TREE MATCHING AND SIMPLIFICATION

by

Paul W. Purdom, Jr.
Indiana University
Bloomington, IN 47405

and

Cynthia A. Brown
Northeastern University
Boston, Massachusetts 02115

TECHNICAL REPORT NO. 181

TREE MATCHING AND SIMPLIFICATION

by

Paul W. Purdom, Jr., Indiana University
and Cynthia A. Brown, Northeastern University

Revised: April, 1986

This material is based on work supported by the National Science Foundation under grant number MCS 83-01742.

Tree Matching and Simplification

Abstract

A fast algorithm for performing simplification and matching is described. The algorithm gives an improvement of up to an order of magnitude on suitable problems. It makes use of a dag data structure and tag fields to avoid redundant matches. Performance studies were done to determine the relative importance of various features in improving the running time. The actual code is given in an appendix.

1. Introduction

In this paper we describe a fast algorithm that can improve the performance of a program that does simplification and matching by a factor of two or more. While this algorithm was developed as part of a program for carrying out a specialized task (the Knuth-Bendix completion procedure [10]), the basic ideas are quite general.

The problem solved by the algorithm is as follows. A set of simplification rules is given. The left and right side of each rule is a term involving a specified set of operators, variables, and constants, where a variable stands for an arbitrary subterm. For example, the following set of rules states the basic group theory axioms (where variables begin with upper-case letters):

$$\begin{aligned} &\times(\times(A,B),C) \rightarrow \times(A,\times(B,C)) \\ &\times(i(A),A) \rightarrow e \\ &\times(e,A) \rightarrow A. \end{aligned}$$

The left sides of the rules form the set of *patterns*; the patterns are matched against the subexpressions of a *subject* expression, which is a term of the same type. Variables in the patterns can match subterms in the subject, provided that variables that occur more than once match the same expression each time. For example, in the expression $\times(e,\times(X,Y))$, the entire expression is matched by the left side of the third group theory rule, with the variable A matching the subexpression $\times(X,Y)$.

Any matched subexpression is replaced by the corresponding right side of the matching rule, with the values from the left side match substituted for the variables. In our example, the expression $\times(e,\times(X,Y))$ is replaced by $\times(X,Y)$. Such a replacement is regarded as a *simplification* of the original subject. Matching and substitution continue until no further simplifications can be done. In this application, the terms in both the patterns and the subject may be thought of as trees, using the usual mapping of expressions to trees. We are thus interested in many-to-one pattern matching and simplification in trees.

This problem arises in many contexts. Simplifying an expression according to a set of rules is done as part of many theorem-proving methods, in symbolic algebra systems, and in some kinds of pattern-based code generation methods. The algorithm described in this paper was developed as part of a Knuth-Bendix completion program. The goal of the program is to produce a set of rules that have certain desirable properties. First, the set must be *Noetherian*: every sequence of simplifications done with the rules must terminate. Second, the set must be *confluent*: the fully simplified term obtained from a given term must be unique. Rules in a confluent set can be applied in any order; the ultimate result will be the same. If a set of rules is both confluent and Noetherian, then each term has a unique normal form, and any complete simplification reduces the term to its normal form. The program starts with an initial set of rules and attempts to discover new rules and add them to the set, until a confluent and Noetherian set (a *complete* set) is obtained. The program cannot complete every rule set (if it could, it could solve problems that are known to be undecidable, such as Hilbert's tenth problem [2]), but it works in a large enough number of cases to be interesting.

The matching and simplification routines we present do not provide for any control over the order in which rules are applied. This is a sensible way to proceed with a complete set of rules. The routines allow for new rules to be added to the rule set after computation has begun, and they provide for keeping all rules in simplest form with respect to each other. The same routines work with any set of tree rewrite rules, but an infinite loop may occur if the rules are not Noetherian.

Our routines make use of a dag data structure to represent the terms in the system. All the terms, including the subject and the left and right sides of the patterns, are contained in a single dag. (This and related data structures for fast matching are discussed by O'Donnell [13]. Also see the references in [13].) The simplification algorithm uses two major improvements over a naive approach: it keeps track of terms that are known to be in simplest form (so that no attempt is made to resimplify them) and it remembers simplifications it has done on terms that are not in simplest form. The common dag is the key to these improvements; if a new term is equal to one that is already in the dag and that has previously been simplified, then no further work needs to be done on it.

Our simplification algorithm has many similarities with the congruence closure technique [1, 12, 4]. In particular, it uses a dag to avoid rederiving results. Unlike the congruence closure techniques, it can handle simplification rules that contain variables.

The matching algorithm is relatively simple. It also has two major features. The data structure records when an expression contains no variables; two expressions with no variables can match only if they are identical. Second, the matching algorithm takes advantage of the dag data structure to speed up matching of subjects with repeated subexpressions.

With these four features, the methods we present were able to speed up the simplification portion of our Knuth-Bendix program by factors ranging from 2.5 to 12.8, with the improvement factor increasing with problem size. This resulted in an overall improvement in the running time of the entire program by a factor of 1.4 to 4.1. A similarly significant improvement could be expected for any program that makes heavy use of simplification and matching.

The next sections of this paper provide a more detailed description of the data structures and algorithms used in our program. This is followed by an analysis of the relative contributions of each of the four features mentioned above to the overall reduction in running time. Appendix 1 gives the actual code for the matching and simplification routines.

2. Simplification

Finding a way to simplify a term requires two steps: finding a match between the left side

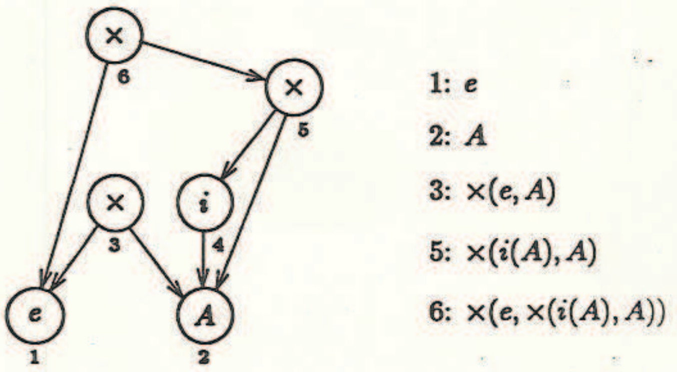

Fig. 1. The dag for the terms $\times(e, \times(i(A), A))$, $\times(i(A), A)$, e, $\times(e, A)$, and A.

of a rule and a subexpression of the term, and substituting the right side of the rule for the matched subexpression. The first part of this task is by far the more costly, and offers the greatest scope for improving the speed of simplification. This can be done in two ways: by speeding up the matching algorithm, and by reducing the number of times it needs to be used. The matching algorithm is the subject of the next section; in this section, we focus on making the simplifier clever enough to avoid unnecessary matches.

Our algorithms are based on a dag data structure. All terms are represented by nodes in a single dag with no repeated nodes. If two terms have the same main operators and the same ordered set of children, then they are represented by the *same* node. For example, if we have the subject $\times(e, \times(i(A), A))$ and the rules $\times(i(A), A) \rightarrow e$ and $\times(e, A) \rightarrow A$, then the terms will be stored in the dag shown in Fig. 1.

To speed up simplification, two pieces of information are kept at each node of the dag. The *canonical* field is *true* if the node is in its simplest form; that is, if none of the rules in the present rule set can simplify the node. If there is a way to simplify the node, then the *canonical* field is set to *false*. If a simplification is known for a node, then the *simplifies_to* field points to the simplified version of it (also in the dag). Otherwise, the *simplifies_to* field is **nil**. These fields, together with the unified dag data structure, eliminate a lot of redundant matching. If a new term has any subterms in common with others in the system, they will not have to be reexamined by the simplification routine.

