# 5

# SOME THEOREM-PROVING STRATEGIES BASED ON THE RESOLUTION PRINCIPLE

JARED L. DARLINGTON
RHEINISCH-WESTFÄLISCHES INSTITUT
FÜR INSTRUMENTELLE MATHEMATIK
BONN, GERMANY

The formulation of the resolution principle by J. A. Robinson (1965a) has provided the impetus for a number of recent efforts in automatic theorem-proving. In particular, the program PG1 (Wos *et al.* 1964, 1965), written by L. Wos, G. A. Robinson and D. F. Carson for the Control Data 3600, utilises the resolution principle in conjunction with a 'unit preference strategy' and a 'set of support strategy' to produce efficient proofs in first-order functional logic and group theory. The present author also has experimented with the resolution principle, and has incorporated it into a pair of COMIT theorem-proving programs written at the Institut für Instrumentelle Mathematik in Bonn and currently running on the Institute's IBM 7090. These programs have generated proofs of some interesting propositions of number theory, in addition to theorems of first-order functional logic and group theory.

The resolution principle, like many other theorem-proving methods, is a refutation algorithm, in the sense that it seeks to generate a contradiction from an initial set of clauses

$$C_1, C_2, \ldots, C_k$$

resulting from the negation of the theorem or formula to be tested, usually in conjunction with some already established axioms or theorems. These initial or input clauses $C_i$ are a set of logical expressions in conjunctive normal form: that is, the $C_i$ are regarded as joined by logical 'and', while an

individual clause is either a single literal $L$, in which case it is called a 'one-literal clause' or a 'unit clause', or a disjunction of $n$ literals

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

which case it is called an '$n$-literal clause' or simply an '$n$-clause'. A 'literal' is an $n$-place predicate expression

$$F(x_1, x_2, \ldots, x_n)$$

or its negation

$$\bar{F}(x_1, x_2, \ldots, x_n)$$

whose arguments are individual variables, individual constants, or functional expressions. Quantifiers do not occur in these formulae, since existentially quantified variables have been replaced by functions of universally quantified ones, and the remaining variables may therefore be taken as universally quantified. For example, the number-theoretic proposition

> 'For all $x$ and $y$, if $x$ is a divisor of $y$ then there exists some $z$ such that $x$ times $z$ equals $y$'

may be symbolised as

(i) $$\bar{D}(x, y) \vee T(x, f(x, y), y)$$

in which '$D(x, y)$' stands for '$x$ is a divisor of $y$' and '$T(x, y, z)$' stands for '$x$ times $y$ equals $z$'. The formula (i) is a 2-literal clause, and may be read as follows:

> 'Either $x$ is not a divisor of $y$, or there is a function $f(x, y)$ of $x$ and $y$ such that $x$ times $f(x, y)$ equals $y$'.

Another example is the 'first proposition of Euclid':

> 'For all $x$, $y$, and $z$, if $x$ is a prime and $x$ is a divisor of $y.z$, then $x$ is a divisor of either $y$ or $z$',

which may be symbolised as

(ii) $$\bar{P}(x) \vee \bar{T}(y, z, u) \vee \bar{D}(x, u) \vee D(x, y) \vee D(x, z).$$

Both (i) and (ii) are general formulae, which admit of infinitely many instantiations, or substitution instances, within the positive integers. Exhaustive instantiation, in fact, was the basis of a number of early theorem-proving programs. Starting with a finite number of individual constants, all possible substitutions were made in the input clauses, and the resulting conjunction of substitution instances was tested for truth-functional consistency. If a contradiction was not obtained, more constants were produced by substituting the available constants for the variables in the functional expressions $f(x, y)$, etc., that stood for the existential variables, and the additional constants were used to generate more substitution instances. This process continued until a contradiction was obtained or (the more usual case) the machine ran out of time or storage. The Herbrand theorem guaranteed that exhaustive instantiation would eventually produce a proof if one existed, given enough time and storage, but the large number of irrelevant substitution instances

generated often prevented the machine finding proofs of even relatively simple theorems. It soon became evident, therefore, to researchers in the field that the primary problem in automatic theorem-proving is that of impeding the generation of irrelevant clauses and inferences. A way in which this might be done was suggested by D. Prawitz *et al.* (1960) and elaborated upon by M. Davis (1963). The basic idea was that any substitution instance of a clause is bound to be irrelevant to a proof so long as the literals occurring in it are not 'mated', i.e., negated, by other literals occurring in other clauses. Davis in fact proved that any substitution instance

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

that contains at least one unmated literal may be erased without affecting consistency, and that the test for consistency may be confined to 'linked conjuncts', i.e., conjuncts (or clauses) wherein each literal $L_i$ is negated by a mate $\bar{L}_i$ occurring in some other clause. This 'theorem on linked conjuncts' provides a necessary, though not a sufficient, condition of relevance: any substitution instance containing an unmated literal is demonstrably irrelevant to consistency and may be deleted forthwith, but the remaining substitution instances are not necessarily all relevant. Davis showed (1963) that a theorem-proving strategy based on the search for linked conjuncts is capable of producing short proofs of many theorems, and gave as a detailed example the proposition that any left inverse in an associative group is also a right inverse.

