(F) = n!\ vol\ (N\,(P))$, where $vol(A)$ is the volume of an n-dimensional set A.

Theorem 6. The number of columns in the Dixon matrix of an unmixed set F of $n + 1$ polynomials in n variables is

$$O\left(vol\left(\sum_{i=1}^{n}\pi_i(N(P))\right)\right)$$

A multihomogeneous system of polynomials of type $(\ l_1, \ldots, l_r; d_1, \ldots, d_r\ )$ is defined to be an unmixed set of $\Sigma_{i=1}{}^{r}\ l_i + 1$ polynomials in $\Sigma_{i=1}{}^{r}\ l_i$ variables, where

(1) $r =$ number of partitions of variables,
(2) $l_i =$ number of variables in the ith partition, and
(3) $d_i =$ total degree of each polynomial in the variables of the ith partition.

The size of the Dixon matrix for such a system can be derived to be

$$O\left(\frac{\prod_{i=1}^{r}\left(\sum_{j=1}^{i} l_j + 1\right)^{l_i}}{n+1}\prod_{i=1}^{r}\frac{d_i{}^{l_i}}{l_i!}\right)$$

Table 2. Timings—Direct Determinant Expansion in Maple

Example	Dixon *RSC*	Macaulay *GCP*	Macaulay *RSC*	Sparce *RSC*
3	6.56	N/R	26.5	76.2
4	1.6	*	84	290
5	0.9	N/R	850	23.4

For asymptotically large number of partitions, the size of the Dixon matrix is

$$O\left(\frac{mvol(\mathcal{F})}{n+1}\right)$$

which is proportional to the mixed volume of F for a fixed n .

The size of the Dixon matrix is thus much smaller than the size of Macaulay matrix, since it depends upon the BKK bound instead of the Bezout bound. It also turns out to be much smaller than the size of the sparse resultant matrix for unmixed systems:

$$O\left(vol\left(\sum_{i=0}^{n}\mathcal{N}(P)\right)\right)$$

Since the projections are of successively lower dimension than the polytopes themselves, the Dixon matrix is smaller. Specifically, the size of the Dixon matrix of multihomogeneous polynomials is of smaller order than their n -fold mixed volume, whereas the size of the sparse resultant matrix is larger than the n -fold mixed volume by an exponential multiplicative factor.

Table 1 gives some empirical data comparing different methods on a suite of examples taken from different application domains including geometric reasoning, geometric formula derivation, implicitization problems in solid and geometric modeling, computer vision, chemical equilibrium, and computational biology, as well as some randomly generated examples. Macaulay *GCP* stands for Macaulay's method augmented with the generalized characteristic polynomial construction (perturbation method) for handling singular matrices. Macaulay *RSC*, Dixon *RSC*, and Sparse *RSC* stand for, respectively, Macaulay's method, Dixon's method, and the sparse resultant method augmented with rank submatrix computation for handling singular matrices. All timings are in seconds on a 64 Mbit Sun SPARC10. An asterisk in a time column means either that the resultant could not be computed even after running for more than a day or the program ran out of memory. N/R in the *GCP* column means there exists a polynomial ordering for which the Macaulay and the denominator matrix are nonsingular and therefore *GCP* computation is not required. Determinant computations are performed using multivariate sparse interpolation. Examples 3, 4, and 5 consist of generic polynomials with numerous parameters and dense resultants. Interpolation methods are not appropriate for such examples, and timings using straightforward determinant expansion in Maple are in Table 2. Further details are given in Ref. 41.

We included timings for computing resultant using Gröbner basis construction (discussed in a later subsection) with block ordering (variables in one block and parameters in another) using the Macaulay system. Gröbner basis computations were done in a finite field with a much smaller characteristic (31991) than in the resultant computations. Gröbner basis results produce exact resultants, in contrast to the other methods, where the results can include extraneous factors.

As can be seen from the tables, all the examples were completed using the Dixon *RSC* method. Sparse *RSC* also solved all the examples, but always took much longer than Dixon *RSC*. Macaulay *RSC* could not complete two problems and took much longer than Sparse *RSC* (for most problems) and Dixon *RSC* (for all problems).

Characteristic-Set Approach. The second approach for solving polynomial equations is to use Ritt's *characteristic-set* construction 42. This approach has been recently extended and popularized by Wu Wentsun. Despite its success in geometry theorem proving, the characteristic-set method does not seem to have gotten as much attention as it should. For many problems, characteristic-set construction is quite efficient and powerful, in contrast to both resultants and the Gröbner basis method discussed in the next section.

Given a system S of polynomials, the characteristic-set algorithm transforms S into a *triangular form* S', much in the spirit of Gauss's elimination method, with the objective that the zero set of S' (the variety of S') is "roughly equivalent" to the zero set of S . (This is precisely defined later in this subsection.) From the triangular form, the common solutions can then be extracted by solving the polynomial in the lowest variable first, back-substituting the solutions one by one, then solving the polynomial in the next variable, and so on. A total ordering on variables is assumed. Multivariate polynomials are treated as univariate polynomials in their highest variable.

Similarly to elimination steps in linear algebra, the primitive operation used in the transformation is that of *pseudodivision* of a polynomial by another polynomial. It proceeds by considering polynomials in the lowest variables. (It is also possible to devise an algorithm that considers the variables in descending order.) For each variable x_i, there may be several polynomials of different degrees with x_i as the highest variable. GCD-like computations (dividing polynomials by each other) are performed first to obtain the lowest-degree polynomial, say h_i, in x_i . If h_i is linear in x_i, then it can be used to eliminate x_i from the rest of the polynomials. Otherwise, polynomials in which the degree of x_i is higher are pseudodivided by h_i to give polynomials that are of lower degree in x_i . The smallest remainder is then used to pseudodivide other polynomials, until no remainder is generated. This process is repeated until for each variable, a minimal-degree polynomial is generated such that every polynomial generated thus far pseudodivides to 0. The set of these minimal-degree polynomials for each variable constitutes a characteristic set.

If the number of equations in S is less than the number of variables (which is typically the case when elimination or projection needs to be computed), then the variable set $\{ x_1, \ldots, x_n \}$ is typically classified into two subsets: *independent* variables (also called *parameters*) and *dependent* variables. A total ordering on the variables is chosen so that the dependent variables are all higher in the ordering than the independent variables. For elimination, the variables to be eliminated must be classified as dependent variables. The construction of a characteristic set can be employed with a goal to generate a polynomial free of variables being eliminated. We denote the independent variables by $u_1, \ldots, u_k$ and dependent variables by $y_1, \ldots, y_i$, and the total ordering is, $u_1 < \ldots < u_k < y_1 < \ldots < y_i$, where $k + l = n$.