Initially, all nodes except variables have their *canonical* fields set to *false* and their *simplifies_to* fields set to **nil**. (A variable can never be simplified, and so may be marked canonical *a priori*.) There are several routines that can change the values in these fields. Obviously, the simplification routine will set the fields according to whether or not it finds a simplification. The routine that adds a new rule to the system also resets all *canonical* fields to *false* (except those for variables), although it leaves the *simplifies_to* field alone. The Knuth-Bendix program has a routine that copies a term and renames all the variables, giving them a standard set of names. If this routine discovers a non-**nil** *simplifies_to* field, or a *true canonical* field, in either the term being copied or in the new term (which may already be in the dag), then it sets the corresponding field in the other term to the same value. If provision must be made for a rule's becoming invalid (and not just simplifying away, as happens in Knuth-Bendix) then the routine that removes a rule will also have to reset the fields to their initial values.

The simplification routine (called *Reduce_term* in the program) starts at the root node of a term. It first checks the *canonical* field; if it is *true*, then no further work needs to be done. If it is *false*, then the routine checks the *simplifies_to* field to see if it already knows how to simplify the term. If it knows a simplification, then the chain of *simplifies_to* pointers is collapsed until the field for each term in the chain points to a node whose *simplifies_to* field is **nil**. (A beneficial side effect of the collapsing feature is that looping simplifications, which generate a repeating sequence of terms, are detected. This feature does not, however, detect all cases of nontermination [3].)

If the *canonical* field is *false* and the *simplifies_to* field is **nil**, then the node must be processed by the simplifier. First, the simplifier calls itself recursively on the children of the node, and checks the resulting term to see if it is different from the original term. If the resulting term is different, it is checked to see if it is already known to be in simplest form or has a known simplification, the *simplifies_to* field of the original term is set to point to the simplified term, which becomes the current node. Finally, if the current node is not canonical and does not have a known simplification, the simplification routine tries applying each rule in the system to the root of the current node. If the term cannot be simplified, its *canonical* field is set to *true*. If it can be simplified, then the *simplifies_to* field is set to point to the resulting simplified term, which must itself be checked for further simplifications. This process continues until a term which cannot be simplified any more is obtained. The actual code for the simplification routine is in Appendix 1.

We also have a specialized version of the simplifier (called *Reduce_term_with_one_rule*) that tries to simplify an expression using a single rule rather than a set of rules. Such a routine is useful when a new rule is added to the system: if an expression (which has already been simplified using the old rules) cannot be further simplified using the new rule, then it cannot be simplified using the new rule in conjunction with the old ones. The attempt to simplify using a single rule is much cheaper than trying the whole set of rules. If a simplification is found, then the expression must be further processed using the complete set of rules.

An important property of a simplification method is the order in which a term and its subterms are simplified. The algorithm in this paper uses an order which is similar to leftmost-innermost order. (An *innermost* order is one that works bottom-up, simplifying subterms before the main term; a *leftmost* order processes subterms from left to right.) The present algorithm differs from leftmost-innermost by applying known simplifications to a term before doing any other work. Such reductions are still innermost as long as the algorithm is used with a fixed set of rules; but when a rule is added after some simplifications have already been done, the innermost order may be violated on later simplifications. Our measurements show that, for our test problems, resetting the *simplifies_to* fields to **nil** when a rule is added does not slow down matching by more than 4 percent. (These measurements are not shown in the tables in Section 4.) Resetting *simplifies_to* will ensure a strictly innermost simplification order.

To ensure complete simplification it is necessary, when a term simplifies, to recheck the resulting term and its subterms for further simplifications. Gallier and Book [5] point out that, for innermost reduction orders, it is not necessary to recheck those subterms that result from the substitution of values for variables in the initial simplification; those values are guaranteed to be in simplest form. If normalized rules (rules that have been simplified with respect to each other) are used, then it is also not necessary to resimplify those subterms that are obtained by copying variable-free subterms of the right side of a rule. The only nodes that need to be resimplified are those that are copies of right side nodes which have variable nodes as descendants.

Our algorithm does not make direct use of either of these principles, but these cases are among those it avoids resimplifying by using the information in the *canonical* field. The values of the variables are simplified during the earlier stages of the process, and the variable-free parts of right sides of rules are simplified when the rules are derived. When a simplification is done, only those nodes which are copies of right side nodes with variables as descendants have their *canonical* fields set to indicate the need for further processing.

An even more powerful method of avoiding resimplification (which we did not try) begans by attempting to unify each nonvariable right side node with the root of each left side. During a simplification the only nodes that need to be considered for resimplification are those nodes that are copies of right side nodes which unify with the left side of a rule. (For normalized rules, these will be a subset of the right side node with variable nodes as descendants.) This approach is surely not worthwhile for the Knuth-Bendix procedure on small problems, because the set of rules changes too rapidly to make up for the unification time, but it would probably be worthwhile for applications that used a fixed set of simplification rules.

3. Matching

Even with the improvements given above, the simplification algorithm still spends most of its time doing matching. The studies for this paper were done using a simple matching algorithm that tries to match a term with one rule at a time. (We are also investigating methods that match all the rules at once; such methods rely heavily on the use of a dag to represent the set of rules. A preliminary report of that work is given in [14].) The simple approach that we describe here works well, and we use it as a standard of comparison when studying more sophisticated algorithms.

The matching algorithm of this paper does have two clever features, which require a *contains_variables* field and a *match* field for each pattern node. The *contains_variables* field is set to *true* if the node corresponds to a term that contains at least one variable. If the current attempt at matching has already matched the pattern node with a subject node, then the *match* field of the pattern node points to the subject node. Otherwise, the *match* field is **nil**.

The matching algorithm tries to match a single subject with a single pattern. It begins at the root nodes for the pattern and the subject. The root of the pattern may contain a variable, a constant, or an operator. (The initial call to the match routine will never have a variable as the pattern, but the recursive calls that the match routine makes may have a variable as the pattern.) If the root of the pattern contains a variable that has not yet been assigned a value (i.e. its *match* field is **nil**), then the pattern matches. If the root of the pattern contains a variable that has already been assigned a value, the pattern matches if and only if the subject is identical to the value of the variable (contained in the *match* field of the pattern node). Because of the dag data structure, the subject and the value of the variable must in fact be the same node. This means that the test for equality of the subject and the value of the variable can be done with a single pointer comparison.

If the root of the pattern contains a constant, it can match only itself. Because of the dag, in this case the pattern matches the subject only if they are the same node.

The remaining case is that the root of the pattern is an operator with one or more children. In this case we first check that the subject has the same main operator. If the *match* field of the pattern node is not **nil**, the subject must be the same node as the value of the *match* field. If the *contains_variables* field is *false*, the pattern and the subject match if and only if they are the same node. Finally, if the pattern and the subject have the same main operator, the pattern has a **nil** *match* field, and the *contains_variables* field is *true*, the pattern matches the subject if and only if their corresponding children match. The remaining cases do not lead to a match.

In all cases, if a match is found, the *match* field of the pattern node is set to point to the subject node.

4. Performance

The speed of the simplification algorithm was measured by running the Knuth-Bendix procedure on three algebras:

1. Group theory:

$$\begin{aligned}&\times(\times(A,B),C)\rightarrow\times(A,\times(B,C))\\&\times(i(A),A)\rightarrow e\\&\times(e,A)\rightarrow A,\end{aligned}$$

2. Knuth's second version of central groupoids [10]:

$$\begin{aligned}&\times(\times(A,A),A)\rightarrow a(A)\\&\times(A,\times(A,A))\rightarrow b(A)\\&\times(\times(A,B),\times(B,C))\rightarrow B\\&\times(b(A),B)\rightarrow\times(A,B),\end{aligned}$$

3. The dihedral group of order 8:

$$\begin{aligned}&\times(\times(A,B),C)\rightarrow\times(A,\times(B,C))\\&\times(i(A),A)\rightarrow e\\&\times(e,A)\rightarrow A\\&\times(a,\times(a,\times(a,\times(a,\times(a,\times(a,\times(a,a)))))))\rightarrow e\\&\times(b,b)\rightarrow e\\&\times(a,\times(b,\times(a,b)))\rightarrow e.\end{aligned}$$

The first algebra is a typical small problem for the Knuth-Bendix procedure, and has been used to test the running time of many programs [7, 8, 10, 14]. The completed rule set for this algebra has ten rules, and the intermediate sets (which contain some rules that eventually simplify away) are not much bigger. The second algebra is much more difficult. The final set has only 13 rules, but the intermediate sets have up to forty rules. This algebra has no constants. The time for this algebra has been previously measured in [10, 14]. The third algebra is also difficult; its final rule set has 30 rules. The first algebra requires a few seconds. The second and third ones take tens of seconds.