The resolution principle takes over the idea of searching for mated literals, but applies it directly to the original clauses rather than to their substitution instances. Instead of searching for mates among the substitution instances, the resolution principle employs an algorithmic procedure to determine in advance whether two literals $L_1$ and $L_2$ in two different clauses could yield contradictory substitution instances. For example, if $L_1$ and $L_2$ are

(iii) $$T(G(x, y), x, y)$$

and

(iv) $$\bar{T}(G(u, K(v)), w, K(v)),$$

respectively, it may be determined that these two clauses will yield contradictory substitution instances, and that these (apart from the negation sign in (iv)) are all instances of the formula

(v) $$T(G(x, K(y)), x, K(y)).$$

In the terminology of J. R. Guard (1964), formula (v) may be called a 'general matching formula' ('GMF') for $L_1$ and $\bar{L}_2$, since it stands for all the common substitution instances of $L_1$ and $\bar{L}_2$. Whenever a pair of clauses $C_1$ and $C_2$ can be found that contain literals $L_1$ and $L_2$ such that $L_1$ matches $\bar{L}_2$, i.e., such that $L_1$ and $\bar{L}_2$ have common substitution instances, then a 'resolvent' of $C_1$ and $C_2$ may be generated by disjunctively joining $C_1'$ and $C_2'$, where $C_1'$ is formed by making the same substitutions throughout $C_1 - L_1$ that are necessary to make $L_1$ equal to the GMF, and $C_2'$ is formed by making the same substitutions throughout $C_2 - L_2$ that are necessary to make $\bar{L}_2$ equal

to the GMF. For example, suppose (iv) is replaced by the 2-literal clause

(vi) $$\bar{T}(G(u, K(v)), w, K(v)) \vee T(u, v, w)$$

in which the first literal is identical to (iv). Then (v) is the GMF of (iii) and the negation of the first literal in (vi), and the latter will equal the GMF (v) so long as the following substitutions are made throughout (vi):

$$u=x, v=y, w=x$$

This leaves

(vii) $$T(x, y, x)$$

as the resolvent of (iii) and (vi). Resolution is thus a functional logic analogue of the operation of 'cut' in propositional logic, whereby a single clause

$$K_2 \vee K_3 \vee \ldots \vee K_m \vee L_2 \vee L_3 \vee \ldots \vee L_n$$

can be deduced from the two separate clauses

$$K_1 \vee K_2 \vee \ldots \vee K_m$$

and

$$\bar{K}_1 \vee L_2 \vee \ldots \vee L_n.$$

If $C_1$ and $C_2$ are both unit clauses, as in (iii) and (iv), then there is nothing left over after the resolvent is formed: that is, the sole resolvent of two contradictory unit clauses is the null clause. Such a result amounts to a proof that the initial set $C$ of clauses is unsatisfiable. The resolution principle searches for a contradiction between two unit clauses, and if none can be found, it generates more resolvents until a pair of contradictory unit clauses is produced. The algorithm in its original form called for the generation of

$$S_0, S_1, S_2, \ldots, S_i, \ldots$$

where $S_0$ is the set of input clauses, and $S_i$ consists of $S_{i-1}$ plus all resolvents and 'factors' (see below) of resolvents of the elements of $S_{i-1}$. This procedure, though provably complete (as shown by Robinson in 1965a), is extremely wasteful, and is hardly more efficient than the earlier methods based on exhaustive instantiation. The reason for this inefficiency is that resolution normally produces longer and longer clauses (a resolvent of a 3-clause and a 4-clause, for example, is ordinarily a 5-clause, unless one or more of its literals are redundant), while a proof of unsatisfiability requires the generation of shorter, and ultimately unit, clauses. One way of generating shorter clauses is by means of a technique called 'factoring' (by Robinson), whereby a clause $C_i$ may be factored if it contains two matching literals $L_1$ and $L_2$. The factor is formed by making the same substitutions throughout $C_i$ that are necessary to make $L_1$ equal to $L_2$, and then deleting one of these two literals along with any other redundant literals. The following clause

(viii) $$\bar{P}(x) \vee \bar{T}(y, y, u) \vee \bar{D}(x, u) \vee D(x, y)$$

is a factor of the earlier example (ii), formed in an obvious way by substituting $y$ for $z$ throughout (ii) and then deleting the redundant literal $D(x, y)$.

A second and more generally applicable technique for generating shorter clauses is to resolve unit clauses against clauses of length $n(n \geq 1)$, thereby

producing clauses of length $n-1$ or less. It turns out in fact that many theorems can be proven by generating only resolvents that have at least one unit clause as parent ('resolvend'), and the resulting simplification of proofs is considerable.

