4

Computational Logic: The Unification Computation

J. A. Robinson
College of Liberal Arts
Syracuse University

People who write papers on theorem proving have unfortunately tended to ignore the practical computational questions which arise when a program is to be written to do deduction on an actual machine with real money. The present writer is to his shame an example of this.

Yet if *computational logic** is to develop into a genuinely useful science it is precisely the computational issues which have to be given the most attention. There is already a growing number of applications, within the field of artificial intelligence, of computational logic: in question-answering, program-writing, proving programs correct, robot control, and so on. These applications are exposing the sadly weak deductive power of current deduction algorithms – at any rate of those realizations of them which have been in use.

One possible way to improve matters is for everyone in the field to bring out into the open forum his ideas on how programs should be written, with specific proposals on such 'hard engineering' details as representation of data objects, layout of store, and the like. At least, such public discussion will, one would hope, eventually unearth the best techniques which actually exist and are in use somewhere, for implementing the various subprocesses involved in mechanical deduction. It is of course also to be hoped that open discussion will stimulate the discovery of new and better techniques. We certainly want to advance the art; but at present we appear to have only the dimmest notion of what the state of the art is.

In the spirit of these observations, this paper describes and discusses a program for computing unification substitutions. The unification computation occurs at the very heart of most modern deduction algorithms. Whatever else goes on during a deduction calculation, an *enormous* amount of unification computation has to be done. It is the 'addition and multiplication' of

* Surely a better phrase than 'theorem proving', for the branch of artificial intelligence which deals with how to make machines do deduction efficiently?

deduction work. There is accordingly a very strong incentive to design the last possible ounce of efficiency into a unification program. The incentive is very much the same as that for seeking maximally efficient realizations of the elementary arithmetic operations in numerical computing – and the problem is every bit as interesting.

Whether the program about to be described is as efficient as it is possible to be, is doubtful. It is very hard even to give a precise meaning to claims of efficiency in computing, let alone to demonstrate their truth. Nevertheless the author thinks the program is very close to maximal efficiency, and offers it as a challenge. He will be delighted to be defeated.

The unification algorithm has as *input* a finite collection $P=\{P_1, \ldots, P_n\}$ of finite sets of expressions. Each finite set P_i is made up of either *terms* or *atoms* (in the sense of the first-order predicate calculus). Each expression therefore *either* (a) consists of a (relation- or function-) *symbol* and a list of terms as *arguments*, the length of the list (which might be zero) being the 'arity' of the symbol; *or* (b) is a *variable*. For example, the following are expressions, written in the conventional fashion:

$$f(x, g(x, y))$$
$$P(k(h(a, b)), f(a, g(x, y))).$$

The *object* of the unification algorithm is to calculate (if one exists) a substitution θ which 'unifies' P, or to indicate (if such be the case) that there is no substitution which unifies P.

When we say that θ unifies P we mean that, for each i, $i=1, \ldots, n$, θ turns each expression in P_i into the *same* expression; or what is the same, that $P_i\theta$ is a singleton. A substitution is an operation which can be performed on expressions (and, by an 'abuse of language', as above, on *sets* of expressions) whereby every occurrence within an expression of each of a given list of distinct *variables* $(x_1, \ldots, x_n)$ is replaced by the corresponding member of an equally long given list of (not necessarily distinct) *terms* $(t_1, \ldots, t_n)$. One writes $E\theta$ for the result of performing the substitution θ on the expression E, and if S is a set of expressions, then $S\theta$ is the set: $\{E\theta \mid E \text{ in } S\}$.

It is usual to write $\{t_1/x_1, \ldots, t_n/x_n\}$ for the substitution in which each occurrence of x_i is replaced by one of t_i. The number n is the substitution's *size*.

For example, if θ is $\{p(r(a))/a, p(a)/x, g(a, b)/y\}$ then:

$$f(x, g(x, y))\theta = f(p(a), g(p(a), g(a, b))).$$

As an example of the phenomenon of unification, consider the system $P=\{P_1, P_2\}$ where:

$$P_1=\{f(x, g(x, y)), z\}$$
$$P_2=\{h(z, y), h(f(a, b), f(d, c))\}.$$

The substitution:

$$\theta=\{x/a, f(d, c)/y, f(x, g(x, f(d, c)))/z, g(x, f(d, c))/b\}$$

unifies P. Indeed, we have:

$$P_1\theta = \{f(x, g(x, f(d, c)))\}$$
$$P_2\theta = \{h(f(x, g(x, f(d, c))), f(d, c))\}.$$

On the other hand, the system $P = \{P_1\}$, where

$$P_1 = \{Q(x, f(x)), Q(f(x), x)\}$$

cannot be unified by any substitution.