Using naive implementations of matching and simplification, the Knuth-Bendix procedure spends more time in those routines than in all other parts of its computation combined, according to our measurements on these and other typical algebras. Using the algorithms in this paper reduces the running time for matching and simplification by a factor of 2.5 to 12.8, depending on the particular algebra being completed. The total time for the Knuth-Bendix procedure is reduced by a factor of up to 4.1. The savings factor is largest for the largest algebra. With the improved algorithm, simplification still uses more time than any other single part of the procedure, but it no longer uses more time than all the other parts put together.

The Knuth-Bendix program was written using the Web programming system [9]. The Web system produces a Pascal program as one of its outputs. (The other output is a nicely formatted listing, as in Appendix 1.) The time for the simplification routine (including the time used by the subroutines it calls) and for the entire Knuth-Bendix procedure was measured using the *clock* function of the Berkeley Pascal compiler. The program ran on a VAX11/780 computer. The *clock* function reports running time in units of one millisecond.

Running times measured in this way are subject to two sources of error. First, there are random effects that show up when a program is run repeatedly. These random effects can be reduced by averaging the times from several runs of the program, and the size of these effects can be estimated by studying the variation in the running time from repeated runs. Second, there are nonrandom effects. Since the execution time of the routines is comparable to the clock period, the time for a routine can appear too high if a clock pulse always comes near the start of the routine, or too low if one comes just after the end of the routine. (If the phase of the clock is random, then the error is also random, but if the phase of the clock is correlated with the routine then the effect is not entirely random.) Also, the measured time depends somewhat on how busy the computer is; it appears to go up about 10 percent when the load factor is large.

In order to gauge the relative importance of the features added to the simplification and matching algorithms, we measured the running times of algorithms that included various combinations of features. Changes in the running times for the simplification algorithms show the importance of each feature. The variation in the running time for the rest of the program is also interesting, since it gives an indication of the magnitude of the nonrandom errors in the measurements.

To obtain a reliable value for the running time, and a reliable estimate of the error, the Knuth-Bendix program containing each combination of features was run five times on each algebra. The resulting errors were usually less than 10 percent. Since special features of the algorithms reduced the running time by more than a factor of 2, the exact size of the errors has no bearing on many of the conclusions.

Table 1 shows the amount of time used by various modifications of *Reduce_term*, the

Feature				Problem		
c	s	v	m	group	groupoid 2	dihedral 8
				96 (± 6)	1518 (±44)	4440 (±165)
+				39 (± 4)	404 (±28)	912 (± 26)
	+			75 (± 5)	948 (±14)	1858 (± 34)
		+		99 (± 6)	1600 (±42)	3982 (±154)
			+	102 (±15)	1564 (±58)	3918 (± 48)
	+	+	+	72 (± 9)	1067 (±75)	1716 (± 53)
+		+	+	46 (± 4)	397 (±13)	812 (± 24)
+	+		+	29 (± 7)	220 (±25)	303 (± 28)
+	+	+		30 (± 6)	228 (± 8)	262 (± 21)
+	+	+	+	29 (± 6)	220 (±15)	261 (± 12)

Table 1. The running time required for simplification using a set of rules. The running time is shown as a function of the features in the algorithm and of the problem. In the feature column, a plus indicates that a feature is used while a blank indicates that it is not. The time is given in hundredths of a second.

major simplification routine. Table 2 shows the same information for *Reduce_term_with_one_rule*, and Table 3 shows the same information for the rest of the program. The feature associated with each field is indicated in the tables by 'c' for the *canonical* field, 's' for the *simplifies_to* field, 'v' for the *contains_variables* field, and 'm' for the *match* field. The time for each part includes the time spent in subroutines called by that part. It is given in hundredths of a second (rounded to the nearest integer). Following each time is the expected error for the time, based the variation in the time used by five consecutive runs. For each of the three algebras, the variation of the times in Table 3 is about as expected (based on the variation within each group of five runs). Preliminary runs gave similar results, except that occasionally the variation between runs was several times the expected amount. Caution should be observed when comparing times that are closer than four times the stated standard errors: differences in such times may not be statistically significant. To give an idea of the overall speed of the simplification algorithm, it processes about 500 nodes per second.

Some will consider our naive algorithm to be slightly too simple. The variable nodes never simplify. Skipping variable nodes would speed up the naive version of the algorithm applied to the groupoid 2 problem by 30 percent. For the group problem the speed-up would be 25 percent and for the dihedral 8 problem the speed-up would be 3 percent. Similar results apply to all versions of the algorithm that do not use the *canonical* field. Those versions of the algorithm that use the canonical field would run slightly slower, because variable nodes are already marked as canonical.

The measurements clearly indicate that the improved algorithm is much faster than the naive algorithm for all three problems. They also clearly indicate that the *canonical* field is responsible a large proportion of the improvement, while the *simplifies_to* and *contains_variables* fields are responsible for smaller but still significant improvements.

Counting how often each step of the algorithm is done gives a second way to determine

Feature				Problem		
c	s	v	m	group	groupoid 2	dihedral 8
				14 (± 2)	194 (±15)	169 (± 5)
+				15 (± 4)	106 (±12)	109 (±10)
	+			21 (± 5)	193 (± 7)	175 (± 6)
		+		17 (± 2)	199 (±39)	167 (±25)
			+	18 (± 2)	199 (±26)	162 (± 6)
	+	+	+	19 (± 5)	217 (± 7)	167 (±13)
+		+	+	14 (± 4)	104 (± 6)	115 (± 8)
+	+		+	12 (± 4)	110 (±10)	110 (±12)
+	+	+		13 (± 5)	115 (± 6)	105 (±15)
+	+	+	+	15 (± 3)	112 (±10)	98 (± 9)

Table 2. The running time required for simplification using one rule.

Feature				Problem		
c	s	v	m	group	groupoid 2	dihedral 8
				154 (±10)	875 (±16)	1050 (±28)
+				147 (± 6)	882 (±41)	1054 (±25)
	+			139 (± 5)	839 (±26)	1028 (±47)
		+		152 (± 6)	865 (±12)	1082 (± 9)
			+	145 (±15)	829 (±48)	1026 (±28)
	+	+	+	140 (±15)	805 (±32)	981 (±31)
+		+	+	136 (± 7)	784 (±17)	922 (±13)
+	+		+	138 (± 7)	833 (±69)	938 (±22)
+	+	+		146 (± 6)	857 (±25)	1027 (±22)
+	+	+	+	141 (± 3)	792 (±36)	900 (±21)
Average				144 (± 1)	836 (± 3)	1001 (± 6)

Table 3. The running time required for the Knuth-Bendix procedure exclusive of matching and simplification.

how important each feature is. To save space, we omit the detailed counts, but we include the following conclusions based on the counts. (For more details consult [15].)

Most of the terms that need to be simplified are already known to be in simplest form. When a term is not marked as canonical, it is unlikely that any simplification is known for it. (Of course, a lot of time is saved when a simplification is known for a large term). Just as it is unusual for a term to have a simplification, it is unusual for the children of a term to have a simplification. Simplification takes a long time because each term that might need more simplification must be matched against each rule.

There are methods for doing matching that don't require a direct comparison between each rule and each term [6, 14]. These methods, however, require a good bit of preprocessing. The authors are investigating these other methods, but so far we obtain approximately the same performance with the simple methods as with the sophisticated methods. Our present advice is to first program the simple method and then try one of the more sophisticated methods if the simple method is too slow.

Most matches terminated quite early. The total number of calls to the match routine is about twice the total number of calls from *Reduce_term* and *Reduce_term_with_one_rule*. Therefore, the match routine usually had to look at about two nodes before it could determine that a rule did not apply. One simple idea that should often be effective in speeding up matching is to divide the rules into lists based on the main operator of the left side [11]. In this way many hopeless attempted matches can be avoided altogether.