To check whether another equation, say $f = 0$, follows from S [that is, whether f vanishes on (most of) the common zeros of S], f is pseudodivided using a characteristic set S' of S . If the remainder of the pseudodivision is 0, then $f = 0$ is said to follow from S under the condition that the *initials* (the leading coefficients) of polynomials in S' are not zero. This use of characteristic-set construction is extensively discussed in Refs. 24,43, and 44 for geometry theorem proving. Wu 45 also discusses how the characteristic set method can be used to study the zero set of polynomial equations. These uses of the characteristic-set method are discussed in detail in our introductory paper (46). In Ref. 47, a method for constructing a family of irreducible characteristic sets equivalent to a system of polynomials is discussed; unlike Wu's method discussed in a later subsection, this method does not use factorization over extension fields.

The following sub-subsections discuss the characteristic set construction in detail.

Preliminaries. Assuming an ordering $y_1 < y_2 < \ldots < y_{i-1} < y_i < \ldots < y_n$, the *highest variable* of a polynomial p is y_i if $p \notin Q;[u_1, \ldots, u_k, y_1, \ldots, y_i]$ and $p \notin Q;[u_1, \ldots, u_k, y_1, \ldots, y_{i-1}]$; that is, y_i appears in p and

every other variable in p is $< y_i$. The *class* of p is then said to be i . A polynomial p is $\geq$ another polynomial q if and only if

(1) the highest variable of p, say y_i, is $>$ the highest variable of q, say y_j (i.e., the class of p is higher than the class of q), or
(2) the highest variable of p and q is the same, say y_i, and the degree of p in y_i is $\geq$ the degree of q in y_i .

A polynomial p is *reduced with respect to* another polynomial q if (1) the highest variable y_i of p is $<$ the highest variable of q, say y_j (i.e., $p < q$), or (2) $y_i \geq y_j$ and the degree of y_j in q is $<$ the degree of y_j in p .

A list C of polynomials ($p_1, \ldots, p_m$) is called a *chain* if either (1) $m = 1$ and $p_1 \neq 0$, or (2) $m > 1$ and the class of p_1 is > 0, and for $j > i$, p_j is of higher class than p_i and reduced with respect to p_i ; we thus have $p_1 < p_2 < \ldots < pm$. [A chain is the same as an *ascending set* defined by Wu 44.] A chain is a triangular form.

Pseudodivision. Consider two multivariate polynomials p and q, viewed as polynomials in the main variable x, with coefficients that are polynomials in the other variables. Let I_q be the initial of q . The polynomial p can be pseudodivided by q if the degree of q is less than or equal to the degree of p . Let $e =$ degree $(p) -$ degree$(q) + 1$. Then

$$I_q^e p = s\, q + r$$

where s and r are polynomials, and the degree of r in x is lower than the degree of q . The polynomial r is called the *remainder* (or *pseudoremainder*) of p obtained by dividing by q . It is easy to see that the common zeros of p and q are also zeros of the remainder r, and that r is in the ideal of p and q .

For example, if $p = xy^2 + y + (x + 1)$ and $q = (x^2 + 1)y + (x + 1)$, then p cannot be divided by q but can be pseudodivided as follows:

$$(x^2 + 1)^2(xy^2 + y + x + 1) = ((x^3 + x)y + (1 - x))((x^2 + 1)y + (x + 1)) + (x^5 + x^4 + 2x^3 + 3x^2 + x)$$

with the polynomial $x^5 + x^4 + 2x^3 + 3x^2 + x$ as the pseudoremainder.

If p is not reduced with respect to q, then p reduces to r using q by *pseudodividing* p by q, giving r as the remainder of the result of pseudodivision.

Characteristic Set Algorithm.

Definition 7. Given a finite set Σ of polynomials in $u_1, \ldots, u_k, y_1, \ldots, y_l$ a characteristic set Φ of Σ is defined to be either

(1) $\{p_1\}$, where p_1 a polynomial in $u_1, \ldots, u_k$, or
(2) a chain $\langle p_1, \ldots, p_l \rangle$, where p_1 is a polynomial in $y_1, u_1, \ldots, u_k$ with initial I_1, p_2 is a polynomial in $y_2, y_1, u_1, \ldots, u_k$ with initial $I_2, \ldots, p_l$ is a polynomial in $y_l, \ldots, y_1, u_1, \ldots, u_k$ with initial I_l, such that

(1) any zero of Σ is a zero of Φ, and
(2) any zero of Φ that is not a zero of any of the initials I_i is a zero of Σ .

Then $\text{Zero}(\Sigma) \subseteq \text{Zero}(\Phi)$ as well as $\text{Zero}(\Phi/\Pi_{i=1}^{l} I_i) \subseteq \text{Zero}(\Sigma)$, where using Wu's notation, $\text{Zero}(\Phi/I)$ stands for $\text{Zero}(\Phi) - \text{Zero}(I)$. We also have

$$\text{Zero}(\Sigma) = \text{Zero}\left(\Phi / \prod_{i=1}^{l} I_i\right) \cup \bigcup_i \text{Zero}(\Sigma \cup \{I_i\})$$

Ritt 42 gave a method for computing a characteristic set from a finite basis Σ of an ideal. A characteristic set Φ is computed from Σ by successively adjoining Σ with remainder polynomials obtained by pseudodivision. Starting with $\Sigma_0 = \Sigma$, we extract a minimal chain C_i from the set Σ_i of polynomials generated so far as follows. Among the subset of polynomials with the lowest class (with the smallest highest variable, say x_j), include in C_i the polynomial of the lowest degree in x_j . This subset of polynomials is excluded for choosing other elements in C_i . Among the remaining polynomials, include in C_i a polynomial of the lowest degree in the next class (i.e., the next higher variable), and so on. So C_i is a chain consisting of the lowest-degree polynomials in each variable in Σ_i .

Compute nonzero remainders of polynomials in Σ_i with respect to the chain C_i . If this remainder set is nonempty, we adjoin it to Σ_i to obtain Σ_{i+1} . Repeat the above computation until we have Σ_j such that every polynomial in Σ_j pseudodivides to 0 with respect to its minimal chain. The set Σ_j is called a *saturation* of Σ, and that minimal chain C_j of Σ_j is a characteristic set of Σ_j as well as Σ . The above construction is guaranteed to terminate since in every step, the minimal chain of Σ_i is $>$ the minimal chain of Σ_{i+1}, and the ordering on chains is well founded.

The above algorithm can be viewed as augmenting Σ with additional polynomials from the ideal generated by Σ, much like the completion procedures discussed in the section on equational inference, until a set Δ is generated such that

(1) $\Sigma \subseteq \Delta$,
(2) Σ and Δ generate the same ideal, and
(3) a minimal chain Φ of Δ pseudodivides every polynomial in Δ to 0.

There can be many ways to compute such a Δ . For detailed descriptions of some of the algorithms, the reader may consult. Ref. 24 42,43,44,46.

