8-1978

ON LIST STRUCTURES AND THEIR USE IN THE PROGRAMMING OF UNIFICATION

F. Lockwood Morris

Syracuse University, lockwood@ecs.syr.edu

4-78

ON LIST STRUCTURES
AND THEIR USE IN THE PROGRAMMING OF UNIFICATION

F. LOCKWOOD MORRIS

AUGUST 1978

SCHOOL OF COMPUTER
AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

On List Structures and Their Use in the Programming of Unification

F. Lockwood Morris
Syracuse University

Abstract: The notion of list structure is discussed, and a new construct is introduced into LISP which permits the computation of list structures containing cycles without recourse to operations which alter existing structures. It is shown that list structures can be used to represent both finite and "rational" infinite terms. Substitutions (generalized for a term algebra which includes infinite terms) are discussed, "tables" are introduced as an abstract data type, and two methods of representing substitutions by tables, together with their interrelation, are considered. Concise programs are given for a succession of forms of Robinson's unification algorithm, including one which operates in almost linear time, and for the application of substitutions to terms.

Key words and phrases: List structures, list processing, record processing, LISP, recursive definition of data structures, substitution, unification, fast unification.

CR Categories: 4.22, 4.34, 5.21, 5.25, 5.39.

This research was supported by NSF Grant MCS75-22002.

Author's present address: School of Computer and Information Science,
Syracuse University, Syracuse, New York 13210.

On List Structures and Their Use in the Programming of Unification

1.0 INTRODUCTION

To date, most published descriptions of algorithms of theoretical interest have been essentially imperative in style and much concerned with operations which change data structures, possibly for the reason that considerations of efficiency seem to force the use of a language (even for very high-level descriptions) which is closely modeled on the primitive operations of the random-access digital computer. This paper will try to suggest that such need not be the case, by using a non-imperative programming notation to develop the unification algorithm of J.A.Robinson [13,15] through a variety of forms, culminating in a realization of the "almost linear" algorithm discovered by Huet and Kahn [7] and by Robinson [14].

Section 2 is devoted to definitions and to the laying of a conceptual groundwork. It is argued that the avoidance of operations which alter existing linked structures facilitates the conceptualization of entire such structures as timeless entities, which in turn may serve to narrow the intellectual gap between an abstract algorithm and its realization as a program. (Hoare [6] seems to argue for a similar view of complex data structures, although he is led to disparage the class of structures containing cycles which will feature prominently here.)

In Section 3 the programming notation used in this paper is introduced. It is in fact the non-imperative subset of a dialect of Lisp presently under development at Syracuse by the author and his colleagues. The principal novelty of this dialect is a provision for the computation of list structures as solutions to recursive equations, and hence for the creation, without recourse to assignment, of structures containing cycles. Such recursive definitions have been proposed several times, e.g. by Burge [3] and Landin [9], but an implementation technique is outlined here which is believed to be new. The list structures whose creation such definitions allow, which would appear as infinite expressions according to the usual Lisp "S-expression" notation, are perhaps suitably called unbounded, by analogy with the "finite, but unbounded" universe of Riemanian geometry.

In section 4 "tables" are introduced - a family of abstract data types encapsulating the notion of a tabulated function or predicate. Using this notion, the simplest non-primitive operations for unbounded list structures - testing for equality and copying - are programmed.

In Section 5 elementary (exponential-time) programs for unification are developed. It is convenient to discuss in parallel classical unification and *unrestricted unification*, which considers a variable to be unifiable with a term containing it. The infinite terms which may arise by unrestricted unification receive natural representations as unbounded list structures. There are seen to be advantages in both space and time to approaching classical unification as the addition of a separate cycle-detection phase to unrestricted unification.

In Section 6 the (textually slight) changes to the programs of the preceding section are introduced which yield a realization of the Huet-Kahn-Robinson algorithm.

2.0 *ONTOLOGY AND NOMENCLATURE*

Recall that the elementary objects of Lisp are divided into *atoms* and *pairs* (the latter sometimes called "dotted pairs".) The elementary operations are *cons*, which given any two objects as arguments has as result a pair with first component the first of these and second component the second; *car* and *cdr*, which retrieve from any pair its first or second component respectively; *atom*, a predicate true of atoms and false of pairs; and *eq*, which we shall take here to be the identity relation on objects. (*Eq* is universally so implemented, although it was originally specified as a relation only on atoms.) Lisp does not make a distinction of "L-value" *vs* "R-value" (location *vs* contents); if one wishes to speak in that way one should say that in Lisp the L-values have taken over, so that a Lisp object may be thought of either as an instance of a record in memory or as the address quantity which locates it.

Rather than confining attention to individual pairs, one may contemplate the complex of all objects accessible from a given one by chains of zero or more car's and cdr's as far as these are defined - i.e., do not entail taking car or cdr of an atom. Such a complex we call here a *list structure* (for reasons of tradition and euphony, not because any notion of one-dimensional list is involved.)

Most existing versions of Lisp have two additional primitives, *rplaca* and *rplacd*. for altering the components of a pair object. There is, however, good reason for doing without them; namely, in the absence (which we shall take for granted from here on) of pair-altering primitives, *list structures are incorruptible* - e.g., if a variable is assigned a list structure, it will continue to refer to that very list structure until reassigned. It is just this incorruptibility that makes the act of abstraction which regards an entire list structure as a single entity performable without precaution. Properties of list processing programs can then be proved by reasoning about list structures in the same sense that properties of Algol programs are proved by reasoning about numbers and truth values. One might argue that this provision of complex entities is the chief high-level property of Lisp.

List structures are not, in fact, a very abstract notion; by use of the predicate *eq*, distinct but isomorphic structures can be distinguished, and moreover the precise isomorphism class of a given structure as a rooted directed graph is determinable. Put more colloquially, the "sharing" pattern of substructures is significant. Both confluences and cycles may exist; that is, distinct sequences of car's and cdr's may lead to one structure, and a non-empty sequence of car's and cdr's may lead from a structure to itself. The curious situation that there is no evident way of creating list structures with cycles will be dealt with in the next section. We shall not, however, introduce any means of creating, or consider as possibly existing, list structures which contain an infinite number of distinct pairs or atoms.

A further abstraction is often fruitful: the identification of all list structures which cannot be distinguished without the use of *eq* on pairs. We shall call the entities arising from this identification *reduced list structures*, and give the name *equal* to the corresponding equivalence relation on list structures. Stated positively, two list structures are equal if and only if every finite composition of *car* and *cdr* which leads to an atom in either leads also to the same atom in the other. (This definition of *equal* generalizes the predicate traditionally so called in Lisp, which is defined for acyclic structures only.)