The theory of unification is discussed in Robinson (1965), where the unification algorithm is given and proved correct. Here we shall be concerned only with the practical implementation of the abstract algorithm.

The idea of the abstract algorithm is to attempt to 'shrink' the given system P to a set of singletons, by means of successive substitutions of size one. Each substitution 'equates' two corresponding subexpressions which must be made identical if P is to be unified. The algorithm has to locate these two subexpressions, and then perform the necessary substitution, each time through its cycle. The following description sets forth the abstract algorithm in a systematic way.

Step 1. Given $P = \{P_1, \ldots, P_n\}$ as input, set

$$j = 0,$$
$$\theta_0 = \varepsilon \text{ (the identity substitution)},$$

and go to step 2.

Step 2. (for $j \geqslant 0$): if $P_i\theta_j$ is a singleton for each i, $i = 1, \ldots, n$, STOP, with $\theta = \theta_j$ as output. Otherwise, go to step 3.

Step 3. Let k be the earliest number $\leqslant n$ such that $P_k\theta_j$ is *not* a singleton. Let E, F be any two expressions in $P_k\theta_j$. Scan E and F in parallel, from left to right, and locate the leftmost position in which E and F do not have identical symbols. Let e and f be the subexpressions respectively of E and F which begin in that position. If neither e nor f is a variable, STOP. Otherwise, go to step 4.

Step 4. If one of e, f is a variable which is properly contained in the other, STOP. Otherwise, go to step 5.

Step 5. Choose x and t so that $\{x, t\} = \{e, f\}$. Set $\theta_{j+1} = \theta_j\{t/x\}$, add 1 to j, and go to step 2.

End.

If the algorithm stops in either 3 or 4, the given P cannot be unified.

Since, for each k, $1 \leqslant k \leqslant n$,

$$P_k\theta_{j+1} = (P_k\theta_j)\{t/x\}$$

we can, at the end of each cycle, update the sets

$$P_1\theta_j, \ldots, P_n\theta_j$$

simply by performing the substitution $\{t/x\}$ on them. This means *locating* each occurrence of x in each expression, and *replacing* it by an occurrence of t. It might seem that the amount of work required to do this would be proportional to the size of t and the number of occurrences of x. As we shall see,

however, the amount of work required can be limited to a very small constant, independent both of the number of occurrences of x and of the size of t.

In designing an actual realization of the unification algorithm the first question which arises is how best to represent the expressions in the computer store. We want to minimize the amount of scanning, copying, and rearranging which will have to be done during the computation. It must be as simple and fast a process as possible to detect differences between expressions. A representation using *strings* of symbols seems out of the question. The applicative structure of an expression would then have to be recomputed (essentially by parsing it) each time it was referred to. Some list- or tree-oriented organization is wanted, so that we can immediately access the immediate subexpressions of an expression. On the other hand, if we use a general-purpose list-processing framework of representation then we incur a certain amount of superfluous structure and unwanted overhead. For example, to locate the ith element of a list we have to 'chain' our way to it from the start of the list. The representation we use should be as 'close to the machine' as possible, so that if necessary the algorithm can be written directly in machine language yet not become unduly complex.

Accordingly, the following representation is used.

We have a collection of tables: *symbol*, *args*, *vble*, *arity*, *subst*, *term*, *equals*, and *part*, each of which has the same number N of rows. The number N is the number of expressions in the set we are representing. All the tables have one column, with the exception of *args*; and *args* has as many columns as the arity of that symbol, in the expressions being represented, whose arity is largest.

The way in which expressions are represented by the tables is best explained by defining the function *expression*. Intuitively, each row of the tables represents an expression, and in particular row i represents $expression(i)$. The definition is:

$$expression(i) = \textbf{if } subst(i) \textbf{ then } expression(term(i)) \textbf{ else } symbol(i) \mid arglist(i, arity(i))$$

where:

$$arglist(i, j) = \textbf{if } j = 0 \textbf{ then } empty \textbf{ else } arglist(i, j-1) \mid expression(args(i, j)).$$

(The vertical stroke denotes concatenation, and *empty* is the empty expression.)