Definition 8. A characteristic set $\Phi = \{p_1, \ldots, p_l\}$ is *irreducible* over $Q;[u_1, \ldots, u_k, y_1, \ldots, y_l,]$ if for $i =$ 1 to l, p_i cannot be factored over Q_{i-1} where $Q_0 = Q;(u_1, \ldots, u_k)$, the field of rational functions expressed as ratios of polynomials in $u_1, \ldots, u_k$ with rational coefficients, and $Q_j = Q_{j-1}(\alpha_j)$ is an algebraic extension of Q_{j-1}, obtained by adjoining a root α_j of $p_j = 0$ to Q_{j-1} that is $p_j(\alpha_j) = 0$ in Q_j for $1 \leq j < i$.

If a characteristic set Φ of Σ is irreducible, then $\text{Zero}(\Sigma) = \text{Zero}(\Phi)$, since the initials of Φ do not have a common zero with Φ . Ritt defined irreducible characteristic sets.

Not only can different orderings on dependent variables result in different characteristic sets, but one ordering can generate a reducible characteristic set whereas another one gives an irreducible characteristic set. For example, consider $\Sigma_1 = \{(x^2 - 2x + 1) = 0, (x - 1)z - 1 = 0\}$. Under the ordering $x < z$, Σ_1 is a characteristic set; it is however reducible, since $x^2 - 2x + 1$ can be factored. The two polynomials do not have a common zero, and under the ordering $z < x$, the characteristic set Σ_2 of Σ_1 includes only 1.

In the process of computing a characteristic set from Σ, if an element of the coefficient field (a rational number if $l = n$) is generated as a remainder, this implies that Σ does not have a solution, or is *inconsistent*. For instance, the above Σ_1 is indeed inconsistent, since a characteristic set of Σ_1 includes 1 if $z < x$ is used.

If Σ does not have a solution, either

- a characteristic set Φ of Σ includes a constant [in general, an element $Q(u_1, \ldots, u_k)$], or
- Φ is reducible, and each of the irreducible characteristic sets includes constants [respectively, elements of $Q(u_1, \ldots, u_k)$].

Theorem 9. Given a finite set Σ *of polynomials, if (i)* its characteristic set Φ is irreducible *(ii)* Φ does not include a constant, and *(iii)* the initials of the polynomials in Φ do not have a common zero with Φ then Σ has a common zero.

Ritt was apparently interested in associating characteristic sets only with *prime* ideals. A prime ideal is an ideal with the property that if an element h of the ideal can be factored as $h = h_1 h_2$, then either h_1 or h_2 must be in the ideal. A characteristic set C of a prime ideal PI is necessarily irreducible. It has the desirable property that a polynomial p pseudodivides to 0 using C if and only if p is in PI . As discussed in the next sub-subsection, in contrast, for any ideal, prime or not, a polynomial reduces by its Gröbner basis to 0 if and only if the polynomial is in the ideal.

For ideals in general, it is not the case that every polynomial in an ideal pseudodivides to 0 using its characteristic set. For example, consider the characteristic set Σ_1 above with the ordering $x < z$; even though 1 is in its ideal, 1 cannot be pseudodivided at all using Σ_1 . It is also not the case that if a polynomial pseudodivides to 0 by a characteristic set, then it is in the ideal of the characteristic set, since for pseudodivision a polynomial can be multiplied by initials.

Elimination Using the Characteristic-Set Method. For elimination (projection) of variables from a polynomial system Σ, variables being eliminated are made greater than other variables in the ordering, and a characteristic set is computed from Σ using this ordering, with the understanding that the characteristic set will include a polynomial for each eliminated variable, as well as a polynomial in the independent parameters (i.e., the other variables not being eliminated).

As soon as a polynomial in the independent parameters is generated during the construction of a characteristic set, it is a candidate for a projection operator; the conditions under which this polynomial is zero ensure a common zero of the original system Σ . As the characteristic set computation proceeds, lower-degree simpler polynomials serving as projection operators may be generated. Thus the characteristic set so computed may include a lower-degree polynomial in the parameters, with fewer extraneous factors, along with the resultant. Just as with resultant-based approaches for elimination, the characteristic-set method does not generate the exact resultant. In contrast, as we shall see in the next section, the Gröbner basis method can be used to compute the exact eliminant (resultant).

Proving Conjectures from a System of Equations. A direct way to check whether an equation $c = 0$ follows (under certain conditions) from a system f of equations is to

- compute a characteristic set $\Phi = \{p_1, \ldots, p_l\}$ from f , and
- check whether c pseudodivides to 0 with respect to Φ .

If c has a zero remainder with respect to Φ, then the equation $c = 0$ follows from Φ under the condition that none of the initials used to multiply c is 0. In this sense, $c = 0$ "almost" follows from the equations corresponding to the polynomial system S . The algebraic relation between the conjecture c and the polynomials in Φ can be expressed as

$$I_1^{i_1} \cdots I_l^{i_l}\, c = q_1 p_1 + \cdots + q_l p_l$$

where I_j is the initial of $p_j, j = 1, \ldots, l$. It is because of this relation that c is not in the ideal generated by Φ or S .

To check whether $c = 0$ exactly follows from the equations corresponding to S (i.e., the zero set of c includes the zero set of S), it must be checked whether Φ is irreducible. If so, then $c = 0$ exactly follows from the equations; otherwise, a family of irreducible characteristic sets must be generated from S, and it must be checked that for each such irreducible characteristic set, c pseudodivides to 0. This is discussed in the next sub-subsection.

The above approach is used by Wu for geometry theorem proving (43,44). A geometry problem is algebraized by translating (unordered) geometric relations into polynomial equations. A characteristic set is computed from the hypotheses of a geometry problem. For plane Euclidean geometry, most hypotheses can be formulated as linear polynomials in dependent variables, so a characteristic set can be computed efficiently. A conjecture is pseudodivided by the characteristic set to check whether the remainder is 0. If the remainder is 0, the conjecture is said to be generically valid from the hypotheses; that is, it is valid under the assumption that the initials of the polynomials in the characteristic set are nonzero. The initials correspond to the degenerate cases of the hypotheses. If the remainder is not 0, then it must be checked whether the characteristic set is reducible or not. If it is irreducible, then the conjecture can be declared to be not generically valid. Otherwise, the zero set of the hypotheses must be decomposed, and then the conjecture is checked for validity on each of the components or some of the components. This method has turned out to be quite effective in proving many geometry theorems, including many nontrivial theorems such as the butterfly theorem, Morley's theorem, and Pappus's theorem. Decomposition is necessary when case analysis must be performed to prove a theorem, for example theorems involving exterior angles and interior angles, or incircles and outcircles. An interested reader may consult Ref. 24 for many such examples.

A refutational way to check whether $c = 0$ follows from f is to compute a characteristic set of $S \cup \{cz - 1 = 0\}$, where z is a new variable. This method is used in Ref. 48 for proving plane geometry theorems.

Decomposing a Zero Set into Irreducible Zero Sets. The zero set of a polynomial system Σ can be also computed exactly using irreducible characteristic sets. The zero set of Σ can be presented as a union of zero sets of a family of irreducible characteristic sets. Below, this construction is outlined.