Regarded as an operation on reduced list structures, *cons* is a univocal function. Evidently one may think of reduced list structures as ordered binary trees whose leaves are atoms and whose interior nodes are unlabelled, and which may (on account of cycles in the corresponding list structures) have infinite paths. However, because of the finiteness of list structures, we do not get all such trees but only those which have finitely many distinct subtrees.

We next turn to the definition of *terms*, the abstract entities with which the unification algorithm is concerned. Given a fixed stock of *variables* and a *signature* (the assignment of a non-negative *arity* to each of a stock of *operators*), the corresponding algebra, T, of (finite and infinite) terms may be abstractly characterized as the maximal system enjoying the following properties:

1. (Closure) T is an algebra with the given signature and containing the given variables.

2. (Unique decomposition) Every term either is a variable or else is a composite f(tl, ... ,tk) for exactly one choice of operator f and subterms tl, ... ,tk.

3. (Identity by finite paths) Two terms are the same if and only if every well-defined path of decomposition in either leads to the same variable in both or to composite subterms with the same operator in both.

A term which has only finitely many distinct subterms we will call rational. (The analogy here is with rational numbers regarded as terminating or periodically infinite decimals. One can imagine that, by some elaborate generalization of the vinculum notation for repeating decimals, any rational term might be written down in a finite number of symbols.)

To represent terms by list structures, we may follow the familiar plan of choosing a distinct atom, not NIL, to represent each operator and each variable, and representing a composite f(tl, ... ,tk) by a list of length k+l:

(f* tl* ... tk*)

- actually this is the structure

cons[f*,cons[tl*, ... cons[tk*,NIL] ...]] -

where f*, tl*, ... ,tk* represent f, tl, ... ,tk. One observes that what this scheme provides is a unique representation of every rational term by a reduced list structure, and thus a non-unique way of representing terms by list structures; such list structures we will call term structures. We will suppose that there is a fixed Lisp predicate, varp, which distinguishes atoms representing variables from all other objects. (Reduced) list structures which contain only finite paths - in particular the term structures which represent finite terms - we will call bounded.

Finally, we define a substitution to be an endomorphism of T, and note that any function from variables to terms can be extended to a unique substitution. (See [5] for an explicit construction of algebras similar to T and for the proof of this property.) We shall be dealing here exclusively with finite sets of rational terms and, as a consequence, exclusively with substitutions which are the identity on all but finitely many variables.

3.0 NOTATION

Our programming notation in this paper is an informal and condensed representation of Lisp, called "M-notation" by McCarthy et.al.[10] in contrast to the machine-acceptable "S-notation". Identifiers are written in lower case. Lisp atoms (strings of upper-case characters) stand for themselves - thus M-notation has X where S-notation has (QUOTE X). We show function application with square brackets, separating multiple arguments with commas as in f[x,y]. We indicate the conditional expression by the keywords if - then - else, write the truth values as true, false, symbolize logical negation by ¬, and use infixed operators ∧ , ∨ as shorthand for the evident conditionals, e.g. x ∨ y for if x then true else y, so that a conjunction or disjunction may be well-defined even though its right-hand argument is not.

Applications of the primitives car and cdr are indicated by prefixed operators a and d; applications of eq by infixed =. The predicate null is definable:

null[x] = x=NIL,

NIL being the particular atom conventionally used as list terminator, but for convenience application of null is indicated by an operator: prefixed n.

To show applications of cons we use the punctuation of S-notation - round parentheses and the dot - in template fashion, so that (x.y) means cons[x,y], and - in conformity with Lisp conventions for lists - (x y z w) means the same as (x.(y.(z.(w.NIL)))).

Functions of zero or more arguments may be denoted by lambda-abstraction:

λ[x1, ... ,xn]e

where the xi are identifiers and e is any expression.

We shall use two additional variable-binding notations which, unlike lambda-abstraction, ascribe values to the identifiers introduced:

```
{let    }
{letrec } x1 = a1
      and x2 = a2
      and ...
in b .
```

The construct with let is merely a convenient abbreviation for the application:

[λ[x1, ... ,xn]b] [a1, ... ,an] .

(Indeed by using let, and by exploiting the further convention of writing function definitions in the familiar "dummy argument" form:

f[x,y] = a

rather than the strict

f = λ[x,y]a

we shall contrive to avoid explicit appearances of lambda-abstraction in what follows.) As the reduction to an application makes clear, the scope of the variables xi is the body b.

The meaning of the corresponding letrec construct, for which the scopes of the xi are both b and all the right-hand-side expressions aj, is not to be disposed of so easily. The purpose of the construct is to provide for the definition of both mutually recursive functions and unbounded list structures. (The implementation of recursive functions is by now well understood, and for simplicity the following discussion will be confined to the problem of list structures.)

Reflection may make it apparent that any prescription to compute each of quantities x1, ..., xn by zero or more applications of cons to any of x1, ..., xn themselves and other independently computed quantities (but without directly circular prescriptions of the form xi = xj = xk = ... = xi) should determine a system of list structures, unique up to isomorphism, which solves it in the sense that if, e.g., it is prescribed that x3 be cons[x5,cons[NIL,x3]] then the resulting structures are such that ax3=x5, adx3=NIL, and ddx3=x3. (To be precise, for uniqueness up to isomorphism rather than merely up to equality we must additionally require that the prescribed cons's all give rise to distinct pairs.)

It is not immediately clear how this attempt to smuggle rudiments of least-fixed-point finding into a language which has imperative aspects, and in which the order and multiplicity of evaluation of expressions are therefore significant, is to be made to work. The approach which does least violence to ordinary notions of program execution seems to be to lay down that a1, ..., an and b are to be evaluated once each - the value obtained for b being that yielded by the whole letrec construct - by the ordinary evaluation mechanism. This is to say that the ai need not be restricted to any particular syntactic form, and that the calls of cons they cause to be made are not treated in any special way. It follows that an identifier xi, even before the value of its corresponding ai has been obtained, must if evaluated yield an object which, however deficient in properties, is amenable at least to cons (and presumably to eq, if we take seriously our characterization of that predicate as "identity of objects").

If one supposes that the realization of objects in the machine is such that objects lacking all properties save identity can be created, and that moreover these can fuse with other objects (i.e. that under certain circumstances distinct objects having no conflicting properties can come to be one) then a straightforward semantics for letrec is evident: The identifiers x1, ..., xn are at first associated with n such ghostly objects; the expressions a1, ..., an are then evaluated and the object resulting for each ai fused with the value associated to the corresponding xi; and the final state of these associations is used to provide the environment for the evaluation of the body b.