The arrays *vble* and *subst* are boolean arrays. If $subst(i) = F$ for all i, $1 \leqslant i \leqslant N$, the representation is said to be 'normal'. We always arrange matters so that *in a normal representation, distinct rows represent distinct expressions*, i.e., so that

$$\textbf{if } expression(i) = expression(j) \textbf{ then } i = j.$$

By ignoring *subst*, we accordingly get what is called the 'normal' reading of the tables, in which row i represents the expression:

$$normalexpression(i) = symbol(i) | normalarglist(i, arity(i))$$

where:

$$normalarglist(i,j) = \textbf{if } j=0 \textbf{ then } empty \textbf{ else } normalarglist(i,j-1) | normalexpression(args(i,j))$$

Thus a normal representation is one which satisfies:

$$expression(i) = normalexpression(i).$$

The system which is to be unified is represented in the array *part*. The method is this: $part(i)$ is an integer satisfying:

$$1 \leqslant part(i) \leqslant i$$

and having the meaning that $normalexpression(i)$ is to be made identical with $normalexpression(part(i))$ by the substitution which is to be computed. If $part(i) = i$ then of course this is vacuously accomplished.

	symbol	*args*	*arity*	*vble*	*subst*	*term*	*equals*	*part*	*N*
1	*f*	2 3	2	*F*	*F*		1	1	13
2	*x*		0	*T*	*F*		2	2	
3	*g*	2 4	2	*F*	*F*		3	3	
4	*y*		0	*T*	*F*		4	4	
5	*z*		0	*T*	*F*		5	1	
6	*h*	5 4	2	*F*	*F*		6	6	
7	*h*	8 11	2	*F*	*F*		7	6	
8	*f*	9 10	2	*F*	*F*		8	8	
9	*a*		0	*T*	*F*		9	9	
10	*b*		0	*T*	*F*		10	10	
11	*f*	12 13	2	*F*	*F*		11	11	
12	*d*		0	*T*	*F*		12	12	
13	*c*		0	*T*	*F*		13	13	

Figure 1

During the computation, the array *equals* is modified in such a way that it always has the meaning:

$$\textbf{if } equals(i) = equals(j) \textbf{ then } expression(i) = expression(j).$$

Note that, in general, in order to determine whether $expression(i) = expression(j)$, it is necessary to 'scan the two expressions in parallel' by recursively evaluating *expression* at i and at j; whereas it can be determined whether $equals(i) = equals(j)$ by two read-references to the tables, and one comparison.

It follows that in a normal representation, all entries of *equals* must be distinct. We adopt the convention that the 'normal' setting of *equals* is given by:

$$equals(i) = i.$$

To illustrate these ideas, we give in figure 1 the tables comprising the normal representation of the input for the example given earlier, in which the system to be unified is:

$P_1 = \{f(x, g(x, y)), z\}$,
$P_2 = \{h(z, y), h(f(a, b), f(d, c))\}$.

In a normal representation, the entries in *term* are irrelevant, and so are left blank in the figure.

We are now ready to explain the computation. It is carried out by executing the function *unify*. This function yields T or F as result, according as the system represented in the tables is or is not unifiable. In the former case, the tables are modified as a side-effect of the computation. Where $subst(i) = T$, it is then $expression(term(i))$ which is to be substituted for the variable $normalexpression(i)$. The function *unify* itself is very simple. It consists of computing $equate(j, part(j))$ for each row j:

function *unify*; **vars** j; $1 \rightarrow \mathbf{j}$;
A: **if** $j > N$ **then** *true*
elseif $part(j) = j$ **then** $j+1 \rightarrow j$; **goto** A
elseif $equate(j, part(j))$ **then** $j+1 \rightarrow j$; **goto** A
else *false* **close end**;

The function *equate* has either T or F as its value, and it has an important side-effect. If the value of $equate(i, j)$ is T, then in the course of its being evaluated the tables will have been so modified that, after evaluation is completed, $expression(i) = expression(j)$. In addition, *equals* will have been modified as explained earlier. The function *equate* is defined as follows:

function *equate* $i\,j$;
A: **if** $equals(i) = equals(j)$ **then** *true*
elseif $subst(i)$ **then** $term(i) \rightarrow i$; **goto** A
elseif $subst(j)$ **then** $term(j) \rightarrow j$; **goto** A
elseif $symbol(i) = symbol(j)$ **then** $equateargs(i, j)$
elseif $vble(i)$ **then**
    if $occur(i, j)$ **then** *false* **else**
    $true \rightarrow subst(i)$; $j \rightarrow term(i)$; $identify(i, j)$; *true* **close**
elseif $vble(j)$ **then**
    if $occur(j, i)$ **then** *false* **else**
    $true \rightarrow subst(j)$; $i \rightarrow term(j)$; $identify(i, j)$; *true* **close**
else *false* **close end**;