A reducible characteristic set can be decomposed using factorization over algebraic extensions of $Q;(u_1, \ldots, u_k)$. In the case that any of the polynomials in a characteristic set can be factored, there is a branch for each irreducible factor, as the zeros of p_j are the union of the zeros of its irreducible factors. Suppose a characteristic set $\Phi = \{p_1, \ldots, p_l\}$ from Σ is computed such that for $i > 0$, $p_1, \ldots, p_i$ cannot be factored over $Q_0, \ldots, Q_{i-1}$, respectively, but p_{i+1} can be factored over Q_i . It can be assumed that

$$gp_{i+1} = p_{i+1}^{\mathbf{1}} \cdots p_{i+1}^{j}$$

where g is in $Q;[u_1, \ldots, u_k, y_1, \ldots, y_i]$ and where $p_{i+1}{}^1 \ldots p^j{}_{i+1} \in Q$; $[u_1, \ldots, u_k, y_1, \ldots, y_i, y_{i+1}]$ and these polynomials are reduced with respect to $p_1, \ldots, p_i$. Wu 43 proved that

$$\begin{aligned}&\mathrm{Zero}(\Sigma)\\ &= \mathrm{Zero}(\Sigma^{11}) \cup \cdots \cup \mathrm{Zero}(\Sigma^{ij}) \cup \mathrm{Zero}(\Sigma^{21}) \cup \cdots \cup \mathrm{Zero}(\Sigma^{2i})\end{aligned}$$

where $\Sigma^{1h} = \Sigma \cup \{p^h{}_{i+1}\}$, $1 \le h \le j$, and $\Sigma^{2h} = \Sigma \cup \{I_h\}$, where I_h is the initial of p_h, $1 \le h \le l$.

Characteristic sets are instead computed from new polynomial sets to give a system of characteristic sets. (To make full use of the intermediate computations already performed, Σ above can be replaced by a saturation Δ of Σ used to compute the reducible characteristic set Φ.) Whenever a characteristic set includes a polynomial that can factored, this splitting is repeated. The final result of this decomposition is a system of irreducible

characteristic sets:

$$\text{Zero}(\Sigma) = \bigcup_i \text{Zero}(\Phi_i / J_i)$$

where Φ_i is an irreducible characteristic set and J_i is the product of the initials of all of the polynomials in Φ_i .

In Ref. 47, another method for decomposing the zero set of a set of polynomials into the zero set of irreducible characteristic sets is discussed. This method does not use factorization over extension fields. Instead, the method computes characteristic sets à la Ritt using Duval's D5 algorithm for inverting a polynomial with respect to an ideal. The inversion algorithm splits a polynomial in case it cannot be inverted. The result of this method is a family of characteristic sets in which the initial of every polynomial is invertible, implying that each characteristic set is irreducible.

Complexity and Implementation. Computing characteristic sets can, in general, be quite expensive. During pseudodivision, coefficients of terms can grow considerably. Techniques from subresultant and GCD computations can be used for controlling the size of the coefficients by removing some common factors *a priori*. In the worst case, the degree of a characteristic set of an ideal is bounded from both below and above by an exponential function in the number of variables. Other results on the complexity of computing characteristic sets are given in Ref. 49.

We are not aware of any commercial computer algebra system having an implementation of the characteristic-set method. We implemented the method in the *GeoMeter* system (50,51), a programming environment for geometric modeling and algebraic reasoning. Chou developed an efficient implementation of the characteristic-set algorithm with many heuristics and compact representation of polynomials (24). His implementation has been used to prove hundreds of geometry theorems. This seems to suggest that despite the worst case complexity of computing characteristic sets being double-exponential, characteristic sets can be computed efficiently for many practical applications.

Gröbner Basis Computations. In this subsection, we discuss another method for solving polynomial equations using a Gröbner basis algorithm proposed by Buchberger 52. A Gröbner basis is a special basis of a polynomial ideal with the following properties:

(1) every polynomial in the ideal simplifies to 0 using its Gröbner basis, and
(2) every polynomial has a unique normal form (canonical form) using a Gröbner basis.

A Gröbner basis of an ideal is thus a canonical (confluent and terminating) system of simplification by the polynomials in the ideal.

A Gröbner basis algorithm can be used for elimination, as well as for generating triangular forms from which common solutions of a polynomial system can be extracted. A Gröbner basis of a polynomial ideal can also be used for analyzing many structural properties of the ideal as well as its associated zero set (variety); see Refs. 23 and 25 for details.

Variables in polynomials are totally ordered. But unlike the case of characteristic set construction, polynomials are viewed as multivariate rather than as univariate polynomials in their highest variables. A polynomial is used as a simplification (rewrite) rule in which its highest monomial is used to replace the remaining part of the polynomial. Simplification of a polynomial by another polynomial is thus defined differently from pseudodivision.

The discussion below assumes the coefficient field to be Q; the approach however carries over when coefficients come from any other field. Extensions of the approach have been worked out when the coefficients are from a Euclidean domain (53).

Reduction Using a Polynomial. Recall that a term or power product of the variables $x_1, x_2, \ldots, x_n$ is $x_1\ \alpha_1\ x_2\ \alpha_2\ \ldots\ x\ \alpha_{nn}$ with $\alpha_j \geq 0$ and its degree is $\alpha_1 + \alpha_2 + \ldots + \alpha_n$, denoted by deg($t$), where $t = x^{\alpha 1}{}_1\ x\ \alpha_{22}\ \ldots\ x_n\ \alpha_n$. Assume an ordering $x_1 < x_2 < \ldots < x_n$.

Total orderings (denoted by $>$) on terms are defined as those satisfying the following properties:

(1) *Compatibility with multiplication*. If t, t_1, t_2 are terms, then $t_1 > t_2 \Rightarrow t\ t_1 > t\ t_2$.
(2) *Termination.* There can be no strictly decreasing infinite sequence of terms such as

$$t_1 > t_2 > t_3 > \cdots$$

Term orderings satisfying property 2 are called *admissible* term orderings. Two commonly used term orderings are

(1) The *lexicographic order*, $\succ$l, in which terms are ordered as in a dictionary; that is, for terms $t_1 = x\alpha_{1_1}\ x\alpha_{22}\ \ldots\ x\alpha_{n_n}$ and $t_2 = x_1\beta_1\ x^{\beta_2}{}_2\ \ldots\ x\beta^n{}_n$, $t_1 \succ$l t_2 iff $\exists i \leq n$ such that $\alpha_j = \beta_j$ for $i < j \leq n$ and $\alpha_i > \beta_i$.
(2) The *degree order*, $\succ$d, in which terms are compared first by their degrees, and equal-degree terms are compared lexicographically; that is,

$$t_1 \succ_d t_2 \text{ iff} \Big(\deg(t_1 > \deg(t_2) \text{ or } \Big(\deg(t_1 = \deg(t_2) \text{ and } t_1 \succ_l t_2 \Big) \Big)$$

Given an admissible term order $>$, for every polynomial f in Q[$x_1, x_2, \ldots, x_n$], the largest term (under $>$) in f that has a nonzero coefficient is called the *head term* of f , denoted by head (f). Let ldcf(f) denote the *leading coefficient* of f , i.e., the coefficient of head (f) in f . Every polynomial f can be written as

$$f = \text{ldcf}(f)\ \text{head}(f) + \hat{f}, \qquad \text{where} \quad \text{head}(f) \succ \text{head}(\hat{f})$$

We write tail(f) for $\hat{f}$. For example, if $f(x, y) = x^3 - y^2$, then head(f) = x^3 and tail(f) = $-y^2$ under the total degree ordering $\succ$d; under the purely lexicographic ordering with $y \succ_i x$, we have head (f) = y^2 and tail(f) = x^3 .