The writing of useful programs is greatly facilitated by specifically ruling, in addition to what is set down above, that a1, ..., an are to be evaluated in the order written, and that each fusion is to be performed immediately its right-hand-side value is obtained. This means that during, for example, the evaluation of a2, the internal structure of x1 is well-defined and accessible to inspection except at those points where it incorporates any of x2, ..., xn.

The uses of **letrec** which appear in the remainder of this paper may be understood in the light of the above account (including the rule of left-to-right evaluation), and the reader not specifically interested in problems of implementation is invited to skip to the last paragraph of this section. It appears that a system of fusible objects as described above can be realized, by building in the Fischer-Galler algorithm - described in Section 6 below - as the implementation of *eq*.

However, it is of interest to see what can be done within the conventional implementation of Lisp objects, for which an object is just a record instance in the memory, and *eq* is comparison of record addresses. In this case, objects are not fusible; moreover, pairs may be allocated records of a different size or in a separate area of memory from atoms, so that to be, even as a record with undefined fields, is either to be a pair or to be an atom.

An approximation to the above evaluation scheme is still possible, and, with some care, the programmer can avoid provoking the mistakes to which it is subject. It seems not to be difficult in practice to arrange the computations one would like to make with **letrec** in such a way that any value of an identifier which must be called into existence before it has been computed in the ordinary sense turns out in the end to be a dotted pair - this is not surprising, since any cycle in a list structure consists of pairs - and moreover a pair which has been created solely to be the value of that identifier, rather than being part of a previously existing and independently accessible list structure. For programs which have been so arranged, the following evaluation method will work: Associate to any xi whose value is needed before that of ai has been obtained a new pair object with undefined *car* and *cdr* components. When the value - call it pi - of ai is at hand, and if xi is found to be associated to such a pair (so that a fusion should be performed), instead copy the *car* and *cdr* components of pi into the object which is the value of xi. If pi is in fact an atom, then an error must be reported. Even worse, if there exists any other means of access to pi, then the system has erroneously come to contain two accessible and distinct (although *equal*) objects which should be one; it is not feasible in Lisp to detect the commission of this form of error, any more than it is feasible in general to reclaim unused storage except by garbage collection.

All this sounds very unsatisfactory, but there is a sufficient syntactic condition which will ensure that neither form of error occurs: that each of the expressions a2, ..., an should have *cons* as main connective. Programs that observe the restrictions of the preceeding paragraph - as do all those to be presented in this paper - may readily be transcribed into any conventional language, such as Pascal, which provides pointers and records with assignable fields.

It is hoped to discuss *letrec* and these implementation techniques (which generalize naturally to permit application of *car* and *cdr* to objects whose components are not yet defined, resulting in the creation of trees of pairs which grow from the top down) more fully elsewhere.

We shall suppose one further augmentation to Lisp: the germ of an exception-handling facility in the shape of a "function" named *error* (actually provided by many existing Lisp systems) which aborts any computation calling it. Our purposes here do not require us to consider the problem of integrating exception-handling with the other control structures of the language.

4.0 TABLES

The programs to be presented in this paper will need to use data structures representing tabulated functions, relations, and sets. (We shall use "table" as a catchall term covering all of these.) Many efficient implementations of these notions are known, and we want to avoid here getting enmeshed in the details of any one of them. Plainly the right way of proceeding is to specify abstract data types, and to impose just the necessary restrictions on the behavior of the primitive operations which act on them. This might in principle best be done by the axiomatic method; however for these particular notions inefficient (linear access time) representations by lists are so simple that we shall instead give straightforward exemplary definitions of functions to manipulate the list representations.

For tabulated functions, then, we need means of creating an empty table, of adding an argument-value pair, of discovering if an argument appears in a table, and of producing the value corresponding to a given argument. Representing the table by a list of its argument-value pairs, we are led to the definitions:

$$\begin{array}{l}
\mathrm{nila}[\,] = \mathrm{NIL} \\
\mathrm{extend}[x,y,a] = ((x.y).a) \\
\mathrm{defined}[x,a] = \neg \mathbf{n}a \wedge [\mathbf{a}a \underline{=} x \vee \mathrm{defined}[x,\mathbf{d}a]] \\
\mathrm{assoc}[x,a] = \mathbf{if}\ \mathbf{aa}a \underline{=} x\ \mathbf{then}\ \mathbf{da}a\ \mathbf{else}\ \mathrm{assoc}[x,\mathbf{d}a]\ .
\end{array}$$

(What we have here is of course the data structure known to Lisp tradition as an "association list". Accessing functions generally used in practice with association lists, and indeed with many other implementations of the notion of tabulated function, adopt some more or less awkward device by which one primitive can do the work of both *defined* and *assoc*. Since our first aim here is clear programs, we prefer to separate them; note that *assoc*[x,a] will produce a result only in case *defined*[x,a] is true.)

What is meant by calling these definitions "exemplary" is that any other quadruple of functions *nila*, *extend*, *defined*, *assoc* is acceptable for which the behavior of *defined* and *assoc* on objects built up by *nila* and *extend* is indistinguishable from that of the versions given here, whatever the actual form of the objects may be.

We shall also require tabular representations of finitely supported binary relations on data objects. This is the same as to say "finite sets of ordered pairs of objects", but note that "ordered pair" here is the extensional concept from mathematics, not the Lisp dotted pair. Suitable primitives are *nilr* (empty relation), *extendr* (include an instance in a relation), and *memr* (does a relation hold between two objects?), which may be given the exemplary definitions

$$\begin{array}{l}
\mathrm{nilr} = \mathrm{nila} \\
\mathrm{extendr} = \mathrm{extend} \\
\mathrm{memr}[x,y,r] = \neg \mathbf{n}r \wedge [[\mathbf{aa}r \underline{=} x \wedge \mathbf{da}r \underline{=} y] \vee \mathrm{memr}[x,y,\mathbf{d}r]]\ .
\end{array}$$

Similarly, for finite sets of single objects we may take the primitives *nils*, *extends*, *mems*, with exemplary definitions

$$\begin{array}{l}
\mathrm{nils} = \mathrm{nila} \\
\mathrm{extends} = \mathrm{cons} \\
\mathrm{mems}[x,s] = \neg \mathbf{n}s \wedge [\mathbf{a}s \underline{=} x \vee \mathrm{mems}[x,\mathbf{d}s]]\ .
\end{array}$$

It is tempting to be a little less exemplary and to ensure against duplicate occurrences in sets and relations by defining, e.g.,

$$\mathrm{extendr}[x,y,r] = \mathbf{if}\ \mathrm{memr}[x,y,r]\ \mathbf{then}\ r\ \mathbf{else}\ ((x.y).r)\ .$$

I am indebted to Professor E.E.Sibert for pointing out that these additional checks, besides being logically unnecessary, would never come into play in the applications developed here.