This function is the heart of the program. Its auxiliary functions are defined as follows:

function *equateargs* $i\,j$; **vars** k; $arity(i) \rightarrow k$;
A: **if** $k = 0$ **then** $identify(i, j)$; *true*
elseif $equate(args(i, k), args(j, k))$ **then** $k-1 \rightarrow k$; **goto** A
else *false* **close end**;
function *occur* $i\,j$;
A: **if** $subst(j)$ **then** $term(j) \rightarrow j$; **goto** A
elseif $arity(j) = 0$ **then** $i = j$

```
else occurinargs(i,j) close end;
function occurinargs i j; vars k; arity(j)→k;
A: if k=0 then false
elseif occur(i, args(j, k)) then true
else k−1→k; goto A close end;
function identify i j; vars k e f;
equals(j)→e; equals(i)→f; 1→k;
if e=f then exit;
A: if k>N then exit;
if equals(k)=e then f→equals(k);
k+1→k; goto A close end;
```

We have written these function programs in POP-2 in the expectation that readers of these Workshop volumes will readily understand them, and can run them, exactly as given here, if they have access to a POP-2 system. If not, the routines are quite simple and transparent enough to write out in another programming language.

Execution of *unify* with the store arranged as in figure 1 produces the value *true* and leaves *subst*, *term* and *equals* as shown in figure 2. The other tables are 'read only', and remain unchanged. The entries shown in parenthesis are the original entries, unchanged during the computation. Thus we see that only rows 4, 5, 9, and 10 of *subst* and *term* are changed during the computation: and *they are changed only once.* In *equals* it is rows 1, 2, 3, 5, 6, and 11 which are changed during the computation. In general, the entries in *equals* may be changed several times before reaching their final values.

	subst	*term*	*equals*
1	(*F*)		8
2	(*F*)		9
3	(*F*)		10
4	*T*	11	(4)
5	*T*	1	8
6	(*F*)		7
7	(*F*)		(7)
8	(*F*)		(8)
9	*T*	2	(9)
10	*T*	3	(10)
11	(*F*)		4
12	(*F*)		(12)
13	(*F*)		(13)

Figure 2

The *number of entries* in *subst* which are changed to *T* during the computation – four, in our example – is the same as the *number of cycles* of the abstract unification algorithm needed for the problem. Each entry corresponds to the making of a substitution $\{t/x\}$ throughout the entire set of expressions – indeed x is *symbol*(i) and t is *expression*(*term*(i)), where i

is the number of the row in which the entries to *subst* and *term* are made. Thus we see that, to represent the result of making the substitution $\{t/x\}$ throughout *all* the expressions, we make only *two* write-references to the tables, *regardless of the complexity of the term t and of the number of occurrences of the variable x in the various expressions in the set.* This feature is very important for the speed of the program.

It will be seen that the final state of the tables *subst*, *term*, together with the remaining tables, as given in figures 1 and 2, represents the set of expressions into which the computed substitution transforms the original expressions. The transformation is summed up in figure 3, which shows tables of values of the functions *normalexpression* and *expression*, computed from the final representation.

i	*normalexpression* (i)	*expression* (i)
1	$f(x, g(x, y))$	$f(x, g(x, f(d, c)))$
2	x	x
3	$g(x, y)$	$g(x, f(d, c))$
4	y	$f(d, c)$
5	z	$f(x, g(x, f(d, c)))$
6	$h(z, y)$	$h(f(x, g(x, f(d, c))), f(d, c))$
7	$h(f(a, b), f(d, c))$	$h(f(x, g(x, f(d, c))), f(d, c))$
8	$f(a, b)$	$f(x, g(x, f(d, c)))$
9	a	x
10	b	$g(x, f(d, c))$
11	$f(d, c)$	$f(d, c)$
12	d	d
13	c	c

Figure 3

To reset the tables to normal after a computation it suffices to execute

$$true \rightarrow subst(i);\ i \rightarrow equals(i);$$

for $i = 1, \ldots, N$. This extremely fast reset feature is considerably important in situations where *unify* is being executed repeatedly for a series of distinct partitions of the same set of expressions – a very common situation in applications.