A polynomial f (i.e., the equation $f = 0$) is viewed as a rewrite rule

$$\text{ldcf}(f)\ \text{head}(f) \rightarrow -\text{tail}(f)$$

Let f and g be two polynomials; suppose g has a term t with a nonzero coefficient that is a multiple of head(f) ; that is,

$$g = at + \hat{g}, \qquad \text{where} \quad a \in \mathbb{Q} \qquad \text{and} \qquad t = t'\ \text{head}(f)$$

for some term t'. Then g is said to be *reducible* with respect to f, written as a *reduction by* $\rightarrow f$,

$$g \xrightarrow{f} h$$

where

$$h = g - \hat{a}t'f = -\hat{a}t' \operatorname{tail}(f) + \hat{g} \quad \text{and} \quad \hat{a} \operatorname{ldcf}(f) = a$$

The polynomial g is said to be *reducible* with respect to a set (or basis) of polynomials $F = \{f_1, f_2, \ldots, f_r\}$ if it is reducible with respect to one or more polynomials in f; otherwise, we say that g is reduced or g is a *normal form* with respect to F.

Given a polynomial g and a basis $F = \{f_1, f_2, \ldots, f_r\}$, through a *finite sequence of reductions*

$$g = g_1 \xrightarrow{F} g_2 \xrightarrow{F} g_3 \ldots \xrightarrow{F} g_s$$

such that g_s cannot be reduced further, a normal form g_s of g with respect to F can be computed. Because of the admissible ordering on terms used to choose head terms of polynomials, any sequence of reductions must terminate. Further, for every g_i in the above reduction sequence, $g_i - g \in (f_1, f_2, \ldots, f_r)$.

For example, let $F = \{f_1, f_2, f_3\}$, where

$$\begin{aligned} f_1 &= x_1^2 x_2 - 2x_2 x_3 + 1 \\ f_2 &= x_1 x_2^2 - x_3^2 + 2x_1 \\ f_3 &= x_2^2 x_3 - x_1^2 + 5 \end{aligned}$$

and

$$g = 3x_1^2 x_2^2 + x_3^1 - 1$$

Under $\succ$d, we have head $(f_1) = x^2{}_1\, x_2$, head $(f_2) = x_1\, x_2{}^2$, head $(f_3) = x_2{}^2\, x_3$, and g is reducible with respect to f. One possible reduction sequence is

$$\begin{aligned} g &= g_1 \xrightarrow{f_1} g_2 \\ &= 6x_2^2 x_3 - 3x_2 + x_1^3 - 1 \xrightarrow{f_3} g_3 = 6x_1^2 - 3x_2 + x_1^3 - 31 \end{aligned}$$

and g_3 is a normal form with respect to F. It is possible to reduce g in a another way that leads to a different normal form. For example,

$$g = g_1 \xrightarrow{f_2} g_2' = 3x_1 x_3^2 + x_1^3 - 6x_1^2 - 1$$

and the normal form g'_2 is different from g_3.

Gröbner Basis Algorithm.

Definition 10. A finite set of polynomials $G \in Q; [x_1, x_2, \ldots, x_n]$ is called a Gröbner basis for the idea (G) it generates if and only if every polynomial in $Q;x_1, x_2, \ldots, x_n]$ has a unique normal form with respect to G .

In other words, the reduction relation defined by a Gröbner basis is canonical (confluent and terminating). Buchberger (52,54) showed that every ideal in $Q; [x_1, x_2, \ldots, x_n]$ has a Gröbner basis. He also designed an algorithm to construct a Gröbner basis for any ideal I in $Q;x_1, x_2, \ldots, x_n]$ starting from an arbitrary basis for I .

Much like the superpositions and critical pairs discussed for first-order equations in an earlier section on equational inference, head terms of polynomials can be analyzed to determine whether a given basis is a Gröbner basis. For the above example, the reason for two different normal forms (g_3 and g'_2) for g with respect to f is that the monomial $3x^2{}_1\, x^2{}_2$ in g can be reduced by two different polynomials in the basis F in different ways; so head(g) was a common multiple of head(f_1) and head (f_2) .

Buchberger proposed a completion procedure to compute a Gröbner basis by augmenting the basis f by the polynomial $g_3 - g'_2$ [the augmented basis still generates the same ideal since $g_3 - g'_2 \in (f\ )$], much like the completion procedure for equations discussed in the first section. The polynomial $g_3 - g_2{}'$ corresponds to the equation between two different normal forms of a critical pair. Much like a critical pair, an s-polynomial of two polynomials f_1, f_2 is defined as follows. Let

$$m = \mathrm{lcm}(\mathrm{head}(f_1), \mathrm{head}(f_2)) = m_1 \quad \mathrm{head}(f_1) = m_2 \quad \mathrm{head}(f_2)$$

where m_1, m_2 are terms; m plays the role of a superposition. Define

$$s\text{-poly}(f_1, f_2) = m_2\, \mathrm{ldcf}(f_2)\, f_1 - m_1\, \mathrm{ldcf}(f_1)\, f_2$$

Given a basis F for an ideal I and an admissible term ordering $>$, the following algorithm returns a Gröbner basis for I for the term ordering $>$:

$$f(x, y) = x^3 - y^2, \qquad g(x, y) = 2x^2 + (y - 1)^2 - 2$$

In the above, $NF_{\mathrm{G}}(f)$ stands for *any* normal form of f with respect to the basis G . Unlike completion for equational theories, the above procedure always terminates, since by Dickson's lemma there are only finitely many noncomparable terms that can serve as the leading terms of polynomials in a Gröbner basis. For proofs of termination and correctness of the algorithm, the reader is referred to Refs. 23 and 25. The above algorithm does not use any heuristics or optimizations. Most Gröbner basis implementations use several modifications to Buchberger's algorithm in order to speed up the computations.