There are several ways of writing a program that gives priority to unit clauses and the resolvents descended from them. In addition to our two COMIT programs, which we may call 'D1' and 'D2', there is the afore-mentioned program PG1 of Wos *et al.* D1 proceeds by generating successively $S_0$, $S_1$, $S_2$, etc., exactly as the original resolution algorithm, but with the restriction that each resolvent must have at least one unit clause as a parent, or (to allow for the possibility that there may initially be no unit clauses) at least one $n$-clause ($n \geqslant 1$) as a parent, where $n$ is the length of the shortest clause. D2 proceeds by resolving the unit clauses (or the $n$-clauses, as in D1) first against each other, then against the first 2-clause, the second 2-clause, etc., then against the first 3-clause, the second 3-clause, etc. Whenever two clauses $C_1$ and $C_2$ produce one or more resolvents $R_1$, $R_2$, etc., then the $R_i$ take priority over the older clauses in the sense that they are stored at the front of their appropriate lists (unit clauses, 2-clauses, etc.) and the last $R_i$ to be generated is the first to be resolved against the unit clauses. PG1 is similar to D2, the main difference being that, while D2 generates all the resolvents $R_1$, $R_2$, etc., of a pair of clauses $C_1$ and $C_2$ before proceeding further, PG1 generates only the first resolvent $R_1$ (or the $i$th resolvent $R_i$, if $R_{i-1}$ has already been generated) of $C_1$ and $C_2$ and then immediately proceeds to resolve $R_1$ (or $R_i$) against the unit clauses (or the $n$-clauses, where $n$, as in D1 and D2, is the length of the shortest clause).

The three methods described may be compared in terms of how many resolvents they generate at a time. D1 generates all the resolvents (provided that each resolvent has at least one $n$-clause as a parent) of a set $C$; D2 generates all the resolvents of a given pair $C_1$ and $C_2$ (provided that either $C_1$ or $C_2$ is an $n$-clause); and PG1 generates only one resolvent of $C_1$ and $C_2$ at a time. The difference between the three programs may be illustrated in terms of the 'Gilmore problem', i.e., the set of clauses

(1) $F(x, y)$
(2) $G(x, y) \vee \bar{F}(y, P(x, y)) \vee \bar{F}(P(x, y), P(x, y))$
(3) $\bar{F}(y, P(x, y)) \vee \bar{F}(P(x, y), P(x, y)) \vee \bar{G}(x, P(x, y)) \vee \bar{G}(P(x, y), P(x, y))$

first used as an example by P. C. Gilmore (1960). D1 generated the following proof of this example in 19 seconds (in this and subsequent examples, we shall omit parentheses within literals where no ambiguity can result):

*D1 proof of Gilmore example* (19 *seconds*)

| | |
|---|---|
| (1) $Fxy$ | premise |
| (2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$ | premise |
| (3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | premise |
| (4) $Gxy \vee \bar{F}PxyPxy$ | (1) & (2) |
| (5) $Gxy \vee \bar{F}yPxy$ | (1) & (2) |

| | |
|---|---|
| (6) $\bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (3) |
| (7) $\bar{F}yPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (3) |
| (8) $Gxy$ | (1) & (4) |
| (9) $Gxy$ | (1) & (5) |
| (10) $\bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (6) |
| (11) $\bar{G}xPxy \vee \bar{G}PxyPxy$ | (1) & (7) |
| (12) $\bar{G}PxyPxy$ | (8) & (10) |
| ..... | |
| ..... | |
| (22) contradiction | (8) & (12) |

The gap between (12) and (22) is because nine extra clauses were generated before the contradiction between (8) and (12) was discovered. D1 in fact will usually generate additional clauses after a contradictory pair has been generated, because it does not test each unit clause immediately upon generation, as does PG1, but waits until its 'generation' is complete. Furthermore, clauses (5), (7), (9) and (11) are irrelevant to the proof, and need not be shown in the final printout. Two of these, (9) and (11), are actually redundant, and may be deleted.

D2 generated the following proof, in 10 seconds:

*D2 proof of Gilmore example* (10 *seconds*)

| | |
|---|---|
| (1) $Fxy$ | premise |
| (2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$ | premise |
| (3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | premise |
| (4) $Gxy \vee \bar{F}PxyPxy$ | (1) & (2) |
| (5) $Gxy \vee \bar{F}yPxy$ | (1) & (2) |
| (6) $Gxy$ | (1) & (5) |
| (7) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}PxyPxy$ | (3) & (6) |
| (8) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy$ | (3) & (6) |
| (9) $\bar{F}yPxy \vee \bar{F}PxyPxy$ | (6) & (8) |
| (10) $\bar{F}PxyPxy$ | (1) & (9) |
| (11) $\bar{F}yPxy$ | (1) & (9) |
| (12) contradiction | (1) & (11) |

Clauses (4), (7) and (10) are superfluous, and may be omitted from the final printout. In addition, an extra $Gxy$ was generated, from (1) and (4), but was immediately deleted.

Finally, we may present the proof that PG1 would produce if it were given the Gilmore example.