Since it was rare for a term to have a known simplification, the running time would not increase much if the collapsing feature in applying known simplifications were eliminated. Elimination of the collapsing feature would permit a simple modification of the algorithm to cause it to report each rule as the rule was applied. If previously known simplifications were considered only after the children of a node were simplified, or if the *simplifies_to* field were reset after each rule was added to the system, the algorithm would always produce leftmost-innermost simplifications. Again this would not have a large effect on the running time.

When the simplification algorithm of this paper applies a rule, it first finishes applying the rule and then simplifies the result. If the result is not in simplest form, a little extra work has been done. It might be slightly faster to check the result as it was being produced and resimplify each piece of it at that time. This would, however, require a new data structure to contain the values for variables. At present the values are stored in the match fields of the variables, so it is not possible to start a second simplification until the first one is finished. Since rules are not applied very often, nested simplification cannot save much time. If it uses data structures that slow down matching, it can cost time.

Using the *match* field to avoid rematching nodes saved no time on two of the three algebras; it just contributed a small amount of overhead. There are, of course, other matching problems where the *match* field would be quite useful in reducing the time. We can also see from Table 4 that for two of the three algebras the *contains_variables* field did not speed up the matching of terms with operators as the main operation. The main effect of the *contains_variables* field was to speed up the matching of lists that had a single constant term. With some recoding, the *contains_variables* field of terms could be eliminated while retaining the *contains_variables* field for list cells. This would leave the running time the same, save some space in the data structure, and make the program more complex.

The Knuth-Bendix procedure that was used to test the algorithms did no garbage collection. The amount of memory used was not too large, however, because new nodes were built only when a suitable node did not exist. Table 5 shows, for example, that for the dihedral algebra the system considered building 11,890 term nodes [(*Rebuild_term*) − (*Build_term* in *Rebuild_term*) + (*Build_term*)], but that it only built 823 term nodes. If garbage collection were to be done, the logical time to do it would be just after a new rule was introduced. For the dihedral algebra this would result in 11619 nodes been considered for garbage collection (so not much time would be spent). The effect of this on simplification time would be no more than that of resetting the *simplifies_to* field to nil after each new rule, which increases the time by less than 4 percent.

5. Conclusions

In summary, our experiments show that the combination of the dag data structure and remembering which terms have previously been found to be in canonical form improves the speed of the simplification algorithm substantially. Other features, such as remembering for each noncanonical term a way to simplify the term, remembering whether a term contains variables, and avoiding rematching nodes in the dag that are visited twice, were also useful on the problems studied. The algorithm is only slightly more difficult to program than the naive program, and it leads to substantial savings in time.

Appendix 1

This section gives the actual code for the simplification and matching algorithms. The paragraphs are numbered for easy reference to sections of code. The program is in Web [7], a language developed by Knuth for writing the TEX system for typesetting. The language is essentially Pascal with named modules.

1. Data structures.

The most important data type is *term*. A cell of type term is used for each node in the system. A node contains a symbol, specified by the *term_type* and *name* fields. The symbol can be an *operator*, a *constant*, or a *variable*. (A constant can be thought of as an operator of arity zero, but we use a special category for efficiency.) A node's *child_list* points to a list of its children. It is nil for variables and constants and non-nil for operators. Each node has a unique *number* field. The *contains_variables* field is *true* if the current node or any of its descendants is a variable. The *match* field is used by the match routine to record which node the current node matches. The *canonical* field is used to record whether or not the node is known to be in simplest form. The *simplifies_to* field points to a node that the current node is known to simplify to, or is nil. The *simplifies_to* field must be nil whenever the *canonical* field is *true*.

A term is represented by a minimum *dag* (directed acyclic graph). A node never occurs on its own list of children or as a descendant of any of its children. If two nodes have the same *term_type*, the same *name*, and the same *child_list*, then they must be the same node. This permits checking the equality of trees by comparing pointers.

[The fields have been designed for clarity. Two pointer fields can be often be saved in suitable languages, if space is a problem. The address of the node can be used in place of the *number* field. Many applications of simplification keep the rules in *normal form*, where neither the left or right side of any rule can be simplified by any other rule. When this is the case, the *match* and *simplifies_to* fields will never both contain non-nil values. In such cases one can save space by combining them together. Alternately, one can save one bit per node by combining the *canonical* and *simplifies_to* fields into a single field that has a value for *true* unequal to any pointer. In a similar way, two bits can be saved by combining the *name_type* and *child_list* fields. If you use normalized rules and save all the space, then you only need room for the *name* field and two pointers (where the pointer fields must be able to hold a couple of extra values).]

```
⟨Types. 1⟩ ≡
  term_pointer = ↑term; term_list_pointer = ↑term_list;
  name_type = (is_variable, is_constant, is_operator); name_index = integer;
  term = record term_type: name_type;
    name: name_index;
    child_list: term_list_pointer;
    number: integer;
    contains_variables: Boolean;
    match: term_pointer;
    canonical: Boolean;
    simplifies_to: term_pointer;
    end;
```

See also sections 2 and 3.

2. The second major data type used by the simplification and matching routines is the *term_list*, which is a list of terms. The *next* field points to the next list cell and the *first* field points to the term associated with the current list position. Each list cell has a unique *number* field. The *contains_variables* field is *true* if any term on the list contains a variable.

Two lists that have the same *first* field and *next* field must use the same node. This permits checking the equality of lists by comparing pointers.

```
⟨Types. 1⟩ +≡
  term_list = record next: term_list_pointer;
    first: term_pointer;
    number: integer;
    contains_variables: Boolean;
    end;
```

3. A list of rules is made up of cells that have a *first* field that points to a rule and a *next* field that points to the next list cell. A rule is a list of two equivalent terms. It is represented by an *equal_list*. An *equal_list* cell has a *first* field that points to a term and a *next* field that points to the next list cell. The left side of a rule is the first term on the list and the right side is the second term. For *equal_list* cells, there is no requirement that two cells with the same *first* field and the same *next* field use the same node. (This is convenient in parts of the Knuth-Bendix algorithm that are not given here. The *equal_list* cells do not take part in the matching algorithm.)

```
⟨Types. 1⟩ +≡
  rule_list_pointer = ↑rule_list; equal_list_pointer = ↑equal_list;
  rule_list = record next: rule_list_pointer;
    first: equal_list_pointer;
    end;
  equal_list = record next: equal_list_pointer;
    first: term_pointer;
    end;
```

4. Simplification and Matching Routines. The following routines are declared on the top level.

```
⟨Global procedures. 4⟩ ≡
  ⟨Function Reduce term. 5⟩
  ⟨Function Reduce term with one rule. 11⟩
  ⟨Function Match term. 16⟩
  ⟨Function Copy reduction term. 19⟩
  ⟨Procedure Reset match term. 21⟩
```

5. Function Reduce term. The function *Reduce_term* is used to simplify a term. The function is called with the parameter *term* pointing to the term that needs to be simplified and the parameter *rules* pointing to the list of rules to be used for simplification. It reduces *term* and all of its subterms as much as possible by repeated application of the rules on the list *rules*.

The function must process a term until it is reduced to a term whose *canonical* field is *true*. If the term is canonical to begin with, the function has nothing to do except set the value to the original term and return. If the term is not canonical, then it will be processed using a combination of three basic actions. These actions are, first, checking the *simplifies_to* field to see if the function already knows how to simplify the term (and if it does know, using the indicated term in place of the current one); second, simplifying the children of the term; and third, using the rules to try to simplify the term at the top level.

Checking the *simplifies_to* field is done first. If it is not nil, then the *canonical* field of the resulting term is checked. If that field is *true*, we are done. Otherwise, processing continues with the new term.

The next step is to simplify the children. If simplifications are found, then the current node is replaced with the node containing the simplified children, and the *canonical* and *simplifies_to* fields of the resulting node are checked as before. This step is repeated until no more simplifications of the children are found.