Observe that tables as considered here, being definable as list structures, are likewise incorruptible entities. It is hoped that the resulting programs will be more transparent than they would be if they relied on the use of data structures subject to side-effects.

As a first example of operating on arbitrary list structures by the help of tables, we consider programming the predicate *equal*. Recall that two list structures are *unequal* if and only if some finite path to an atom in one corresponds to a path to a different atom, or to a non-atom, in the other. But if there exists a sequence of car's and cdr's leading to two such incompatible objects, then there exists one of which all the initial subsequences lead to distinct ordered couples of pair objects in the two structures. (If an ordered couple repeats, then we can get a shorter sequence by deleting the part between its first and second appearances.) Therefore, if we simply amputate any incipient infinite search - by decreeing that any couple of objects encountered for a second time in an equality computation *are* equal - we will ensure termination without invalidating the answers produced. Thus we are led to the program

```
equal[x,y] = equa[x,y,nilr[]]

equa[x,y,r] = if atom[x] ∨ atom[y] then x=y
        else if memr[x,y,r] then true
        else let r' = extendr[x,y,r]
             in equa[ax,ay,r'] ∧ equa[dx,dy,r']
```

Convergence is assured, because the relation r, which increases in size with every recursive call, cannot grow beyond the product of the numbers of dotted pairs present in the two input list structures. It may be very slow, however, for small but tightly knotted structures. In fact, it appears that an equality predicate with running time linear or nearly so in the actual sizes of its input structures requires the same sophisticated equivalence-classing techniques as does (nearly) linear unification.

A second programming example, which will illustrate the use of letrec as well as that of tables, is given by the problem of making an isomorphic copy of a list structure, built out of the same atoms but with all new dotted pairs. Here the observation to make is that the necessary intermediate data structure tabulates a function mapping pairs in the old list structure to their image pairs in the new one, and that **letrec** allows us to create a value of this function out of thin air, before we know the subobjects of which it will ultimately be the *cons*. We define:

```
copy[x] = extract[x,cop[x,nila[]]]

extract[x,h] = if atom[x] then x else assoc[x,h]

cop[x,h] = if atom[x] ∨ defined[x,h] then h
    else letrec h' = cop[dx,cop[ax,extend[x,x',h]]]
         and x' = (extract[ax,h'] . extract[dx,h'])
         in h' .
```

Here *cop*[x,h], the function which performs the depth-first search, may be thought of as computing two results: x', a copy of x, and h', an extension of h. Lisp, in common with most existing languages, regrettably does not provide functions with multiple results. They can be simulated with explicit cons's, car's, and cdr's, but these are likely to distract attention from the productive list manipulation being done. Moreover, a two-result *cop* would not allow of such simple nesting of its recursive calls. It seems to make a neater program, therefore, to observe that x', if not identical with x, is hidden within h', and to provide the function *extract* for recovering x' from h', although this leads to calls of *assoc* which are not strictly necessary.

Equal and *copy* will not be used in the subsequent development of programs for unification; however, their patterns of operation will be discernable within several other functions.

5.0 UNIFICATION

Robinson's most abstract enunciation [15] of his unification algorithm may be summarized as follows so as to be valid for both classical (finite terms only) and unrestricted unification. We shall use upper-case italic letters to range over abstract terms, and shall denote abstract substitutions by small Greek letters, in particular the identity substitution by ε. We write application of substitutions to terms with the substitution on the right, and use the corresponding convention for composition of substitutions, i.e.

$$T(\sigma\circ\tau)\ \underline{=}\mathrm{df}\ (T\sigma)\tau \quad .$$

We shall employ the notation $\{X1 \longmapsto T1, \ldots, Xn \longmapsto Tn\}$ to give explicitly a substitution which maps the named variables as indicated, and is the identity on all others.

To compute, then, a most general unifying substitution of terms A, B if possible:

1. (Initialization): Set $\sigma = \varepsilon$.

2. (Iteration):

    1. If $A\sigma = B\sigma$, then terminate successfully: σ is a most general unifier.

    2. Otherwise, select any pair of unequal homologous subterms X, Y of $A\sigma$, $B\sigma$ which are not applications of the same operator, and if just one of the two is a variable, let X be that one.

    3. If neither is a variable - that is, if X and Y are composite with different operators - then terminate unsuccessfully: a "clash" has been found.

    4. Otherwise, construct ρ such that $\rho = \{X \longmapsto Y\rho\}$, replace σ by $\sigma \circ \rho$, and repeat from 2.1; however,

    5. if there is no such ρ , then terminate unsuccessfully.

We note (but do not prove) the following facts:

1. Selection of ρ can be impossible only if X occurs in Y (otherwise take $X\rho = Y = Y\rho$), and if moreover we disallow infinite terms. If infinite terms <u>are</u> allowed, then by unendingly replacing occurrences of X by Y, starting with X itself, we obtain a term whose substitution for X in either X or Y yields itself.

2. Each ρ introduced eliminates a variable; therefore, if there are only finitely many variables to begin with, the loop can run for only finitely many iterations.

3. Because the substitutions which arise do eliminate the variables which they modify, they (in particular the successive values of σ) are all idempotent, i.e. satisfy $\sigma \circ \sigma = \sigma$.

4. If the algorithm terminates successfully, the final value of σ has the defining properties of a most general unifier, namely $A\sigma = B\sigma$, and for any θ with $A\theta = B\theta$, $\theta = \sigma\circ\theta$.

This statement of the algorithm might at first blush suggest that, even if we do not recompute $A\sigma$ and $B\sigma$ from scratch at each iteration, we still must start a fresh search each time for differences between them. This is, however, untrue: if we denote the successive values assumed by as σ_0, σ_1, ..., and if certain homologous parts of some $A\sigma_i$, $B\sigma_i$ are already unified by some σ_j (j>=i), then they will continue to be unified by all values of σ subsequent to σ_j. Thus we may write a recursive program for unification which proceeds, in effect, by a left-to-right sweep over the final unified term.

We shall first give as abstract a version as possible of the recursive organization of the algorithm, retaining substitutions, their application, and their composition as primitive ideas, but supposing that terms are represented by list structures as indicated in Section 2. We must consequently restrict all terms to be rational from here on.