Examples. Consider the ideal I generated by

$$G_1 = \{y^2 + 2x^2 - 2y - 1, x^3 + 2x^2 - 2y - 1\}$$

(f defines a cusp and g defines an ellipse). Then

$$G_2 = \{g_1(x) = x^6 + 4x^5 + 4x^4 - 6x^3 - 4x^2 + 1$$
$$g_2(x, y) = y + \frac{1}{2}(-x^3 - 2x^2 + 1)\}$$

is a Gröbner basis for I under the degree ordering with $y > x$,

$$G_3 = \{h_1(y) = y^6 - 6y^5 + 17y^4 + 4y^3 - 9y^2 - 6y - 1$$
$$h_2(x, y) = x + \tfrac{1}{4}(2y^5 - 13y^4 + 40y^3 - 10y^2 - 18y - 5)\}$$

is a Gröbner basis for I under the lexicographic ordering with $y \succ x$, and

$G := F;$
$B := \{\langle f_i, f_j\rangle | f_i, f_j \in F\};$
while B is nonempty do
$h := NF_G(s - \text{poly}(f_i, f_j));$ %for some $\langle f_i, f_j\rangle \in B$
if $h \neq 0$ then
$G := G \cup \{h\};$
$B := B \cup \{\langle h, g\rangle | g \in G\};$
$fi;$
$B := B - 0\{\langle f_i, f_j\rangle\};$
od;

is a Gröbner basis for I under the lexicographic ordering with $x \sim x$.

These examples illustrate the fact that, in general, an ideal has different Gröbner bases for different term orderings. For the same term ordering, a *reduced* Gröbner bases is *unique* for an ideal; for every g in a reduced Gröbner basis G, $N\$\$_{G'}(g) = g$ where $G' = G - \{g\}$; i.e., each polynomial in G is reduced with respect to all the other polynomials in G .

Finding Common Solutions: Lexicographic Gröbner Bases. In the Gröbner basis G_2 for the above example, there is a polynomial $g_1(x)$ that depends only on x and one that depends on both x and y (in general, there can be several polynomials in a Gröbner basis that depend on x, y).

Given a reduced Gröbner basis $G = \{g_1, \ldots, g_k\}$ and an admissible term ordering $>$, G can be partitioned based on the variables appearing in the polynomials. If G includes a single polynomial in x_1, a finite set of polynomials in x_1, x_2, a finite set of polynomials in x_1, x_2, x_3, and so on, then variables are said to be *separated* in the Gröbner basis G . For example, variables are separated in Gröbner bases G_2 and G_3, whereas they are not separated in G_1 . From a basis in which variables are separated, a triangular form can be extracted by picking the least-degree polynomial for every variable in the set of polynomials introducing that variable. (It is possible that some of the variables get skipped in a separated basis.) It was observed by Trinks that such a separation of variables exists in Gröbner bases computed using lexicographic term orderings (23).

The triangular form extracted from a Gröbner basis of I in which variables are separated can be used to compute all the common zeros of I . We first find all the roots of the univariate polynomial introducing x_1 . These give the x_1-coordinates of the common zeros of the ideal I . For each such root α, we can find the common roots of $g_2(\alpha, x_2)$, the lowest-degree polynomial in x_2 that may have both x_1, x_2 ; this gives the x_2-coordinates of the corresponding common zeros of I . In this way, all the coordinates of all the common zeros can be computed. The product of the degrees of the polynomials in triangular form used to compute these coordinates also gives the total number of common zeros (including their multiplicities) of a zero-dimensional ideal I, where an ideal is zero-dimensional if and only if Zero(I) is finite.

If a variable gets skipped in a triangular form (meaning that there is no polynomial in a Gröbner basis introducing it), this implies that I is not zero-dimensional, and that I has infinitely many common zeros. As

the reader might have guessed, a Gröbner basis in which variables are separated can be used to determine the dimension of an ideal.

In principle, any system of polynomial equations can be solved using a lexicographic Gröbner basis for the ideal generated by the given polynomials. However, Gröbner bases, particularly lexicographic Gröbner bases, are hard to compute. For zero-dimensional ideals, a basis conversion has been proposed that can be used to convert a Gröbner basis computed using one admissible ordering to a Gröbner basis with respect to another admissible ordering (25). In particular, a Gröbner basis with respect to a lexicographic term ordering can be computed from a Gröbner basis with respect to a total degree ordering, which is easier to compute.

If a set of polynomials does not have a common zero (i.e., its ideal is the whole ring), then it is easy to see that a Gröbner basis of such a set of polynomials includes 1 no matter what term ordering is used. Gröbner basis computations can thus be used to check for the consistency of a system of nonlinear polynomial equations.

Theorem 11. A set of polynomials in $Q;[x_1, \ldots, x_n]$ has no common zero in C if and only if their reduced Gröbner basis with respect to any admissible term ordering is $\{1\}$.

Elimination. A Gröbner basis algorithm can also be used to eliminate variables as well as to compute the exact resultant. Variables to be eliminated are made higher than the other variables in the term ordering, just as in characteristic set computation. A Gröbner basis G of a set S of polynomials is then computed using the lexicographic term ordering from S . If there are $k + 1$ polynomials from which k variables have to be eliminated, then the smallest polynomial g in the Gröbner basis G is the exact resultant, as it can be shown that this polynomial g is the unique generator of the elimination ideal (contraction) of I in the subring of polynomials in the variables that are not being eliminated.

In Table 1 above, methods for computing multivariate resultants are contrasted with the Gröbner basis method for computing resultants on a variety of problems from different application domains. Even though the timings for the Gröbner basis approach do not compare well with resultant methods, the Gröbner basis method has an edge over the resultant methods in that it computes the resultant exactly. In contrast, resultant methods produce extraneous factors; identifying extraneous factors can require considerable effort.

The results reported in Table 1 for the Gröbner basis method were obtained using *block* ordering instead of lexicographic ordering. In a block ordering, variables are partitioned into blocks, and blocks are lexicographically ordered. Terms are compared considering variables block by block. Starting with the biggest block, the degree of a term in the variables in that block is compared against the degree of another term in these variables. Only if these degrees are the same is the next block considered, and so on. The lexicographic ordering and total degree ordering are particular cases of block orderings; each variable constitutes a block in the former, and all variables together constitute a single block in the latter. For elimination, variables being eliminated together as a block are made lexicographically bigger than the parameters (variables not being eliminated) considered together as another block. Gröbner basis computation is typically faster using a block ordering than using a lexicographic ordering, but slower than using a total degree ordering.

Theorem Proving Using Gröbner Basis Computations. A refutational approach to theorem proving has been developed exploiting the property that a Gröbner basis algorithm can be used to check whether a set of polynomial equations is inconsistent. In Ref. 55, we discussed a refutational method for geometry theorem proving. A geometry theorem proving problem (that does not involve an order relation) is formulated as the problem of checking inconsistency of a set of polynomial equations. This approach can also be used to discover missing degenerate cases as well as missing hypotheses in an incompletely stated geometry theorem. Many examples proved using Geometer, including nontrivial problems such as the butterfly theorem and Pappus's theorem, are discussed in Ref. 55.

An approach based on Gröbner basis computations has also been proposed for first-order theorem proving and implemented in our theorem prover *RRL*. For propositional calculus, formulae are translated into polynomial equations over the Boolean ring generated by propositional variables. Deciding whether a formula is a theorem is done by checking whether the corresponding polynomial equations have a solution over $\{0, 1\}$. This idea is then generalized to work on first-order rings and first-order polynomials. Details can be found in Ref. 16.