If the node is still not marked as canonical, then the next step is to try to apply one of the rules in the rule list to the top level expression of the node. If no rule applies, the node is marked as canonical and the simplification is over. If a rule does apply, the current term is replaced with the simplified term and the whole process is repeated starting with step one. Given a Noetherian set of rules, the process will eventually terminate.

The function *Reduce_term* uses the function *Reduce_list* to simplify the subterms of *term*. It uses *Match* to determine whether a particular rule matches *term*, *Copy_reduction_term* to apply rules, and *Build_term* to build a new node when the children of a node simplify. The function is written in an explicit branching style to avoid redundant tests.

The following numerical values are used in Pascal for the labels.

```
  define Resimplify = 1
  define Use_simplified_value = 2
  define Chase_exit = 3
  define Simplify_children = 4
  define Exit = 5
```

```
⟨Function Reduce term. 5⟩ ≡
function Reduce_term(rules : rule_list_pointer; term : term_pointer): term_pointer;
  label Chase_exit, Exit, Resimplify, Simplify_children, Use_simplified_value;
  var p, q: term_pointer; r: rule_list_pointer; l: term_list_pointer; ⟨Function Reduce list. 10⟩
  begin Resimplify: if ¬term↑.canonical then
    begin if term↑.simplifies_to = nil then goto Simplify_children;
  Use_simplified_value: ⟨Use the simplified value. 6⟩
  Simplify_children: ⟨Simplify children. 7⟩
    ⟨Try to apply the rules. 8⟩
    term↑.canonical ← true;
    end;
Exit: Reduce_term ← term;
  end;
```

This code is used in section 4.

6. Use the simplified value. The current term has a non-nil *simplifies_to* field. Follow the chain of *simplifies_to* pointers to obtain the simplified version of *term*. To save work in the future, collapse the chain by resetting each *simplifies_to* pointer to point to the last node on the chain. The resulting term is not the original term. It will have a nil *simplifies_to* field, but the *canonical* field may be either *true* or *false*, i.e., we don't know how to simplify the resulting term further, but we may or may not know whether it is completely simplified.

The routine *Error*, which issues error messages and stops, is called if the *simplifies_to* chain has a loop. The pointer *q* goes down the chain at one half the speed of *p*. The two pointers will eventually point to the same term if and only if the chain has a loop. This catches some, but not all, infinite derivations.

```
⟨Use the simplified value. 6⟩ ≡
  begin p ← term↑.simplifies_to; q ← p;
  while p↑.simplifies_to ≠ nil do
    begin p ← p↑.simplifies_to;
    if p↑.simplifies_to = nil then goto Chase_exit;
    p ← p↑.simplifies_to; q ← q↑.simplifies_to;
    if q = p then Error(simplification_loop);
    end;
Chase_exit: while term ≠ p do
    begin q ← term↑.simplifies_to; term↑.simplifies_to ← p; term ← q;
    end;
  if p↑.canonical then goto Exit;
  end;
```

This code is used in section 5.

7. Simplify children. The quick tests have now been done. The current term has no known simplification, so the routine tries to simplify the subterms. If any subterms simplify, then the routine builds a new term that is like the old one except that it has the list of simplified subterms instead of the original list. If the resulting term is known to be canonical, then the routine jumps to *Exit*. Otherwise, if the routine knows how to simplify the resulting term, then it jumps back to the first basic step, where it applies the known simplification.

```
⟨Simplify children. 7⟩ ≡
  l ← Reduce_list(rules, term↑.child_list);
  if l ≠ term↑.child_list then
    begin p ← Build_term(term↑.term_type, term↑.name, l); term↑.simplifies_to ← p; term ← p;
    if term↑.canonical then goto Exit;
    if term↑.simplifies_to ≠ nil then goto Use_simplified_value;
    end;
```

This code is used in section 5.

8. Try to apply the rules. The term now has all its children simplified, no simplifications are known for the term, but it is not known whether the term is completely simplified. Therefore, an attempt is made to simplify the term with each rule. This step takes time proportional to the number of rules, so it can be rather time-consuming. (It is a good candidate for the application of clever ideas.)

```
⟨Try to apply the rules. 8⟩ ≡
  r ← rules;
  while r ≠ nil do
    begin ⟨Try to match the term with the rule. 9⟩
    r ← r↑.next;
    end;
```

This code is used in section 5.

9. Try to match term *term* with rule *r*. Call *Match_term* to see if the left side of rule *r* matches *term*. If a match occurs, then (1) call *Copy_reduction* to replace *term* with the right side of *r* (after replacing the variables in the right side of *r* with the values of the variables that result in the match), (2) call *Reset_match_term* to erase the *match* fields of *r* (which are left set by *Match_term* if and only if a match is found), and (3) go back to *Resimplify* to see if any further simplification can be done.

```
⟨Try to match the term with the rule. 9⟩ ≡
  if Match_term(r↑.first↑.first, term) then
    begin p ← Copy_reduction_term(r↑.first↑.next↑.first); term↑.simplifies_to ← p; term ← p;
    Reset_match_term(r↑.first↑.first); goto Resimplify;
    end;
```

This code is used in section 8.

10. Function Reduce list. This function returns the list obtained by reducing all terms on the list *l* using the rules on *rules*.

```
⟨Function Reduce list. 10⟩ ≡
function Reduce_list(rules : rule_list_pointer; l : term_list_pointer): term_list_pointer;
  begin if l = nil then Reduce_list ← nil
  else Reduce_list ← Rebuild_list(l, Reduce_term(rules, l↑.first), Reduce_list(rules, l↑.next));
  end;
```

This code is used in section 5.

11. Function Reduce term with one rule. The function *Reduce_term_with_one_rule* is used to do one step of simplifying a term. The function is called with the parameter *term* pointing to the term that needs to be simplified and the parameter *rule* pointing to a single rule to be used for simplification. It examines *term* (and its subterms) until it finds a term that can be reduced by *rule*. If *term* simplifies (either directly or as a result of simplifying one of its subterms), the *simplifies_to* field of *term* is set to point to the resulting term. If *term* does not simplify, then its *canonical* field is set to *true*. The value of *Reduce_term_with_one_rule* is the term that results from reducing the original term.

The function *Reduce_term_with_one_rule* is used on terms that have already been simplified by an old set of rules. All terms in the system begin with their *canonical* fields set to *false* and their *simplifies_to* fields set to whatever value they had as a result of the simplifications using the old rule set. As terms are simplified using the new rule (together with the old ones) these values change.

This function is intended for normalizing previous rules when a new rule is added to the system. Such rules are expected to be completely simplified with respect to each other. If the term being simplified does not simplify with the new rule, then it is in simplest form for the complete set of rules. If it does simplify, then the result of the first simplification should not be further simplified, because simplifications by the old rules will be missed (leading to incorrectly set *canonical* fields). Instead the term with its first simplification should be saved for later simplification by the complete set of rules. [It is much quicker to use *Reduce_term_with_one_rule* to find those terms that need further simplification by *Reduce_term* than it is to simplify all the nodes with *Reduce_term* in the first place. The function *Reduce_term_with_one_rule* is usually much quicker than *Reduce_term* because it uses just one rule instead of a (possibly) long list of rules.]

```
⟨Function Reduce term with one rule. 11⟩ ≡
function Reduce_term_with_one_rule(rule : equal_list_pointer; term : term_pointer): term_pointer;
  label Exit;
  var p: term_pointer; l: term_list_pointer; ⟨Function Reduce list with one rule. 15⟩
  begin if ¬term↑.canonical then
    begin if term↑.simplifies_to ≠ nil then ⟨Use the simplified value with one rule. 12⟩
    ⟨Simplify children with one rule. 13⟩
    ⟨Try to apply the rule. 14⟩
    term↑.canonical ← true;
    end;
Exit: Reduce_term_with_one_rule ← term;
  end;
```

This code is used in section 4.

12. Use the simplified value with one rule. The term has a non-nil *simplifies_to* field. Apply the simplification and jump to *Exit*.

```
⟨Use the simplified value with one rule. 12⟩ ≡
  begin term ← term↑.simplifies_to; goto Exit;
  end;
```

This code is used in section 11.