We may then write our algorithm as follows:

```
unif[x,y] =
    if atom[x] $\wedge$ atom[y] $\wedge$ x=y then $\epsilon$
    else if $\neg$atom[x] $\wedge$ $\neg$atom[y] then
        let $\tau$ = unif[ax,ay]
        in $\tau\circ$unif[(dx)$\tau$,(dy)$\tau$ ]
    else if varp[x] then $\rho$ where $\rho$ = {x $\mapsto$ y$\rho$ }
    else if varp[y] then $\rho$ where $\rho$ = {y $\mapsto$ x$\rho$ }
    else error[CLASH] .
```

It may be observed that calls of unif are always made with x and y representing homologous subterms of $A\sigma_i$, $B\sigma_i$, where σ_i is one of the substitutions which would arise in the iterative version of the algorithm if it were always to choose the leftmost difference to eliminate; the result of any call of unif is just that additional substitution which must be post-composed with σ_i in order to unify x and y as well.

Unif has been written to retain the equivocation as to whether or not infinite rational terms, as represented by unbounded list structures, are to be countenanced. If only finite terms are allowed, then the circumstance in which a ρ equal to {x $\mapsto$ yρ } does not exist is to be supposed to give rise to an error. Note that execution of unif can terminate (in the absence of clashes) only if all the pairs of term structures x, y which arise are bounded; as the program is developed further this blemish will be removed by the use of tables.

In order to arrive at executable Lisp programs, we need a representation of substitutions by data structures, and with a view to finding an economical representation it is convenient first to introduce the notion of *implicit substitution*. For any substitution τ, we may consider its powers under composition with itself: τ, τ^2, ... Provided there is no set of variables which τ maps into itself without fixed points, these powers will have a limit which we may call τ^* satisfying $\tau^* = \tau \circ \tau^*$ and, for any θ such that $\theta = \tau \circ \theta$, $\theta = \tau^* \circ \theta$. This is because any variable is either left unchanged by τ, eliminated by some finite power of τ, or pushed away into arbitrarily remote subterms by successively higher powers of τ. Then we will say that τ is an *implicit representative* of an ("explicit") substitution σ just in case $\tau^* = \sigma$. Since the successive substitutions σ_i which arise in the process of unification are all idempotent, they are suitable candidates to be implicitly represented; and as we shall see, they in fact have implicit representatives which assign to variables only subterms of the terms which were originally to be unified.

Coming down to actual data structures, we may represent any (explicit or implicit) substitution which alters only finitely many variables by a tabular function with arguments the altered variables and values their assigned terms. Our "tabular representations of implicit substitutions" are what are called by Boyer and Moore [2] simply "substitutions".

Our programs for unification using implicit substitutions will follow essentially the same pattern of recursion as *unif* - that is, sweeping from left to right over the common image of the two input terms under the substitution built up so far; however, they will do so without actually creating any representation of the image term. What we need to make this possible is the function *ult*:

ult[x,s] = *if* defined[x,s] *then* ult[assoc[x,s],s] *else* x .

Here x is any term structure and s is the tabular representation of an implicit substitution, in turn representing, let us say, an explicit substitution σ. The purpose of *ult* is to apply σ to x from hand to mouth; that is to make sure that whenever x is an atom, *ult*[x,s] actually is a usable stand-in for xσ, while not letting itself be put to any trouble by non-atomic x. The effect of this is that chains of alternating ult's and car's (or cdr's) applied to any x, both starting and finishing with ult, will yield just the same atoms in the same places as would those chains with their ult's suppressed applied to $x\sigma$.

We may make one further observation: since a representation of σ, the unifying substitution so far, will need to be everywhere available for the use of ult, it follows that the function analogous to unif whose task it is to compute the most general unifier, τ, of terms $x\sigma$, $y\sigma$ may as well return $\sigma\circ\tau$ rather than τ as its answer; this is what will be wanted in the end, and by accumulating it piecemeal we will avoid having to compose arbitrary substitutions.

It is now easy to write our programs. We give first one for classical unification:

```
mgu[x,y] = unify[x,y,nila[]]
```

The top-level function, mgu, serves only to set up the representation of the identity substitution.

```
unify[x,y,s] = uni[ult[x,s],ult[y,s],s]
```

Unify passes all the work off to uni, but establishes for it the following invariant: that its first two arguments are not defined in its third (meaning that these first two arguments do correspond to an actual subterm of the common image term).

```
uni[x,y,s] = if x=y then s
   else if ¬atom[x] ∧ ¬atom[y]
      then unify[dx,dy,unify[ax,ay,s]]
   else if varp[x] then
      if occ[x,y,s] then error[CYCLE] else extend[x,y,s]
   else if varp[y] then
      if occ[y,x,s] then error[CYCLE] else extend[y,x,s]
   else error[CLASH]
```

Here the job of, say, occ[x,y,s] is to enforce the restriction to finite terms by determining whether x occurs in $y\sigma$; but since $y\sigma$ does not exist, occ must use the same "decomposition" technique as does uni:

```
occ[v,y,s] =
   if atom[y] then v=y
   else occ[v,ult[ay,s],s] ∨ occ[v,ult[dy,s],s]
```

The following invariant is essential to the correctness of occ: y is not defined in s.

The case of unrestricted unification is complicated by the necessity to carry along a tabulated relation of pairs of structures which must be unified, just as equal requires a table of pairs which must be equated, in order to avert infinite recursion. On the other hand, the unrestricted case relieves us of the obligation to forbid substitutions which give rise to unboundedness. Thus we may write the following three functions, closely analogous to mgu, unify, and uni:

```
rmgu[x,y] = runify[x,y,nilr[],nila[]]

runify[x,y,h,s] = runi[ult[x,s],ult[y,s],h,s]

runi[x,y,h,s] = if x=y then s
   else if ¬atom[x] ∧ ¬atom[y] then
      if memr[x,y,h] then s
      else let h' = extendr[x,y,h]
           in runify[dx,dy,h',runify[ax,ay,h',s]]
   else if varp[x] then extend[x,y,s]
   else if varp[y] then extend[y,x,s]
   else error[CLASH] .
```

The function of h here is precisely that of the relation r in equal: we will still compute the unifying substitution, if there is one, even though we ignore any pair of term structures on its second and subsequent appearances. Just as with equal, all the arguments x and y which are ever given to runi are substructures of the original inputs to rmgu, and thus the indefinite expansion of h which non-termination would require is impossible. With some added trouble, h could be made to accumulate all pairs of first two arguments to runi which had been seen so far, but this would be only a half-measure towards representing the whole equivalence relation obtaining at any point between pairs of terms which it is known must be unified with each other.

Two points should be checked before we feel happy about the correctness of mgu and rmgu. First, since the use of ult ensures that uni and runi never see any variables which are defined by the current table s (that is, which are altered by the current implicit substitution), the expressions extend[x,y,s] and extend[y,x,s] always give rise to well-defined substitutions, which do not assign two term structures to the same variable. For the same reason, the implicit substitutions created cannot contain cyclic assignments of variables to each other (e.g., $\{u \mapsto v, v \mapsto u\}$); therefore they correspond to well-defined explicit substitutions. This freedom from direct cycles is also necessary to ensure that ult terminates.