*PG1 proof of Gilmore example*

| | |
|---|---|
| (1) $Fxy$ | premise |
| (2) $Gxy \vee \bar{F}yPxy \vee \bar{F}PxyPxy$ | premise |
| (3) $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}xPxy \vee \bar{G}PxyPxy$ | premise |
| (4) $Gxy \vee \bar{F}PxyPxy$ | (1) & (2) |
| (5) $Gxy$ | (1) & (4) |

| | | |
|---|---|---|
| (6) | $Gxy \vee \bar{F}yPxy$ | (1) & (2) |
| (7) | $\bar{F}yPxy \vee \bar{F}PxyPxy \vee \bar{G}PxyPxy$ | (3) & (5) |
| (8) | $\bar{F}yPxy \vee \bar{F}PxyPxy$ | (5) & (7) |
| (9) | $\bar{F}PxyPxy$ | (1) & (8) |
| (10) | contradiction | (1) & (9) |

Only two superfluous clauses are generated: (6), and an extra $Gxy$ from (1) and (6).

On the basis of the Gilmore example, it would appear that PG1 is the best, and D1 the worst, of the three programs described, with D2 falling somewhere in between but rather nearer to PG1. This judgment is confirmed by many examples, of which we may present one more: the proof of the group-theoretical proposition, which we may call 'RI', that any associative system that has left and right solutions contains a right identity element.

*First D1 proof of RI* (322 *seconds*)

| | | |
|---|---|---|
| (1) | $TGxyxy$ | existence of left solution |
| (2) | $TxHxyy$ | existence of right solution |
| (3) | $TxyFxy$ | closure |
| (4) | $\bar{T}KxxKx$ | no right identity |
| (5) | $\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}xvw \vee Tuzw$ | associativity |
| (6) | $\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}uzw \vee Txvw$ | |
| (8) | $\bar{T}xyu \vee \bar{T}yzy \vee Tuzu$ | factor of (5) |
| (20) | $\bar{T}xyz \vee TzHyyz$ | (2) & (8) |
| (33) | $\bar{T}yzKx \vee \bar{T}zxz$ | (4) & (8) |
| (73) | $TyHxxy$ | (1) & (20) |
| (77) | $\bar{T}xyx$ | (1) & (33) |
| (250) | contradiction | (73) & (77) |

*Second D1 proof of RI* (117 *seconds*)

| | | |
|---|---|---|
| (1) | $TGxyxy$ | |
| (2) | $TxHxyy$ | |
| (3) | $TxyFxy$ | |
| (4) | $\bar{T}KxxKx$ | |
| (5) | $\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}xvw \vee Tuzw$ | |
| (6) | $\bar{T}xyu \vee \bar{T}yzv \vee \bar{T}uzw \vee Txvw$ | |
| (7) | $\bar{T}yzKx \vee \bar{T}zxu \vee \bar{T}yuKx$ | (4) & (5) |
| (9) | $\bar{T}xyz \vee \bar{T}GxKyzKy$ | (1) & (7) |
| (24) | $\bar{T}xyx$ | (1) & (9) |
| (69) | contradiction | (2) & (24) |

*D2 proof of RI* (11 *seconds*)

| | |
|---|---|
| (1) | $TGxyxy$ |
| (2) | $TxHxyy$ |
| (3) | $TxyFxy$ |

| | |
|---|---|
| (4) $\overline{T}KxxKx$ | |
| (5) $\overline{T}xyu \vee \overline{T}yzv \vee \overline{T}xvw \vee Tuzw$ | |
| (6) $\overline{T}xyu \vee \overline{T}yzv \vee \overline{T}uzw \vee Txvw$ | |
| (7) $\overline{T}yzKx \vee \overline{T}zxu \vee \overline{T}yuKx$ | (4) & (5) |
| (10) $\overline{T}GxKyzKy \vee \overline{T}zyx$ | (1) & (7) |
| (11) $\overline{T}xyx$ | (1) & (10) |
| (13) contradiction | (2) & (11) |

PG1 produced a proof of RI essentially the same as that generated by D2 (though PG1 runs much faster than D1 or D2, and obtained its proof of RI in 35 milliseconds).

The latter two proofs depicted above employ an additional heuristic, the 'set of support strategy' of Wos *et al.* (1964, 1965), according to which a subset $S$ of the set $C$ of input clauses is singled out as the 'set of support' of $C$. This means that every resolvent produced must be a descendant of at least one element of $S$, i.e., no resolvents are formed entirely from elements of $C-S$. The most natural choice of $S$ is the set of clauses that formulate the special assumptions of the proof and the negation of the conclusion. In the proofs just presented, the set of support consists of the single unit clause (4), the assumption that there exists no right identity element.

Another aspect of the above proofs of RI that should be mentioned is that factoring turns out not to be essential, and is omitted from the latter two proofs, though it is included in the first proof of RI. It is often possible to obtain proofs without factoring, and it is advisable to try to do so, since the generation of factors has the tendency to produce clauses that are too specific to be of any use in a proof. There are, however, some examples in which factoring appears to be essential, such as the following, which we may call 'W5', since it is based on the fifth example of H. Wang (1964) in the appendix (Wang attributes the example to Quine).