13. **Simplify children with one rule.** The current term has no known simplification, so the routine tries to simplify the subterms. If any subterms simplify, then the routine builds a new term that is like the old one except that it has the simplified list of subterms instead of the original list. If a new term is built, then the routine jumps to *Exit*.

```
⟨Simplify children with one rule. 13⟩ ≡
  l ← Reduce_list_with_one_rule(rule, term↑.child_list);
  if l ≠ term↑.child_list then
    begin p ← Build_term(term↑.term_type, term↑.name, l); term↑.simplifies_to ← p; term ← p;
    goto Exit;
    end;
```

This code is used in section 11.

14. **Try to apply the rule.** See if the rule *rule* can simplify the node. If it does, apply the rule, reset the *match* fields, and go to exit.

```
⟨Try to apply the rule. 14⟩ ≡
  if Match_term(rule↑.first, term) then
    begin p ← Copy_reduction_term(rule↑.next↑.first); term↑.simplifies_to ← p; term ← p;
    Reset_match_term(rule↑.first); goto Exit;
    end;
```

This code is used in section 11.

15. **Function Reduce list with one rule.** This function returns the list obtained by reducing the first reducible term on the list *l* using the rule *rule*.

```
⟨Function Reduce list with one rule. 15⟩ ≡
function Reduce_list_with_one_rule(rule : equal_list_pointer; l : term_list_pointer): term_list_pointer;
  var p: term_pointer;
  begin if l = nil then Reduce_list_with_one_rule ← nil
  else begin p ← Reduce_term_with_one_rule(rule, l↑.first);
    if p = l↑.first then
      Reduce_list_with_one_rule ← Rebuild_list(l, p, Reduce_list_with_one_rule(rule, l↑.next))
    else Reduce_list_with_one_rule ← Build_list(p, l↑.next);
    end;
  end;
```

This code is used in section 11.

16. Function Match term. This function returns *true* if the variables in the *pattern* can be set in such a way as to make the *pattern* the same as the *subject*. Variables in the *subject* cannot be set and are regarded as distinct from the variables in the *pattern* (even if they happen to have the same name), so each variable in the *subject* acts like a distinct constant. (This is the difference between matching and unification; when unifying two terms, the variables in both terms can be set during the attempt to make the two terms identical, and variables with the same name are treated as the same variable, wherever they occur.)

For the initial call to *Match_term* (in *Reduce_term* or in *Reduce_term_with_one_rule*), the *match* fields for all the nodes in the *pattern* must be **nil**. If *Match_term* finds a match, then it will set the *match* field of each node of *pattern* to point to the corresponding node of *subject* (unless the *pattern* node is for a term with no variables). In this case, it is the responsibility of the calling routine to reset the *match* fields back to **nil** (by calling *Reset_match*). If there is no match, then *Match_term* resets the *match* fields back to **nil**.

When there is a match, the *match* fields of variables are used by *Copy_reduction_term* to determine what values the variables should have when the rule (for which *pattern* is the left side) is applied. Each nonroot node that has a non-**nil** *match* field must have at least one ancestor with a non-**nil** *match* field, or it will not be found and reset by *Reset_match*. This is why *match* fields are set on operator nodes. In addition, the *match* field is used to speed up the matching. If a node in the *pattern* dag is visited more than once, the *match* field can be used on the later visits to determine whether the node matches the subject node or not. If the *pattern* node has no variables, it matches the *subject* if and only if it is the same node as the *subject* node. In this case it is quicker to test for matching directly than it is to make use of the *match* field.

The function *Match* uses the function *Match_list* to determine if the children of a pattern node match the corresponding children of the *subject* node.

A *pattern* variable matches the *subject* if its value has not been set (its *match* field is **nil**) or if its value has been set to the *subject* (*match* = *subject*).

A *pattern* constant matches the *subject* if and only if it is the same node as the *subject*.

A *pattern* operator matches the *subject* if and only if the subject node has the same operator and the corresponding children of the two nodes match. If the *pattern* has no variables, this is equivalent to the *pattern* and *subject* being the same node. When an operator node matches, its *match* field is set to point to the *subject* (if the *pattern* has variables). If the *pattern* node is visited again during the same match, whether it matches the *subject* node of the second visit can be quickly determined by using the *match* field to see if the second *subject* node is the same as the first one.

```
⟨Function Match term. 16⟩ ≡
function Match_term(pattern : term_pointer; subject : term_pointer): Boolean; ⟨Function Match list. 18⟩
    begin case pattern↑.term_type of
    is_variable: if pattern↑.match = nil then ⟨Match is true. 17⟩
      else Match_term ← pattern↑.match = subject;
    is_constant: Match_term ← pattern = subject;
    is_operator: if pattern↑.name = subject↑.name then
        if subject↑.term_type = is_operator then
          if pattern↑.contains_variables then
            if pattern↑.match = nil then
              if Match_list(pattern↑.child_list, subject↑.child_list) then ⟨Match is true. 17⟩
              else Match_term ← false
            else Match_term ← pattern↑.match = subject
          else Match_term ← pattern = subject
        else Match_term ← false
      else Match_term ← false;
      endcases;
    end;
```

This code is used in section 4.

17. **Match is true.** Record the match and set the function value.

```
⟨Match is true. 17⟩ ≡
  begin pattern↑.match ← subject; Match_term ← true;
  end
```

This code is used in sections 16 and 16.

18. **Function Match list.** Match each pattern on list *l* with the corresponding subject on list *m*. If the two lists don't match, then call *Reset_match_term* to reset the *match* fields that were set by *match_term* for those patterns that did match. This function stops trying to match as soon as the first term which does not match is found.

```
⟨Function Match list. 18⟩ ≡
function Match_list(l : term_list_pointer; m : term_list_pointer): Boolean;
  begin if l = nil then Match_list ← m = nil
  else if m = nil then Match_list ← false
    else if l↑.contains_variables then
        if Match_term(l↑.first, m↑.first) then
          if Match_list(l↑.next, m↑.next) then Match_list ← true
          else begin Reset_match_term(l↑.first); Match_list ← false;
            end
        else Match_list ← false
      else Match_list ← l = m;
  end;
```

This code is used in section 16.

19. **Function Copy reduction term.** This function returns a new term obtained from *p* by replacing the variables in *p* with their values. The *match* field of each variable node points to the variable's value. The function *Copy_reduction_list* is used to build the new list of children for *p*. For terms that contain no variables, the old and new term are the same.

```
⟨Function Copy reduction term. 19⟩ ≡
function Copy_reduction_term(p : term_pointer): term_pointer; ⟨Function Copy reduction list. 20⟩
    begin case p↑.term_type of
    is_variable: if p↑.match = nil then Error(unbound_variable)
      else p ← p↑.match;
    is_constant: ;
    is_operator: if p↑.contains_variables then p ← Rebuild_term(p, Copy_reduction_list(p↑.child_list));
      endcases;
      Copy_reduction_term ← p;
    end;
```

This code is used in section 4.

20. **Function Copy reduction list.** This function returns a new list obtained from *l* by replacing the variables in *p* with their values. The *match* field of each variable node points to the variable's value. For lists that contain no variables, the old and new lists are the same.

```
⟨Function Copy reduction list. 20⟩ ≡
function Copy_reduction_list(l : term_list_pointer): term_list_pointer;
  begin if l = nil then Copy_reduction_list ← nil
  else if l↑.contains_variables then
      Copy_reduction_list ← Rebuild_list(l, Copy_reduction_term(l↑.first), Copy_reduction_list(l↑.next))
    else Copy_reduction_list ← l;
  end;
```

This code is used in section 19.

21. Procedure Reset match term. This procedure resets the *match* fields of *p* and its children back to nil. It is necessary that every node which needs resetting can be reached from *p* by a path of nodes with non-nil *match* fields. This restriction makes it possible to avoid repeated resetting of nodes in the dag.

```
⟨Procedure Reset match term. 21⟩ ≡
procedure Reset_match_term(p : term_pointer); ⟨Procedure Reset match list. 22⟩
  begin if p↑.match ≠ nil then
    begin p↑.match ← nil; Reset_match_list(p↑.child_list);
    end;
  end;
```

This code is used in section 4.