The second point is a little more complicated: we would like to know that the calls of extend create implicit representatives of the right substitutions; that is, given that a substitution $\{V1 \mapsto T1, .., Vk \mapsto Tk\}$ - call it φ - is such that $\varphi^* = \sigma$, and that we form $\psi = \{X \mapsto Y, V1 \mapsto T1, ..., Vk \mapsto Tk\}$, does $\psi^* = \sigma \circ \tau$, where $\tau = \{X \mapsto Y\sigma\tau\}$?

To see that it does, first note that $\psi^* = \varphi\circ\psi^*$, since φ is only doing ahead of time work that ψ^* would quickly get to anyway; hence $\psi^* = \varphi^*\circ\psi^* = \sigma\circ\psi^*$. But since σ eliminates the variables V1, ..., Vk, we can replace ψ^* on the right by a substitution which changes only X; that is, we have $\psi^* = \sigma\circ\{X\longmapsto X\psi^*\}$. Now

$$\begin{aligned}\{X\longmapsto X\psi^*\} &= \{X\longmapsto X\psi\psi^*\}\\ &= \{X\longmapsto Y\psi^*\}\\ &= \{X\longmapsto Y\sigma\psi^*\}\\ &= \{X\longmapsto Y\sigma\{X\longmapsto X\psi^*\}\}\ ;\end{aligned}$$

hence $\{X\longmapsto X\psi^*\}$ is precisely the required τ .

We now have programs which, in a sense, compute most general unifiers; in fact, however, they produce tabular representations of the corresponding implicit substitutions, and to see if these representation tricks have been worthwhile we should investigate the task of applying substitutions, given in various tabular representations, to term structures. Let us consider first the application of a tabulated *explicit* substitution to a bounded term structure, as a straightforward paradigm for the other variants:

```
dosubst[x,s] =
    if atom[x] then
        if defined[x,s] then assoc[x,s] else x
    else (dosubst[ax,s] . dosubst[dx,s]) .
```

Note that *dosubst* produces a tree as "copy" of the structure x, and though by hypothesis this will be finite it may be exponentially larger than x.

The simplest analogous function which will work for a term structure which may be unbounded again produces basically a tree-shaped duplicate of its argument, introducing cycles only as pointers from certain pairs to ancestors of themselves:

```
rtreesubst[x,s] = rtreesub[x,s,nila[]]

rtreesub[x,s,u] =
   if atom[x] then
      if defined[x,s] then assoc[x,s] else x
   else if defined[x,u] then assoc[x,u]
   else letrec u' = extend[x,newx,u]
        and newx = (rtreesub[ax,s,u'].rtreesub[dx,s,u'])
        in newx .
```

As long as we have been obliged to introduce the table u in order to ensure termination, however, we can get a more economical function by imitating *copy*, and producing a result isomorphic to the argument structure, save for the replacement of variables defined in s by their values:

```
rdosubst[x,s] = substract[x,s,rdosub[x,s,nila[]]]

substract[x,s,h] = if defined[x,s] then assoc[x,s]
                   else if atom[x] then x
                   else assoc[x,h]

rdosub[x,s,h] =
   if defined[x,s] V atom[x] V defined[x,h] then h
   else
      letrec h' = rdosub[dx,s,rdosub[ax,s,extend[x,x',h]]]
      and x' = (substract[ax,s,h'] . substract[dx,s,h'])
      in h' .
```

Note that *substract* and *rdosub* are written so as to follow any prescriptions made by the table s, even if these should include, contrary to the representation of implicit substitutions by tables developed so far, assignments of values to pairs as well as to atoms. This precaution will assume significance in Section 6.

It is straightforward to write analogues of *dosubst* and *rdosubst* for the application of implicit substitutions; as usual, we have to call *ult* in the right places. For bounded term structures:

```
appsubst[x,s] = appsub[ult[x,s],s]

appsub[x,s] =
    if atom[x] then x
    else (appsubst[ax,s] . appsubst[dx,s]) .
```

To reproduce the sharing pattern of a possibly unbounded term structure:

```
rappsubst[x,s] = extract[ult[x,s],rappsubs[x,s,nila[]]]

rappsubs[x,s,h] = rappsub[ult[x,s],s,h]

rappsub[x,s,h] =
 if atom[x] V defined[x,h] then h
 else
   letrec h' = rappsubs[dx,s,rappsubs[ax,s,extend[x,x',h]]]
   and x' = (extract[ult[ax,s],h'] . extract[ult[dx,s],h'])
   in h' .
```

Rappsubst is as economical of storage as could be hoped: *rappsubst*[x,s] creates at most one new dotted pair for each distinct pair occurring in x or in one of the term structures in s. Recall that if the implicit substitution s was created by unification of term structures a and b, then it contains only substructures of a and b.

Time and storage may still be wasted, however, if the same implicit substitution is applied to a number of term structures. In this case, the work of making explicit the values of the variables affected by the substitution will be done over again each time. This duplication might be averted by judicious direct use of rappsubs, which could have the additional merit of profiting by any sharing among the structures being subjected to the substitution. It is convenient, however, to have a function at hand which will convert any substitution from implicit to explicit form, thereby getting as much as possible of the creation of new structure done once and for all. This function will bear a strong resemblance to rappsubst, but to define it we need an additional primitive, domain, applicable to tabulated functions. We require that, for any tabulated function s, domain[s] yield a list of all those objects x such that defined[x,s] is true. Now we can write our conversion function:

```
convert[s] = conv[domain[s],s,nila[]]

conv[l,s,h] =
  if nl then nila[]
  else let h' = rappsubs[al,s,h]
       in extend[al,extract[ult[al,s,h'],conv[dl,s,h']]] .
```

It is easily seen that convert[s] produces a structure in which exactly one dotted pair has been created corresponding to each distinct dotted pair in the term structures which constitute s.

It is worth noting that our unification functions for arbitrary term structures - rmgu, rappsubst, convert, and rdosubst - have potentially much more economical running times than do the straightforward mgu and appsubst for bounded structures. As is well known, unification can produce exponential growth in the size of terms. Mgu in effect traces out almost the entire tree structure of the unified term, and appsubst actually creates such a tree structure, with no sharing whatever. The functions for the unrestricted case, by contrast, produce output list structures no bigger than their inputs, and thus are not immediately debarred from running in linear time. In fact, if one dare suppose that the primitive operations on tables require constant time, one sees that rdosubst does indeed run in time linearly proportional to the total size of its input structures, and if there are few enough distinct variables present that ult may in practice be considered to have running time bounded by a constant, then rappsubst and convert are similarly linear. Rmgu appears to have exponential running time if its inputs exhibit much sharing of substructures, but the possibility mentioned above of threading a single monotonically growing relation through all calls of runi would hold down the running time to quadratic at worst.