*D1 proof of W5* (15 *seconds*)

| | |
|---|---|
| (1) $\overline{G}ya \vee \overline{G}yw \vee \overline{G}wy$ | premise |
| (2) $Gya \vee GyPy$ | premise |
| (3) $Gya \vee GPyy$ | premise |
| (4) $\overline{G}ya \vee \overline{G}ay$ | factor of (1) |
| (6) $\overline{G}ya \vee \overline{G}yy$ | factor of (1) |
| (7) $\overline{G}aa$ | factor of (6) |
| (9) $GaPa$ | (2) & (7) |
| (10) $GPaa$ | (3) & (7) |
| (11) $\overline{G}Paa$ | (4) & (9) |
| (20) contradiction | (10) & (11) |

In the above proof, factoring plays an essential role in the derivation of the key unit clause (7) in two steps from (1). It is not essential, however, in the derivation of (11), which could have been produced directly by resolution of (1) and (9), instead of first factoring (1) to obtain (4) and then resolving (4) and (9) to obtain (11).

The superiority, implicit in the foregoing examples, of D2 and PG1 over D1 is not always realised in practice, since there are some proofs for which D1's horizontal or across-the-board method of generating clauses is more suitable than the vertical or up-the-ladder procedure of D2 and PG1. One such example is the number-theoretic proof that every integer greater than one has a prime divisor, which was used as an example by S. A. Cook (1965), and which we may call 'C1'.

*D1 proof of C1 (51 seconds)*

| | |
|---|---|
| (1) $L1a$ | $a$ is greater than 1 |
| (2) $\bar{P}x \vee \bar{D}xa$ | $a$ has no prime divisor |
| (3) $\bar{L}1x \vee \bar{L}xa \vee PKx$ | assumption for induction: every $x$ greater than 1 but less than $a$ has a prime divisor |
| (4) $\bar{L}1x \vee \bar{L}xa \vee DKxx$ | |
| (5) $Dxx$ | reflexivity of 'divisor' |
| (6) $Px \vee L1Qx$ | if $x$ is not a prime, then $x$ has a divisor greater than 1 but less than $x$ |
| (7) $Px \vee LQxx$ | |
| (8) $Px \vee DQxx$ | |
| (9) $\bar{D}xy \vee \bar{D}yz \vee Dxz$ | transitivity of 'divisor' |
| (14) $\bar{P}a$ | (2) & (5) |
| (15) $L1Qa$ | (6) & (14) |
| (16) $LQaa$ | (7) & (14) |
| (17) $DQaa$ | (8) & (14) |
| (21) $\bar{L}1Qa \vee PKQa$ | (3) & (16) |
| (22) $\bar{L}1Qa \vee DKQaQa$ | (4) & (16) |
| (24) $\bar{D}xQa \vee Dxa$ | (9) & (17) |
| (28) $PKQa$ | (15) & (21) |
| (29) $DKQaQa$ | (15) & (22) |
| (35) $\bar{D}KQaa$ | (2) & (28) |
| (36) $DKQaa$ | (24) & (29) |
| (43) contradiction | (35) & (36) |

(Note that clause (1) is not essential to the proof, since it is implicit in clauses (6)-(8).)

*Incomplete D2 proof of C1 (116 seconds)*

| | |
|---|---|
| (1)-(9), as in the D1 proof of C1 | |
| (10) $\bar{P}a$ | (2) & (5) |
| (11) $L1Qa$ | (6) & (10) |
| (12) $LQaa$ | (7) & (10) |
| (13) $DQaa$ | (8) & (10) |
| (14) $\bar{P}Qa$ | (2) & (13) |
| (15) $L1QQa$ | (6) & (14) |
| (16) $LQQaQa$ | (7) & (14) |

| | |
|---|---|
| (17) $DQQaQa$ | (8) & (14) |
| (18) $\bar{D}Qaz \vee DQQaz$ | (9) & (17) |
| (19) $\bar{D}xQQa \vee DxQa$ | (9) & (17) |
| (20) $DQQaa$ | (13) & (18) |
| (21) $\bar{P}QQa$ | (2) & (20) |
| (22) $L1QQQa$ | (6) & (21) |
| (23) $LQQQaQQa$ | (7) & (21) |
| (24) $DQQQaQQa$ | (8) & (21) |
| (25) $DQQQaQa$ | (19) & (24) |
| (26) $\bar{D}Qaz \vee DQQQaz$ | (9) & (25) |
| (27) $\bar{D}xQQQa \vee DxQa$ | (9) & (25) |
| (28) $DQQQaa$ | (13) & (26) |
| (29) $\bar{P}QQQa$ | (2) & (28) |
| (30) $L1QQQQa$ | (6) & (29) |
| (31) $LQQQQaQQQa$ | (7) & (29) |
| (32) $DQQQQaQQQa$ | (8) & (29) |
| (33) $DQQQQaQa$ | (27) & (32) |
| (34) $\bar{D}Qaz \vee DQQQQaz$ | (9) & (33) |
| (35) $\bar{D}xQQQQa \vee DxQa$ | (9) & (33) |
| (36) $DQQQQaa$ | (13) & (34) |
| (37) $\bar{P}QQQQa$ | (2) & (36) |

. . . . . .

. . . . . .