22. Procedure Reset match list.

```
⟨Procedure Reset match list. 22⟩ ≡
procedure Reset_match_list(l : term_list_pointer);
  begin if l ≠ nil then
    begin Reset_match_term(l↑.first); Reset_match_list(l↑.next);
    end;
  end;
```

This code is used in section 21.

23. Supporting data structures and routines. A brief discussion of the four routines used to produce term and list cells follows. The data structure is a minimum dag. Two terms that have the same operator and the same list of children *must* be represented by the same node. Likewise, two lists that have the same first term and the same tail *must* be represented by the same node. Hash tables are used to quickly look up nodes based on these characteristic fields.

The routines *Build_term* and *Build_list* are used to find the term or list node to use (and to build the node if necessary). The routines *Rebuild_term* and *Rebuild_list* are used to find replacements for old nodes in cases where the old node may be the same as the new node. These routines return the old node if the two nodes are the same and call *Build_term* or *Build_list* if they are different.

24. Types for the term hash table. The hash table has *term_hash_size* buckets. Each bucket consists of a list of the items in the bucket. The list cells have type *term_hash_cell*. Each list cell has a pointer to its term (the *term* field) and to the next cell on the list (the *next* field). The head of the list has type *term_hash_head*. Each head has a pointer to the list (the *hash_list* field) and a pointer to the next nonempty bucket (the *link* field).

The size of the term hash table is *term_hash_size*, and *term_hash_sizem* is *term_hash_size* − 1.

```
  define term_hash_size = 863
  define term_hash_sizem = term_hash_size − 1
⟨Global types. 24⟩ ≡
  term_hash_index = 0 .. term_hash_size; term_hash_pointer = ↑term_hash_cell;
  term_hash_cell = record next: term_hash_pointer;
    term: term_pointer;
    end;
  term_hash_head = record hash_list: term_hash_pointer;
    link: term_hash_index;
    end;
```

See also section 27.

25. Term hash table. The term hash table is an array of cells of type *term_hash_head*. The array has *term_hash_size* + 1 entries, but the last one is not used. The variable *term_hash_avail* is the index of some nonempty bucket (the value *term_hash_size* is used to indicate that there is no such bucket). The nonempty buckets are linked together through their *link* fields. This list is used for resetting the *canonical* fields of all nodes when a new rule is added to the system. The variable *term_hash* is used for initializing the hash table.

```
⟨Global variables. 25⟩ ≡
node_number: integer;
term_hash: term_hash_index;
term_hash_avail: term_hash_index;
term_hash_table: array [term_hash_index] of term_hash_head;
```

See also section 28.

26. Initially, the number of nodes is zero, and the hash table is empty.

```
⟨Global initialization. 26⟩ ≡
  node_number ← 0; term_hash_avail ← term_hash_size;
  for term_hash ← 0 to term_hash_sizem do term_hash_table[term_hash].hash_list ← nil;
```

See also section 29.

27. Types for the list hash table. The hash table has *list_hash_size* buckets. Each bucket consists of a list of the items in the bucket. The list cells have type *list_hash_cell*. Each list cell has a pointer to its entry (the *cell* field) and to the rest of the list (the *next* field).

The size of the list hash table is *list_hash_size*, and *list_hash_sizem* is *list_hash_size* − 1.

```
  define list_hash_size = 863
  define list_hash_sizem = list_hash_size − 1
⟨Global types. 24⟩ +≡
  list_hash_index = 0 .. list_hash_sizem; list_hash_pointer = ↑list_hash_cell;
  list_hash_cell = record next: list_hash_pointer;
    cell: term_list_pointer;
    end;
```

28. List hash table. The list hash table is an array of cells of type *list_hash_head*. The array has *list_hash_size* entries.

```
⟨Global variables. 25⟩ +≡
list_node_number: integer;
list_hash: list_hash_index;
list_hash_table: array [list_hash_index] of list_hash_pointer;
```

29. Initially, the number of list cells is zero, and the hash table is empty.

```
⟨Global initialization. 26⟩ +≡
  list_node_number ← 0;
  for list_hash ← 0 to list_hash_sizem do list_hash_table[list_hash] ← nil;
```

30. Function Build term. This function returns a pointer to the cell with the correct *term_type*, *name*, and *child_list* fields. It builds the cell if the system does not already hold it. The new cell has all of its fields correctly initialized.

The following numerical values are used for labels.

```
  define Term_found = 1
  define Build_term_exit = 2
```

```
⟨Function Build term. 30⟩ ≡
function Build_term(term_type : name_type; name : name_index; child : term_list_pointer): term_pointer;
  label Build_term_exit, Term_found;
  var p: term_pointer; l: term_hash_pointer;
  begin ⟨Hash term node. 31⟩
  ⟨Look up term in the hash table. 32⟩
  ⟨Make a cell for the term if it was not found. 34⟩
  ⟨Add the cell to the hash table. 35⟩
  goto Build_term_exit;
Term_found: Build_term ← l↑.term;
Build_term_exit: end;
```

31. Hash term node. Compute a hash index for the node based on its *term_type*, *name*, and *child_list* fields.

```
⟨Hash term node. 31⟩ ≡
  case term_type of
  is_variable: term_hash ← (name) mod term_hash_size;
  is_constant: term_hash ← (5 * name) mod term_hash_size;
  is_operator: term_hash ← (45 * name + child↑.number) mod term_hash_size;
    endcases;
```

This code is used in section 30.

32. Look up term in the hash table. If a hash table entry is found with the same *name*, *term_type*, and *child_list* fields then goto *Term_found*. Otherwise, fall through.

```
⟨Look up term in the hash table. 32⟩ ≡
  l ← term_hash_table[term_hash].hash_list;
  while l ≠ nil do
    begin if name = l↑.term↑.name then
      if term_type = l↑.term↑.term_type then ⟨Compare the child lists. 33⟩
    l ← l↑.next;
    end;
```

This code is used in section 30.

33. Compare the child lists. The hash table entry has the correct *name* and *term_type* fields. If it also has the correct *child_list* field then go to *Term_found*.

```
⟨Compare the child lists. 33⟩ ≡
  if child = nil then
    begin if l↑.term↑.child_list = nil then goto Term_found;
    end
  else if l↑.term↑.child_list ≠ nil then
      if child = l↑.term↑.child_list then goto Term_found;
```

This code is used in section 32.

34. Make a cell for the term if it was not found. The correct term is not in the hash table, so make the node. The node is initialized to show that it is not known to be simplified unless it is the node for a variable, it is not known to simplify to any other node, and it does not currently match any other node. The node corresponds to a term that contains variables if it is a variable or if its list of children contains some variables.

```
⟨Make a cell for the term if it was not found. 34⟩ ≡
  new(p); p↑.term_type ← term_type; p↑.name ← name; p↑.child_list ← child;
  node_number ← node_number + 1; p↑.number ← node_number; p↑.match ← nil; p↑.simplifies_to ← nil;
  if child = nil then
    begin p↑.contains_variables ← term_type = is_variable; p↑.canonical ← term_type = is_variable;
    end
  else begin p↑.contains_variables ← child↑.contains_variables; p↑.canonical ← false;
    end;
  Build_term ← p;
```

This code is used in section 30.

35. Add the cell to the hash table. Add the new cell at the front of its bucket. If the bucket was previously empty, add the bucket to the list of nonempty buckets.

```
⟨Add the cell to the hash table. 35⟩ ≡
  new(l); l↑.next ← term_hash_table[term_hash].hash_list; l↑.term ← p;
  if term_hash_table[term_hash].hash_list = nil then
    begin term_hash_table[term_hash].link ← term_hash_avail; term_hash_avail ← term_hash;
    end;
  term_hash_table[term_hash].hash_list ← l;
```

This code is used in section 30.

36. Function Build list. This function returns a pointer to the cell with the correct *first* and *next* fields. It builds the cell if the system does not already hold it. The new cell has all of its fields correctly initialized.