The supposition of table access time which is constant or very slowly growing does not seem unrealistic, because on examination the programs developed so far are found nowhere to exploit the continued existence of a table of which an extension has been computed. Therefore, they could be recast in an imperative form, using conventional "side-effecting" symbol table techniques such as hashing in place of extend, assoc, and their kin. No such transliterations will be given here. One might hope them to be performable by an automatic program transformation system; alternatively, it may be that data structures can be devised which realize the general side-effect-free table operations specified here with acceptable efficiency. The author regards the latter question as an interesting challenge to implementors.

The comparative economy in both space and time of our programs for unrestricted unification prompts us to ask whether they can be augmented to solve the classical unification problem without losing these virtues. The answer is yes; the essential ingredient in the augmentation is a function which will verify that an arbitrary list structure is bounded, in other words that the quasi-order "is a substructure of" among its parts is in fact a partial order. One way of performing this verification is to embed the quasi-order in a total order, which Knuth's "topological sort" algorithm [8] shows can be done in linear time. N.V.Murray has observed, however [11], that checking boundedness for list structures requires much less ingenuity than does the general topological sort problem: the absence of cycles can be verified by a simple depth-first search. If the list structure does contain a cycle, then at some point one of the previously visited pairs which, together with atoms, cut off the tracing of the depth-first spanning tree must be an ancestor of itself. This idea gives rise to the following function, which causes an error if its argument is unbounded, and otherwise returns true:

```
cyclefree[x] = let h = cf[x,nils[],nils[]] in true

cf[x,ancestors,relatives] =
   if mems[x,ancestors] then error[CYCLE]
   else if atom[x] ∨ mems[x,relatives] then relatives
   else let anc = extends[x,ancestors]
        in cf[dx,anc,cf[ax,anc,extends[x,relatives]]]
```

An economical way to perform classical unification is to compute the entire (unrestricted) unifying substitution first, and only then check it for boundedness. This seems first to have been observed by Baxter [1], who, although he failed to give a correct algorithm, appears to have foreshadowed to some extent the idea behind almost linear unification. Thus we need a function, expcyclefree, analogous to cyclefree but applicable to an explicit substitution. Much as convert finds it advantageous to use *rdosubs* directly rather than going through *rdosubst*, so *expcyclefree* can get by with a single search over all the list structure in a substitution by calling *cf* directly:

```
   expcyclefree[s] = ecfr[domain[s],s,nils[]]

   ecfr[vl,s,relatives] =
      if nvl then true
      else ecfr[dvl,s,cf[assoc[avl,s],nils[],relatives]]
```

If we prefer to work with implicit substitutions we can perhaps do slightly better, for in the context of classical unification we may suppose that the terms actually appearing in an implicit substitution are finite; hence any cycle introduced by a substitution must involve one of the variables which the substitution eliminates. It follows that the set "ancestors" can in the implicit case consist of variables rather than of (presumably more numerous) dotted pairs. Thus we have:

```
 impcyclefree = icfr[domain[s],s,nils[]]

 icfr[vl,s,relatives] =
   if nvl then true
   else icfr[dvl,s,icf[avl,s,nils[],relatives]]

 icf[x,s,ancestors,relatives] =
   if defined[x,s] then
     if mems[x,ancestors] then error[CYCLE]
     else icf[assoc[x,s],s,extends[x,ancestors],relatives]
   else if atom[x] ∨ defined[x,relatives] then relatives
   else icf[dx,s,ancestors,icf[ax,s,ancestors,
                                    extends[x,relatives]]]  .
```

One may note that a correct version of *impcyclefree* could be written in which *relatives* also was a set of variables, but that it would be liable to a running time explosion in the presence of common substructures. Note also that if any of the boundedness-checking algorithms is to be realized by an imperative program using a single mutable data structure to represent the set of ancestors, then a deletion primitive for this structure will be required.

6.0 FAST UNIFICATION

We have now all the pieces in our hands which will enable us to arrive at the almost linear unification algorithm of Huet, Kahn, and Robinson. What we need is to exploit fully at each point the equivalence relation obtaining between those pairs of non-atomic substructures of the inputs which must be unified if the computation is to succeed. Fortunately, the algorithm of Fischer and Galler

[4] solves exactly our problem: to represent an equivalence relation in such a way that it may expeditiously be both interrogated and progressively coarsened. The idea is to represent each equivalence class by a rooted tree connecting its elements, in which the only traversals that need be performed are from an arbitrary node to the root. To test the truth of the relation for two elements, one tests identity of the roots of their trees; to coalesce two equivalence classes, one attaches the root of one as an immediate descendant of the root of the other.

Tarjan [16] has given a full analysis of the running time of this algorithm, taking into account two modifications which are of great importance to the theoretical behavior, but which are not necessarily worth making in practice for equivalence relations of moderate size, and which in any case need not be allowed to clutter up the present attempt at high-level programming. (The first of these, "path compression", is the short-circuiting by a single arc of every path to a root traversed in the course of interrogation and coarsening. The second, "balancing", is taking care that whenever trees are merged, the smaller becomes a subtree of the larger.)

The information necessary to represent the Fischer-Galler trees is simply a partial function mapping each element not a root to its parent; merging two trees is defining this function at an additional argument. Our tabulated functions, with their operation *extend*, are just what is needed here; moreover the function *ult* is exactly that which finds the root of a tree. Looking at our implicit substitutions in this light, we may see that they are Fischer-Galler representations of relations in which at most one non-variable occurs in each equivalence class and, if it is present, forms the equivalence class representative.

All this suggests a drastic simplification of *rmgu* by making pairs and variables be arguments in the same table. This yields the following functions:

```
fmgu[x,y] = funify[x,y,nila[]]

funify[x,y,s] = funi[ult[x,s],ult[y,s],s]

funi[x,y,s] = if x=y then s
   else if ¬atom[x] ∧ ¬atom[y]
        then funify[dx,dy,funify[ax,ay,extend[x,y,s]]]
   else if varp[x] then extend[x,y,s]
   else if varp[y] then extend[y,x,s]
   else error[CLASH] .
```

At first glance, while it appears plausible that *fmgu* computes the desired equivalence relation, one supposes that some complicated decoding process will be necessary in order to extract from it a corresponding substitution. To the author's surprise, this proved not to be the case: a table produced by *fmgu* can be applied by *rappsubst*, or passed through *convert* and the result applied by *rdosubst*, just as if it were an honest implicit substitution. Thus there is no more programming to be done; merely replacing *rmgu* by *fmgu* gives us a complete suite of functions for the rapid computation and application of unifiers. The mapping of pairs to pairs which these new tables prescribe in addition to their action on variables turns out to be a purely beneficial redundancy. The benefit is increased sharing: the common image of the input terms under their unifying substitution can now be produced by creating exactly one new pair for each equivalence class of substructures which contains a pair.