Clauses (1)-(4) in the above two examples are taken as the set of support; together they formulate the assumption that there exists some $a>1$ that has no prime divisor, but every $1<x<a$ has a prime divisor. The trouble with the attempted D2 proof of C1 is that it has got itself into a loop, and is unable to form any resolvents from clauses (3) or (4). In order to obtain the proof, it is necessary to resolve (3) and (4) against (11) and (12), thereby producing $PKQa$ and $DKQaQa$, but this requires that unit clauses be resolved against 3-clauses, and there are always 2-clauses that take priority. A simpler example of looping is the tendency of the pair of clauses

$$Px,\ \bar{P}x \vee PFx$$

to produce

$$PFx,\ PFFx,\ PFFFx,\ \text{etc.}$$

*ad infinitum*. This sort of looping was recognised as a problem by the authors of PG1, who programmed around the difficulty by placing a limit, called a 'level bound', on the number of times the unit clauses can be resolved against clauses of length $m$ before going on to the $m+1$-clauses. This limit is usually set in advance of computation, at around 4 or 5. The 'level' of the input clauses $C$ is 0, the level of a factor of $C_i$ equals the level of $C_i$, and the level of a resolvent $R$ of $C_1$ and $C_2$ is greater by 1 than the maximum of the levels of $C_1$ and $C_2$. If the level bound is set at $k$, then unit clauses whose level

exceeds $k$, and non-unit clauses whose level exceeds $k-1$, are simply not generated. In the attempted D2 proof of C1, for example, none of the clauses beyond (17) would be generated if $k$ were set at 5, and the program would be forced to back up and generate the correct resolvents. The level bound strategy depends for its effectiveness upon choosing the correct value of $k$: if $k$ is too small, the program cannot generate a proof, and if $k$ is too large, the resulting irrelevant unit clauses could seriously delay the generation of a contradiction.

The procedure of D2 and PG1 of exhausting the consequences of the shorter clauses before proceeding to the longer ones creates another kind of difficulty, which becomes evident in problems like the proof of the irrationality of the square root of a prime, which proposition we may call 'SRP'.

*D1 proof of SRP (437 seconds)*

| | |
|---|---|
| (1) $Pp$ | $p$ is a prime |
| (2) $TpKbKa$ | $p.b^2=a^2$ |
| (3) $\bar{D}xa \vee \bar{D}xb \vee x=1$ | $a$ and $b$ are relatively prime |
| (4) $\neq p1$ | $p$ is not equal to 1 |
| (5) $TxxKx$ | $x.x=x^2$ |
| (6) $\bar{T}xyz \vee Dxz$ | if $x.y=z$, then $x$ is a divisor of $z$ |
| (7) $\bar{D}xy \vee TxFxyy$ | if $x$ is a divisor of $y$, then $x.Fxy=y$ |
| (8) $\bar{D}xy \vee \bar{D}yz \vee Dxz$ | transitivity of 'divisor' |
| (9) $\bar{T}xyz \vee \bar{T}zuv \vee \bar{T}xwv \vee Tuyw$ | special form of cancellation theorem: if $x.y.u=x.w$, then $u.y=w$ (consequent is commuted form of $y.u=w$) |
| (10) $\bar{P}x \vee \bar{T}yzu \vee \bar{D}xu \vee Dxy \vee Dxz$ | 'first proposition of Euclid' |
| (11) $DpKa$ | (2) & (6) |
| (14) $\bar{T}pyz \vee \bar{T}zuKa \vee TuyKb$ | (2) & (9) |
| (15) $\bar{T}yzu \vee \bar{D}pu \vee Dpy \vee Dpz$ | (1) & (10) |
| (17) $\bar{D}pa \vee \bar{D}pb$ | (3) & (4) |
| (32) $\bar{T}pya \vee TayKb$ | (5) & (14) |
| (35) $\bar{D}pKx \vee Dpx$ | (5) & (15) |
| (48) $Dpa$ | (11) & (35) |
| (57) $\bar{D}pb$ | (17) & (48) |
| (58) $TpFpaa$ | (7) & (48) |
| (60) $\bar{D}az \vee Dpz$ | (8) & (48) |
| (95) $\bar{D}pKb$ | (35) & (57) |
| (97) $TaFpaKb$ | (32) & (58) |
| (145) $\bar{D}aKb$ | (60) & (95) |
| (156) $DaKb$ | (6) & (97) |
| (250) contradiction | (145) & (156) |

In the above proof the first three clauses, which formulate the assumption that there exists a prime whose square root is rational, were taken as the set of support. Without the set of support strategy, D1 was unable to generate a proof of SRP within the storage limitations of the machine. Even with the aid of the set of support strategy, D2 was unable to generate a proof of SRP in 30 minutes, since it never got round to generating the essential clause (35), which is a consequence of (1), (5) and (10), and which states that if $p$ is a divisor of $x^2$ then $p$ is a divisor of $x$. An efficient proof requires that (35) be generated fairly early on, but D2 will not form any resolvents with (10), which is the longest clause in the set, until it has first exhausted the consequences of (1)-(9). Furthermore, when D2 starts to work on (10) it resolves it against the most recently generated unit clauses, but the generation of (35) requires that (10) be resolved against the original unit clauses. In other words, D2 (and PG1), which attacks the shortest clauses first, will not produce efficient proofs in cases (such as SRP) where efficiency demands that the longest clauses be attacked first, but D1, which attacks the short and the long clauses more or less equally, will occasionally produce moderately efficient proofs in such cases.