The following numerical values are used for labels.

```
  define List_cell_found = 1
  define Build_list_exit = 2
⟨Function Build list. 36⟩ ≡
function Build_list(first : term_pointer; next : term_list_pointer): term_list_pointer;
  label List_cell_found, Build_list_exit;
  var l: term_list_pointer; h: list_hash_pointer;
  begin ⟨Hash list node. 37⟩
  ⟨Look up list cell in the hash table. 38⟩
  ⟨Make a cell for the list cell if it was not found. 39⟩
  goto Build_list_exit;
List_cell_found: Build_list ← h↑.cell;
Build_list_exit: end;
```

37. Hash list node. Compute a hash index for the node based on its *first* and *next* fields. (The *first* field will be non-nil, but the *next* field may be nil.)

```
⟨Hash list node. 37⟩ ≡
  list_hash ← first↑.number mod list_hash_size;
  if next ≠ nil then list_hash ← (list_hash + 25 * next↑.number) mod list_hash_size;
```

This code is used in section 36.

38. Look up a list cell in the hash table. If a hash table entry is found with the same *first* and *next* fields then go to *List_found*. Otherwise, fall through.

```
⟨Look up list cell in the hash table. 38⟩ ≡
  h ← list_hash_table[list_hash];
  while h ≠ nil do
    begin if first = h↑.cell↑.first then
      if next = nil then
        begin if h↑.cell↑.next = nil then goto List_cell_found;
        end
      else if h↑.cell↑.next ≠ nil then
          if next = h↑.cell↑.next then goto List_cell_found;
    h ← h↑.next;
    end;
```

This code is used in section 36.

39. Make a cell for the list cell if it was not found. The correct cell is not in the hash table, so make the node and add it to the front of its bucket. The list contains variables if either the *first* part or the *next* part does.

```
⟨Make a cell for the list cell if it was not found. 39⟩ ≡
  new(l); list_node_number ← list_node_number + 1; l↑.number ← list_node_number; l↑.first ← first;
  l↑.next ← next; Build_list ← l; new(h); h↑.next ← list_hash_table[list_hash]; h↑.cell ← l;
  if next = nil then l↑.contains_variables ← first↑.contains_variables
  else l↑.contains_variables ← first↑.contains_variables ∨ next↑.contains_variables;
  list_hash_table[list_hash] ← h;
```

This code is used in section 36.

40. Function Rebuild term. This routine returns a pointer to a cell like the one for term *p*, except that it has *child* as its *child_list* field. When the child list is different, the routine calls *Build_term* to do the work. Otherwise, it just returns *p*.

```
⟨Function Rebuild term. 40⟩ ≡
function Rebuild_term(p : term_pointer; child : term_list_pointer): term_pointer;
  begin if p↑.child_list = child then Rebuild_term ← p
  else Rebuild_term ← Build_term(p↑.term_type, p↑.name, child);
  end;
```

41. Function Rebuild list. This routine returns a pointer to a cell like the one for list *old*, except that it has *first* and *next* as its *first* and *next* fields. When these fields are not the same for the new node as for the old node, the routine calls *Build_list* to do the work.

```
⟨Function Rebuild list. 41⟩ ≡
function Rebuild_list(old : term_list_pointer; first : term_pointer; next : term_list_pointer): term_list_pointer;
  begin if (old↑.first = first) ∧ (old↑.next = next) then Rebuild_list ← old
  else Rebuild_list ← Build_list(first, next);
  end;
```

Appendix 2

Place	group	groupoid	dihedral
Reduce_term	441	2060	2412
Resimplify	493	2305	2638
not canonical	200	1061	1329
chain length 1	63	503	553
chain length 2	8	42	127
chain length 3	5	10	37
canonical result	76	536	661
children	144	571	778
Build_term	42	102	345
canonical result	8	15	109
goto Use	20	46	110
Try	116	510	559
Match_term	844	9631	10629
Copy_term	52	245	226
Reduce_list	403	1553	2266
list nil	144	571	778
Reduce_one_term	584	5642	5179
not canonical	222	2001	1997
Reduce_one_list	221	1996	1996
Match_term	213	1985	1981
Copy_term	6	42	22
Reduce_one_list	573	5365	5414
list nil	213	1985	1981
Match_term	2213	23005	23435
variable	482	4057	2480
match=nil	415	3414	2172
constant	91	0	3562
operator	1640	18948	17393
=name	948	8996	12588
=type	875	8119	11754
has variables	848	8119	8087
Matchlist	848	8110	8087
matches	173	1198	817
Match_list	1404	12587	12116
list nil	169	1198	809
no variables	79	0	479

Table 4. The number of times various sections of the algorithm were executed.

Place	group	groupoid	dihedral
Build_term	1048	7238	5144
new term	197	546	823
Build_list	986	6372	7120
new cell	250	566	1124
Rebuild_term	1110	8796	10319
Build_term	579	4203	3593
Rebuild_list	2358	17341	23552
Build_list	965	6340	7070
Canonical reset	1112	11461	11619
Reset_match_term	1406	10688	10589
match not nil	912	6763	4841
Reset_match_list	1452	10530	8087
not nil	540	3767	3246
Copy_term	157	775	961
variable	94	446	422
constant	6	0	103
operator	57	329	436
has variables	57	329	389
Copy_list	156	817	1102
list nil	57	329	389

Table 5. The number of times various support activities were done.

References.

1. Paul Chew, *An Improved Algorithm for Computing with Equations*, 21st Annual Symposium on Foundations of Computer Science (1980) pp 108–117.
2. Martin Davis, *Arithmetical Problems and Recursively Enumerable Predicates*, The Journal of Symbolic Logic **18** (1953) pp 33–41. Also see Martin Davis, *Computability and Unsolvability*, McGraw-Hill, New York (1958).
3. Nachum Dershowitz, *Termination of Rewriting*, University of Illinois Dept. of Computer Science Tech. Report No. UIUCDCS-R-85-1220, First International Conference on Rewriting Techniques and Applications, Dijon France (1985), also in Lecture Notes on Computer Science No. 202, Rewriting Techniques and Applications, pp 180–224, Springer-Verlag, New York (1985).
4. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan, *Variations on the Common Subexpression Problem*, JACM **27** (1980) pp 758–771.
5. J. Gallier and R. V. Book, *Reductions in Tree Replacement Systems*, unpublished manuscript. See: D. Kapur and G. Sivakumar, *Experiments with and Architecture of RRL, A Rewrite Rule Laboratory*, Proceedings of an NSF Workshop on the Rewrite Rule Laboratory, General Electric (1984) p 39.
6. Christoph M. Hoffmann and Michael J. O'Donnell, *Pattern Matching in Trees*, JACM **20** (1982) pp 68–95.
7. Jean Marie Hullot, *Compilation de Formes Canoniques Dans des Théories Équationnelles*, thesis, l'Université de Paris-Sud, 1980.
8. Deepak Kapur and G. Sivakumar, *Experiments with and Architecture of RRL, A Rewrite Rule Laboratory*, Proceedings of an NSF Workshop on the Rewrite Rule Laboratory, General Electric (1984) p 33–47.
9. Donald E. Knuth, *Literate Programming*, The Computer Journal **27** (1984) pp 97–111.
10. Donald E. Knuth and Peter B. Bendix, *Simple Word Problems in Universal Algebras*, in John Leech (ed.), Computational Problems in Abstract Algebra, Pergamon, Oxford (1970) pp 263–297.
11. D. R. Musser, *Abstract Data Type Specification in the Affirm System*, IEEE Transactions on Software Engineering **6** (1980) pp 24–32.
12. Greg Nelson and Derek C. Oppen, *Fast Decision Procedures Based on Congruence Closure*, JACM **27** (1980) pp 356–364.
13. Michael J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press, Cambridge Mass. (1985) pp 215–219.
14. Paul Purdom and Cynthia Brown, *Fast Many-to-One Matching Algorithms*, First International Conference on Rewriting Techniques and Applications, Dijon France (1985), also in Lecture Notes on Computer Science No. 202, Rewriting Techniques and Applications, pp 407–416, Springer-Verlag, New York (1985).
15. Paul W. Purdom, Jr. and Cynthia A. Brown, *Tree Matching and Simplification*, Tech. Report 181, Computer Science Dept., Indiana University (1985).