It remains to give some support to the assertions of the preceding paragraph. Inspection of *funi* makes it evident that every merging of two equivalence classes is necessary, that is, happens only if the terms in both classes must indeed be mapped to one term by any unifying substitution. Thus any clash obtained does correctly indicate that unification is impossible. (This same property, that no unnecessary merging is done, also indicates that any unifying substitution obtained will be a most general one, as argued by Paterson and Wegman [12] in the context of classical unification. One might perhaps show this more rigorously by establishing inductively that any equivalence relation obtained by a successful computation with *fmgu* is the same as the relation implicit in the data structures which *rmgu*, or its suggested quadratic-time variant, would construct for the same pair of inputs.)

What is not so evident is that a table returned by *fmgu*, although not covered by our account of the representation of substitutions by tables, does indeed act like a unifying substitution when it is "applied" by *rappsubst*. Actually, the unifying part is easy: the result of *fmgu* has the two input term structures in the same equivalence class, and so *rappsubst*, which accesses its "substitution" argument only via *ult*, cannot distinguish the two and is bound to map them to isomorphic structures. The only real work is to verify that the function on terms computed by *rappsubst* for a given "substitution" s continues to have the homomorphism property characteristic of substitutions.

Although the point is somewhat obscured by the use of the table h and the function *extract* (indeed the reader may prefer at this point to contemplate *appsubst* in place of *rappsubst*) it is not too hard to see that all will be well provided s enjoys the following property, which might be elevated into a definition of "redundant tabular implicit substitution": If s is defined at a pair x then *ult*[x,s] is another pair, x', such that

(*)   ult[**a**x,s] $\underline{=}$ ult[**a**x',s]   and   ult[**d**x,s] $\underline{=}$ ult[**d**x',s] .

The sufficiency of (*) is readily verified in the case of *appsubst*; we have, for example,

```
    a appsubst[x,s] = a appsub[x',s]
                    = appsubst[ax',s]
                    = appsub[ult[ax',s],s]
                    =(*) appsub[ult[ax,s],s]
                    = appsubst[ax,s]
```

which is half the homomorphism property.

But (*) is guaranteed by the determination of *fmgu*, whenever it merges the equivalence classes of two pairs, to merge also the classes of their respective car's and cdr's.

Continuing to regard the running time of *ult* as "almost constant", we readily see that *fmgu* runs in almost linear time, for *funi* can proceed past its first line only by either finding a clash or else reducing the number of remaining equivalence classes, which initially were only as numerous as the distinct pairs and variables in the inputs, by one.

Our original notion of an implicit substitution would suggest viewing what goes on in *fmgu* as the transformation of certain pairs into variables by fiat. It may be that an abstract version of the unification algorithm can be found which allows from time to time the replacement of one or more instances of a composite subterm by a fresh variable, and which will provide a much more direct and satisfactory demonstration of the correctness of *fmgu*.

One further remark on programming may be of interest: the device of path compression can be introduced without abandoning the notion of tables as inviolable entities. One may define:

```
    trace[x,s] =
       if defined[x,s] then
          let x' = assoc[x,s]
          in let s' = trace[x',s]
             in if defined[x',s']
                then extend[x,assoc[x',s'],s']
                else s'
       else s .
```

Then one has the equivalence

ult[x,s] = **if** defined[x,s] **then** assoc[x,trace[x,s]] **else** x ,

but the usefulness of *trace* is that each modified table it produces can be saved and become the input to the next table lookup. An extensive but straightforward reconstruction of all the unification functions would be required in order to exploit this idea.

ACKNOWLEDGEMENTS

A major impetus for the work reported here has been the research of Professor J.A.Robinson which has resulted in a succession of improvements in both the description and the programming of unification, and in which he has welcomed my participation. I thank him for this, and for an enormous amount of encouragement and criticism in the preparation of this paper. N.V.Murray has kindly shared with me his insights into the principles of fast unification. Professor E.E.Sibert has been an informed reader of early drafts, and has contributed signifcant improvements. I also thank him and the other participants in the Syracuse Lisp group for their collaboration and interest.

REFERENCES

1. Baxter, L.D., "An efficient unification algorithm", Technical Report CS-73-23, Dept. of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario (July 1973).

2. Boyer, R.S. and Moore, J.S., "The sharing of structure in theorem-proving programs", *Machine Intelligence 7*, Edinburgh University Press (1972), pp.101-116.

3. Burge, W.H., *Recursive Programming Techniques*, Addison-Wesley, Reading, Mass. (1975).

4. Galler, B.A. and Fischer, M.J., "An improved equivalence algorithm", *Comm. ACM 7*, 5 (May 1964), pp.301-303.

5. Goguen, J.A., Thatcher, J.W., Wagner, E.G., and Wright, J.B., "Initial algebra semantics and continuous algebras", *JACM 24*, 1 (January 1977), pp.68-95.

6. Hoare, C.A.R., "Recursive data structures", Memo AIM223, Stanford Artificial Intelligence Laboratory (October 1973).

7. Huet, G., *Resolution d'Équations dans des Langages d'Ordre 1, 2, ..., ω*, doctoral thesis, University of Paris VII (September 1976).

8. Knuth, D.E., The Art of Computer Programming, Vol.1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1969).

9. Landin, P.J., "The mechanical evaluation of expressions", The Computer Journal 6, 4 (January 1964), pp.308-319.

10. McCartny, J., Abrahams, P.W., et.al., Lisp 1.5 Programmer's Manual, MIT Press, Cambridge, Mass. (1962).

11. Murray, N.V., forthcoming doctoral thesis, School of Computer and Information Science, Syracuse University, Syracuse, N.Y.

12. Paterson, M.S. and Wegman, M.N., "Linear unification", Proceedings of the Eighth Annual ACM SIGACT Symposium on the Theory of Computing, Hershey, Pa., May 3-5, 1976.

13. Robinson, J.A., "A machine-oriented logic based on the resolution principle", JACM 12, 1 (January 1965), pp.23-41.

14. ____________, "Fast unification" (abstract), Proceedings of a Conference on Mechanical Theorem Proving, Mathematisches Forschungsinstitut Oberwolfach, January 10, 1976.

15. ____________, Logic: Form and Function, Edinburgh University Press (1978).

16. Tarjan, R.E., "Efficiency of a good but not linear set union algorithm", JACM 22, 2 (April 1975), pp.215-225.