Complexity Issues and Implementation. In general, Gröbner bases are hard to compute. It was shown by Mayr and Meyer (56) that the problem of testing for ideal membership is *exponential space complete*. Their construction shows that for ideals given by bases of the form $(m_{1,1} - m_{1,2}, m_{2,1} - m_{2,2}, \ldots, m_{k,1} - m_{k,2})$ where $m_{i,j}$ is a monomial of degree at most d, Gröbner basis computation will encounter polynomials of degree as high as $O(d\ 2^{n})$, double-exponential in n, the number of variables. This is an inherent difficulty and cannot be avoided if one expects to handle all possible ideals. While double-exponential degree explosions are not observed in all problems of interest, high-degree polynomials are frequently encountered in practice.

The second problem comes from the extremely large size of the coefficients of polynomials that are generated during Gröbner basis computations. While intermediate expression swell is a common problem in computer algebra, it seems to be particularly acute in this context.

Despite these difficulties, highly nontrivial Gröbner bases computations have been performed. If the coefficients belong to a finite field (typically Z_p, where p is a word-sized prime), much larger computations are possible. Macaulay, CoCoA, and Singular are specialized computer algebra systems built for performing large computations in algebraic geometry and commutative algebra. Most general computer algebra systems (such as Maple, Macsyma, Mathematica, Reduce) provide the basic Gröbner basis functions. An implementation of the Gröbner basis algorithm also exists in GeoMeter (50,51), a programming environment for geometric modeling and algebraic reasoning. This implementation has been used for proving nontrivial plane geometry theorems.

Acknowledgment

The author would like to thank his colleagues and coauthors of articles on which most of the material in this paper is based—Lakshman Y. N., P. Narendran, T. Saxena, G. Sivakumar, M. Subramaniam, L. Yang, and H. Zhang. The author also thanks S. Lee for editorial comments.

This work was supported in part by NSF grants CCR-9622860, CCR-9712366, CCR-9712396, CCR-9996150, CCR-9996144, and CDA-9503064.

Footnotes

1. Using a sequence of positions of arguments, each subterm in a term can be uniquely identified by its position. For instance, in $(u + -(u)) + -(-(u))$, the position Λ (the empty sequence) identifies the whole term; 1 identifies the subterm $u + -(u)$, the first argument of the top level symbol $+$; 1.1 and 1.2 identify, respectively, u and $-(u)$, the first and second arguments of $+$ in $u + -(u)$; similarly, 2 identifies $-(-(u))$, the second argument of the top level symbol $+$, 2.1 identifies $-(u)$, and 2.1.1 identifies the subterm u which is the argument of $-$ in the subterm $-(u)$, the argument of $-$ in $-(-(u))$.
2. For certain equations such as $x + y = y + x$, the commutativity law, it is not even possible to transform an equation into a terminating rule, no matter which side is considered more complex. Such equations are handled semantically as discussed in a later subsection.
3. This case can be handled by one of the heuristics discussed earlier, including unfailing completion.
4. There are other ways to define rewriting modulo a set of equational axioms. For details, a reader may consult 9. The above definition matches the implementation of *AC* rewriting in our theorem prover Rewrite Rule Laboratory (*RRL*).
5. H. Robbins in 1930 conjectured that three equations defining commutativity and associativity of a binary function symbol $+$, and a third equation $-(-(x + y) + -(x + -(y))) = x$, with a unary function symbol $-$, are a basis for the variety of Boolean algebras.

BIBLIOGRAPHY

1. D. Kapur H. Zhang, An overview of Rewrite Rule Laboratory (RRL), *J. Comput. Math. Appl.*, **29** (2): 91–114, 1995.
2. D. Kapur, M. Subramaniam, Using an induction prover for verifying arithmetic circuits, *J. Softw. Tools Technol. Transfer*, 1999, to appear.
3. D. E. Knuth, P. B. Bendix, Simple word problems in universal algebras, in J. Leech (ed.), *Computational Problems in Abstract Algebras*, Oxford, England: Pergamon, 1970, pp. 263–297.
4. N. Dershowitz, Termination of rewriting, *J. Symb. Comput.*, Vol. 3, 1987, pp. 69–115.
5. L. Bachmair, *Canonical Equational Proofs*, Basel: Birkhaeuser, 1991.
6. D. Kapur, G. Sivakumar, Architecture and experiments with *RRL*, a Rewrite Rule Laboratory, in J. V. Gultag, D. Kapur, and D. R. Musser (eds.), *Proceedings of the NSF Workshop on the Rewrite Rule Laboratory*, Tech. No. GE-84GEN008 Schenectady, NY: General Electric Corporate Research and Development, 1984, pp. 33–56.
7. L. Bachmair, N. Dershowitz, D. A. Plaisted, Completion without failure, in H. Ait-Kaci and M. Nivat (eds.), *Resolution of Equations in Algebraic Structures*, Vol. 2, Boston, MA: Cambridge Press, 1989, pp. 1–30.
8. D. Kapur P. Narendran, A finite Thue system with decidable word problem and without finite equivalent canonical system, *Theor. Comput. Sci.*, **35**: 337–344, 1985.
9. J.-P. Jouannaud, H. Kirchner, Completion of a set of rules modulo a set of equations, *SIAM J. Comput.* **15**: 349–391, 1997.
10. G. E. Peterson, M. E. Stickel, Complete set of reductions for some equational theories, *J. ACM*, **28**: 223–264, 1981.
11. D. Kapur, G. Sivakumar, Proving associative–commutative termination using RPO-compatible orderings, to appear in *Invited Pap., Proc. 1st Order Theorem Proving*, 1999.
12. D. Kapur H. Zhang, A case study of the completion procedure. Proving ring commutativity problems, in J.L. Lassez and G. Plotkin (eds.), *Computational Logic: Essays in Honor of Alan Robinson*, Cambridge, MA: MIT Press, 1991, pp. 360–394.
13. W. McCune, Solution of the Robbins problem, *J. Autom. Reasoning*, **19** (3): 263–276, 1997.
14. R. S. Boyer, J. S. Moore, *A Computational Logic Handbook*, Orlando, FL: Academic Press, 1988, pp. 162–181.
15. H. Zhang, D. Kapur, M. S. Krishnamoorthy, A mechanizable induction principle for equational specifications *Proc. 9th Int. Conf. Autom. Deduction (CADE-9)*, Argonne, IL, 1988, pp. 162–181.
16. D. Kapur, P. Narendran, An equational approach to theorem proving in first-order predicate calculus, *Proc. 7th Int. Jt. Conf. Artif. Intell. (IJCAI-85)*, 1985, pp. 1146–1153.
17. M. S. Paterson M. Wegman, Linear unification, *J. Comput. Syst. Sci.*, **16**: 158–167, 1978.
18. D. Kapur P. Narendran, Complexity of associative–commutative unification check and related problems. *J. Autom. Reasoning* **9** (2): 261–288, 1992.
19. D. Kapur, P. Narendran, Double-exponential complexity of computing a complete set of AC-unifiers, *Proc. Logic Comput. Sci. (LICS)*, Santa Cruz, CA, 1992, pp. 11–21.
20. C. Prehofer, Solving higher order equations: From logic to programming, Tech. Rep. 19508, Munich, Technische Universität, 1995.
21. M. Hanus, The integration of functions into logic programming: >From theory to practice, *J. Logic Program.*, **19–20**: 583–628, 1994.
22. D. Kapur, P. Narendran, F. Otto, On ground confluence of term rewriting systems, *Inf. Comput.*, Vol. 86, San Diego, CA: Academic Press, May 1990, pp. 14–31.
23. T. Becker V. Weispfenning H. Kredel, *Gröbner Bases: A Computational Approach to Commutative Algebra*, Berlin: Springer-Verlag, 1993.
24. S.-C. Chou, *Mechanical Geometry Theorem Proving*, Dordrecht, The Netherlands: Reidel Publ., 1988.
25. D. Cox, J. Little, D. O'Shea, *Ideals, Varieties, and Algorithms*, Berlin: Springer-Verlag, 1992.
26. C. Hoffman, *Geometric and Solid Modeling: An Introduction*, San Mateo, CA: Morgan Kaufmann, 1989.
27. A. P. Morgan, *Solving Polynomial Systems Using Continuation for Scientific and Engineering Problems*, Englewood Cliffs, NJ: Prentice-Hall, 1987.
28. D. Kapur, J. L. Mundy (eds.), *Geometric Reasoning*, Cambridge, MA: MIT Press, 1989.
29. B. Donald, D. Kapur, J. L. Mundy (eds.), *Symbolic and Numeric Methods in Artificial Intelligence*, London: Academic Press, 1992.