There remains to discuss one further type of example, that which requires the deduction of a contradiction from a set of non-unit clauses. Factoring, as in the case of W5, will sometimes suffice to generate the unit clauses essential to the operation of the programs described here. In other cases, it is necessary to produce unit clauses through resolution of non-unit clauses. This is a matter of finding or generating a pair of 2-clauses

$$C_1 \vee C_2,\ C_3 \vee C_4$$

that yield a resolvent

$$C_5 \vee C_5$$

that collapses into a unit clause, $C_5$, upon deletion of one of the redundant literals as, for example, in the deduction of

$$PFFGy$$

from

$$PFx \vee Px,\ \bar{P}FGy \vee PFFGy.$$

PG1 makes provision for this sort of inference by the inclusion of a 'non-unit section', which takes over if the 'unit section' fails to generate a proof within the stated level bound. D1 and D2 also implicitly contain non-unit sections, in that they give priority to the $n$-clauses, which are the shortest non-unit clauses in the event that there are no unit clauses, and their descendants. D1 and D2, however, have no provision for passing to a non-unit section in the event that the unit clauses and their resolvents fail to produce a proof within given limits. An example of how a non-unit section can contribute to a proof is the following, which we may call 'W3', since it is a somewhat simplified version of the third example given by H. Wang (1964) in the appendix (Wang attributes the problem to Church).

*D1 proof of W3 (271 seconds)*

| | | |
|---|---|---|
| (1) | $FPx \vee Fx$ | premise |
| (2) | $\bar{H}Px \vee Hx$ | premise |
| (3) | $\bar{H}Px \vee Gx$ | premise |
| (4) | $\bar{G}Px \vee Fx$ | premise |
| (5) | $\bar{F}Px \vee GPx \vee Hx$ | premise |
| (6) | $\bar{F}Qx \vee \bar{G}Qx \vee \bar{H}Qx$ | premise |
| (7) | $\bar{F}Px \vee HPx \vee \bar{G}x$ | premise |
| (9) | $\bar{H}PPx \vee Fx$ | (3) & (4) |
| (19) | $\bar{H}PQx \vee \bar{F}Qx \vee \bar{H}Qx$ | (3) & (6) |
| (20) | $Fx \vee \bar{F}Px \vee Hx$ | (4) & (5) |
| (78) | $Fx \vee Hx$ | (1) & (20) |
| (89) | $\bar{H}PQx \vee \bar{F}Qx$ | (2) & (19) |
| (129) | $FPx \vee Hx$ | (2) & (78) |
| (131) | $\bar{H}PQx \vee \bar{H}PPQx$ | (9) & (89) |
| (145) | $HPx \vee \bar{G}x$ | (7) & (78) |
| (213) | $\bar{H}PPQx$ | (2) & (131) |
| (257) | $Hx \vee GPx$ | (5) & (129) |
| (269) | $GPPPQx$ | (213) & (257) |
| (272) | $\bar{H}PPPQx$ | (2) & (213) |
| (273) | $HPPPPQx$ | (145) & (269) |
| (280) | $\bar{H}PPPPQx$ | (2) & (272) |
| (281) | contradiction | (273) & (280) |

Upon the generation of the unit clause (213), the value of $n$ was lowered from 2 to 1, and the 'unit section' took over, producing a series of additional unit clauses that quickly resulted in a contradiction. In order to produce the above proof in a reasonable time, it was necessary to employ a special-purpose deletion heuristic, based on the assumption that a resolvent $R$ of $C_1$ and $C_2$, that is no shorter than the longer of $C_1$ and $C_2$, is of use in a proof if and only if it assists in the generation of shorter clauses. After a reasonable time, therefore, such resolvents $R$ may be deleted. In the above proof of W3, for example, the resolvents $R$ that are no shorter than the longer of their parents are deleted after clauses shorter than $R$ are produced. This entails the deletion of clauses (19) and (20), along with a great mass of other 3-clauses, after the 2-clauses (78) and (89) are generated. This deletion heuristic was tailored specially to fit W3, and there is no guarantee that it would work in very many cases, but some such heuristic is essential if W3 is to be proven in a reasonable time, since the combinatorial possibilities between the various clauses are very great and the amount of garbage produced therefore increases at a rapid rate.

W3 is another example for which the procedure of D1 is somewhat more efficient than that of D2, since D1 resolves the 2-clauses fairly immediately against the 3-clauses, thereby generating the important 3-clauses (19) and (20) at an early stage in the proof, while D2 (and PG1) must exhaust the consequences of the 2-clauses before proceeding to resolve the 2-clauses

against the 3-clauses. D2, moreover, cannot prove W3 at all without some sort of level bound restriction, since otherwise it will get into a loop within the 2-clauses and generate