A somewhat subtle feature of the method of representation used here involves the treatment of variables. It turns out that two occurrences of variables are occurrences of the *same* variable if, *and only if*, they are represented by pointers to the same row. It is *not* the case that if we have, e.g., $symbol(i) = $ 'x' and $symbol(j) = $ 'x', with $i \neq j$, then an 'occurrence of x' is equally well represented by putting an i or a j in the appropriate *args* cell. No; in fact we would here have *two* variables, not one, despite the coincidence of their written appearances. This is simply the well-known phenomenon of 'bound' or 'apparent' variables. In the formulas $\forall x P(x)$ and $\forall x Q(x)$, for example, the 'x' of the first formula is a different variable from the 'x' of the second formula.

	symbol	*args*	*arity*	*vble*
1	*P*	2 3 4	3	*F*
2	*x*		0	*T*
3	*y*		0	*T*
4	*u*		0	*T*
5	*P*	3 6 7	3	*F*
6	*z*		0	*T*
7	*v*		0	*T*
8	*P*	2 7 9	3	*F*
9	*w*		0	*T*
10	*P*	4 6 9	3	*F*
11	*P*	12 13 14	3	*F*
12	*x*		0	*T*
13	*y*		0	*T*
14	*u*		0	*T*
15	*P*	13 16 17	3	*F*
16	*z*		0	*T*
17	*v*		0	*T*
18	*P*	14 16 19	3	*F*
19	*w*		0	*T*
20	*P*	12 17 19	3	*F*
21	*P*	22 23 24	3	*F*
22	*g*	23 24	2	*F*
23	*r*		0	*T*
24	*s*		0	*T*
25	*P*	26 27 28	3	*F*
26	*a*		0	*T*
27	*h*	26 28	2	*F*
28	*b*		0	*T*
29	*P*	30 31 30	3	*F*
30	*k*	31	1	*F*
31	*t*		0	*T*

	clause	*length*
1	1 5 8 10	4
2	11 15 18 20	4
3	21	1
4	25	1
5	29	1

	sign
1	*F F F T*
2	*F F F T*
3	*T*
4	*T*
5	*F*

M 5

N 31

Figure 4

This feature of our system of representation is quite crucial for the overall efficiency of a resolution-based computational logic. For in such a case we have to deal with collections of atoms grouped together into so-called *clauses* (and afflicted each with a *sign* – positive or negative – representing the affirmation or denial of the atom). These groups are thought of as the universally quantified disjunctions of their members. It is vital that, when two clauses are to be resolved, no variable occurs in *both* clauses – i.e., that each clause have its 'own' variables. In programs employing resolution of clauses as an inference principle, it is usually necessary to introduce a step in the computation of 'standardization' of variables, in order to make sure, prior to resolving two clauses, that their variables are 'separated', in this sense.

In our system, this step is completely unnecessary. The different clauses *automatically* have their 'own' variables by virtue of the fact that we never

put (in the *normal* representation) the same atom *in more than one clause*. To illustrate this point, we give in figure 4 the tables representing the clauses:

$$\bar{P}(x, y, u)\bar{P}(y, z, v)\bar{P}(x, v, w)P(u, z, w) \quad (1)$$
$$\bar{P}(x, y, u)\bar{P}(y, z, v)\bar{P}(u, z, w)P(x, v, w) \quad (2)$$
$$P(g(r, s), r, s) \quad (3)$$
$$P(a, h(a, b), b) \quad (4)$$
$$\bar{P}(k(t), t, k(t)). \quad (5)$$

Although clauses (1) and (2) contain apparently similar atoms, this is *only* apparent. In the representation, because of the way in which the variables are treated, the clauses (1) and (2) have no variables in common, even though they *do* have variables whose 'print-names' are visually indistinguishable.

The table representation will enjoy this useful 'separation of variables' property if we make each *new* clause, as it is added to those already represented in the tables, 'occupy' its own new set of rows. For instance, in the representation shown in figure 4, clause (1) occupies rows 1 to 10; clause (2) occupies rows 11 to 20; clause (3) occupies rows 21 to 24; clause (4) occupies rows 25 to 28; and clause (5) occupies rows 29 to 31.

In a sequel to this paper, a further program will be presented which extends the ideas given here to one higher level in a hierarchy of programs, the top level of which is a complete proof procedure for the predicate calculus of first order with equality. The next level program accepts as input a set of tables like those in figure 4, and detects a unifiable partition P (if one exists) modulo which the given clauses are contradictory (unsatisfiable in the 'ground' sense). If no such P exists, the program detects this fact and reports it. This computation is exceedingly interesting, and is a finite combinatorial process whose efficiency can be made far greater than previous experience with it may have suggested.

Acknowledgements

I would like to express my gratitude for the helpful suggestions and comments from my colleagues Pat Hayes, Isobel Smith, Bruce Anderson and Rod Burstall of the University of Edinburgh, and John Reynolds of Syracuse University, concerning the topic of this paper. I should also like to thank the Science Research Council for financial support.

REFERENCE

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *J. Ass. comput. Mach.*, **12**, 23–41.