30. I. M. Gelfand, M. M. Kapranov, A. V. Zelevinsky, *Discriminants, Resultants and Multidimensional Determinants*, Boston: Birkhaeuser, 1994.
31. D. Kapur, T. Saxena, L. Yang, Algebraic and geometric reasoning using Dixon resultants, *Proc. Int. Symp. Symb. Algebraic Comput. (ISSAC-94)*, Oxford, England, pp. 99–107, 1994.
32. D. Kapur T. Saxena, Comparison of various multivariate resultant formulations, *Proc. Int. Symp. Symb. Algebraic Comput. (ISSAC-95)*, Montreal, 1995, pp. 187–194.
33. B. L. van der Waerden, *Algebra*, Vols. 1 and 2, New York: Frederick Ungar Publ. Co., 1950, 1970.
34. S. S. Abhyankar, Historical ramblings in algebraic geometry and related algebra, *Am. Math. Mon.*, **83** (6): 409–448, 1976.
35. F. S. Macaulay, *The Algebraic Theory of Modular Systems*, Cambridge Tracts in Math. Math. Phy. Vol. 19, 1916.
36. D. Y. Grigoryev and A. L. Chistov, Sub-exponential time solving of systems of algebraic equations, LOMI Preprints E-9-83 and E-10-83, Leningrad, 1983.
37. J. Canny, Generalized characteristic polynomials, *J. Symb. Comput.*, **9**: 241–250, 1990.
38. D. Kapur, T. Saxena, Extraneous factors in the Dixon resultant formulation, *Proc. Int. Symp. Symb. Algebraic Comput. (ISSAC-97)*, Maui, HI, 1997, pp. 141–148.
39. A. L. Dixon, The eliminant of three quantics in two independent variables, *Proc. London Math. Soc.*, **6**: 468–478, 1908.
40. D. Kapur, T. Saxena, Sparsity considerations in Dixon resultants, *Proc. ACM Symp. Theory Comput. (STOC)*, Philadelphia, pp. 184–191, 1996.
41. T. Saxena, Efficient Variable Elimination Using Resultants, Ph.D. Thesis, Department of Computer Science, State University of New York, Albany, NY, 1996.
42. J. F. Ritt, *Differential Algebra*, New York: AMS Colloquium Publications, 1950.
43. W. Wu, On the decision problem and the mechanization of theorem proving in elementary geometry, in W. W. Bledsoe and D. W. Loveland (eds.), *Theorem Proving: After 25 Years, Contemporary Mathematics*, Vol. 29, Providence, RI: American Mathematical Society, 1984, pp. 213–234.
44. W. Wu, Basic principles of mechanical theorem proving in geometries, *J. Autom. Reasoning*, **2**: 221–252, 1986.
45. W. Wu, On zeros of algebraic equations—an application of Ritt's principle, *Kexue Tongbao*, **31** (1): 1–5, 1986.
46. D. Kapur, Y. N. Lakshman, Elimination methods: An introduction, in B. Donald, D. Kapur, and J. Mundy (eds.), *Symbolic and Numerical Computation for Artificial Intelligence*, San Diego, CA: Academic Press, 1992, pp. 45–89.
47. D. Kapur, *Algorithmic Elimination Methods*, Tutorial Notes for *ISSAC-95*, Montreal, 1995.
48. D. Kapur, H. Wan, Refutational proofs of geometry theorems via characteristic set computation, *Proc. Int. Symp. Symb. Algebraic Comput. (ISSAC-90)*, Japan, 1990, pp. 277–284.
49. G. Gallo, B. Mishra, *Efficient Algorithms and Bounds for Ritt–Wu Characteristic Sets*, Tech. Rep. No. 478, New York: Department of Computer Science, New York University, 1989.
50. D. Cyrluk, R. Harris, D. Kapur, GEOMETER: A theorem prover for algebraic geometry, *Proc. 9th Int. Conf. Autom. Deduction (CADE-9)*, Argonne, IL, 1988.
51. C. I. Connolly, *et al.* GeoMeter: A system for modeling and algebraic manipulation, *Proc. DARPA Workshop Image Understanding*, pp. 797–804, 1989.
52. B. Buchberger, Gröbner bases: An algorithmic method in polynomial ideal theory, in N.K. Bose (ed.), *Multidimensional Systems Theory*, Dordrecht, The Netherlands: Reidel Publ., 1985, pp. 184–232.
53. A. Kandri-Rody, D. Kapur, An algorithm for computing the Gröbner basis of a polynomial ideal over an Euclidean ring, *J. Symb. Comput.*, **6**: 37–57, 1988.
54. B. Buchberger, Applications of Gröbner bases in non-linear computational geometry, in D. Kapur and J. Mundy (eds.), *Geo-metric Reasoning*, Cambridge, MA: MIT Press, 1989, pp. 415– 447.
55. D. Kapur, A refutational approach to theorem proving in geometry, *Artif. Intell. J.*, **37** (1–3): 61–93, 1988.
56. E. Mayr, A. Meyer, The complexity of word problem for commutative semigroups and polynomial ideals, *Adv. Math.*, **46**: 305–329, 1982.

DEEPAK KAPUR
University of New Mexico