$$\bar{H}PPx \vee Fx,\ \bar{H}PPPx \vee Fx,\ \bar{H}PPPPx \vee Fx, \ldots .$$

Since the results described in this paper were obtained, several new strategies for limiting the number of resolvents generated have been proposed, most notably J. A. Robinson's '$P_1$-deduction' (1965b) and B. Meltzer's '$P_p$-deduction' (1966) and 'sharpened set of support' (Meltzer & Poggi 1966), which are all based on Robinson's '$P_1$-deduction theorem', i.e., the theorem that the null clause can always be deduced from an unsatisfiable set $C$ via a chain of resolutions $R_1$, $R_2$, etc., such that one parent of each $R_i$ is 'positive' in the sense of containing no negated literals. Meltzer has shown that the predicates of an unsatisfiable set $C$ can be renamed so that the set of positive clauses is in some sense minimal, and that this can usually be done so that all the positive clauses appear as a subset $T'$ of $T$, where $T$ is the set of clauses that formulate the special hypotheses of the proof and the negation of the conclusion. One may then use $T'$ as a set of support, in conjunction with the restriction that one parent of each resolvent must be positive. This is a 'sharpened' set of support strategy, since $T'$ is usually a proper subset of $T$, and since the restriction that one parent of each resolvent must be positive rules out some resolvents that the ordinary set of support strategy would permit. Preliminary investigation shows that $T'$ can often be reduced to just one clause, as in our earlier example C1 (minus the superfluous first clause), wherein replacing $P$ by $\bar{P}'$, $\bar{P}$ by $P'$, $D$ by $\bar{D}'$, and $\bar{D}$ by $D'$ leaves (2) as the only positive clause. Examples given by Meltzer show that the 'sharpened' set of support strategy is capable of generating short proofs for many theorems, though few data have so far been obtained on the relative efficiency of this strategy in comparison with the ordinary set of support strategy with unit preference. The former strategy does rule out some inferences involving unit clauses that the latter strategy permits, e.g.,

$$\bar{P}x,\ Px \vee \bar{Q}x,\ \therefore \bar{Q}x$$

and the 'non-unit section' of one's program will therefore become relatively more important under the sharpened set of support strategy.

We have so far said nothing about the completeness of the methods discussed. D1 and D2 were not designed to be complete theorem-proving algorithms, but were intended rather as methods of producing efficient proofs for certain types of theorems. In order for them to be complete algorithms, it would be necessary to include an explicit 'non-unit section', as in PG1, general enough to provide for all cases in which the proof depends, in whole or in part, upon the generation of resolvents from pairs of non-unit clauses. The authors of PG1 have shown that their method is complete, so long as the level bound $k$ is equal to or greater than the number of generations

$$S_0, S_1, S_2, \ldots, S_i, \ldots$$

that the original resolution algorithm would have to produce in order to obtain a proof, and so long as the set of support $S$ is chosen in such a way that $C-S$ is satisfiable (i.e., such that the contradiction depends upon adding $S$ to an otherwise consistent set of clauses). $P_1$-deduction and its variants are also provably complete. There is no necessary connection, however, between efficiency and completeness, and if and when really significant mathematical theorems are proven mechanically, the proofs will probably be generated by special-purpose heuristics rather than by theoretically complete general methods.

## REFERENCES


Cook, S. A. (1965). Algebraic techniques and the mechanization of number theory, *RM*-4319-*PR*, RAND Corporation, Santa Monica, California.

Davis, M. (1963). Eliminating the irrelevant from mechanical proofs, *Proceedings of Symposia in Applied Mathematics* XV, pp. 15-30. American Mathematical Society, Providence, Rhode Island.

Gilmore, P. C. (1960). A proof method for quantification theory, *IBM J. Res. Devel.*, **4**.

Guard, J. R. (1964). Automated logic for semi-automated mathematics, *Contract No. AF*19(628)-3250, *Project No.* 5632, *Task No.* 563205, *Scientific Report No.* **1**, Applied Logic Corporation, Princeton, New Jersey.

Meltzer, B. (1966). Theorem-proving for computers: some results on resolution and renaming. *Comput. J.*, **8**, 341-343.

Meltzer, B., & Poggi, P. (1966). An improved complete strategy for theorem-proving by resolution, Internal report, Metamathematics Unit, University of Edinburgh.

Prawitz, D., Prawitz, H., & Voghera, N. (1960). A mechanical proof procedure and its realization in an electronic computer. *J. Assn Comput. Mach.*, **7**, 102-128.

Robinson, J. A. (1965a). A machine-oriented logic based on the resolution principle, *J. Assn Comput. Mach.*, **12**, 23-41.

Robinson, J. A. (1965b). Automatic deduction with hyper-resolution. *Int. J. Computer Math.*, **1**, 227-234.

Wang, H. (1964). Toward mechanical mathematics, *IBM J. Res. Dev.*, **4**. Also appears as Chapter IX in *A Survey of Mathematical Logic.* Peking: Science Press, and Amsterdam: North-Holland Publishing Company, 1964.

Wos, L., Carson, D., & Robinson, G. (1964). The unit preference strategy in theorem proving, *AFIPS Conference Proceedings*, **26**, 615-621. Washington, D.C.: Spartan Books.

Wos, L., Robinson, G. A., & Carson, D. F. (1965). Efficiency and completeness of the set of support strategy in theorem proving, *J. Assn Comput. Mach.*, **12**, 536-541.