Pattern Driven Lazy Reduction: A Unifying Evaluation Mechanism for Functional and Logic Programs

(Extended Summary)

P.A.Subrahmanyam and J-H. You

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

A novel lazy evaluation mechanism, *pattern-driven lazy reduction*, is developed that serves as a unifying evaluation mechanism for both functional and logic programs. The reduction of a function call can be viewed as "semantically" unifying the function call with the left hand side of a defining equation, and applying the unifier to the right hand side. Lazy reduction is achieved by the pattern which the function call matches against. Function reductions are actually "driven" by patterns in this sense. It is shown that this evaluation mechanism works well for both functional programs and logic programs that involve "executable" functions. As a result, logic programs can be enhanced with (1) the availability of a functional computing environment where there is no notion of backtracking, thus alleviating the degree of control difficulties typically encountered in logic programs, and (2) the ability to terminate "infinite computations" without the introduction of complex control issues at the user-level. On the other hand, functional programs can be equipped with the power of logic programming languages, e.g., Prolog.

1. Introduction

Since Kowalski's proposal for using predicate logic as a programming language [12], the theory and pragmatics of logic programming have gained increasing attention so much so that the number of users in the United States is approximately doubled each year. The prototypical representative of logic programming is Prolog (see, for example, [1, 4, 5, 12, 19]), which has been evidenced to be a simple and powerful language.

However, current Prolog systems suffer from a number of problems, including (1) a general difficulty in controlling program execution, (2) the lack of the concept of "evaluation", leading to the logic base sometimes not being transparent, and (3) the unavailability of means to describe terminating computations on infinite data structures.

On the other hand, functional languages with equational flavors have also attracted many researchers (see, for example, [2, 18]). The major drawback of these languages, as compared to Prolog, is the lack of knowledge-based inferencing abilities to cope with search-based computations.

This paper introduces a novel evaluation mechanism, pattern-driven reduction, which can (1) be employed as an evaluation mechanism for functional programs, and (2) serve as a basis for a combined computing environment of logic and functions, which supports computing on infinite data structures, and provides both backtracking free (functional) and non-deterministic (logic) computing components for the user to choose from. Some of the "induced" (and therefore unnatural) control problems typically encountered in Prolog programs can be eliminated or eased by (1) writing program segments as function equations; and (2) explicitly embodying the concept of "evaluation" in the programs written.

In the next section we introduce the notion of pattern-driven reduction in a functional environment. The remainder of the paper continues with discussions of applications of the mechanism in a combined environment of logic and functions. In what follows, we will use Edinburgh DEC-10 Prolog notation[1], except for using ^ (up arrow) for the list constructor *cons*, for example, writing [A|X] as A ^ X (equivalent to the Lisp notation cons(A, X)) for convenience. The reader should think of this and other constructors as merely symbols without pre-defined meanings. We assume that the reader is familiar with logic programming in general.

This Research was sponsored in part by the Defense Advanced Research Projects Agency, US Department of Defense, Contract No. MDA903-81-C-0414.

[1] In Edinburgh DEC-10 Prolog, identifiers with the first letter capitalized denote variables, and with the first letter being lower case denote constants. A Lisp list, for example, (1 2 3), can be written as [1,2,3] in Edinburgh Prolog and can be also written as 1 ^ 2^ 3^ [ ] in this paper.

2. Pattern–Driven Reduction in a Functional Environment

2.1. Syntax of Function Declarations

The functional environment considered here consists of (1) a set of function definitions and (2) a set of "strict" primitive functions, i.e., some primitive boolean functions and arithmetic operations[2]. A function definition consists of a set of equations. To simplify our exposition, we assume a flat computing environment, denoted by EQN. Base constructors (e.g., 0, [ ], ∅, etc) and constructors (e.g., successor, cons, push, etc) can be freely invented by the user. Terms composed of base constructors, constructors and variables can stand for formal parameters and can be used in equations. The left hand side of an equation can have more than one set of parameters (see Example 2 below). The following two simple examples serve to illustrate the syntax of the functional environment.

Example 1. *Inorder listing of a binary tree*

inorder(tr(∅, N, ∅)) = N ^ [ ]
inorder(tr(L, N, R)) = append(inorder(L), N ^ inorder(R))
append([ ], L) = L
append(A ^ L1, L2) = A^ append(L1, L2)

Example 2. *A Curried version of the function map*

map(F)([ ]) = [ ]
map(F)(A ^ L) = F(A)^ map(F)(L)

Note that the syntax of Example 2 resembles the Curried version of function definitions. Doing this introduces the concept of "resulting functions" into equations. The function variable F can now be instantiated to a resulting function, not necessarily a defined function.

2.2. Reduction of Function Terms

The reduction of a function term $f(s_1,...,s_n)$ can be viewed as finding a substitution Q for the left hand side $f(t_1,...,t_n)$ of an equation such that $f(t_1,...,t_n)Q = f(s_1,...,s_n)$. The "result" of the reduction is obtained by applying Q to the right hand side of the equation[3]. This one way unification is called *biased unification* in this paper. The corresponding reduction procedure is called *One_Step_Reduction*.

In trying to unify two terms, say, $f(t_1,...,t_n)$ and $f(s_1,...,s_n)$, suppose there are subterms t_i and s_i that are not unifiable in the conventional sense. If s_i (or t_i) is reducible and the reduced version of it is unifiable with t_i (respectively s_i), then we say that $f(t_1,...,t_n)$ and $f(s_1,...,s_n)$ are *semantically unifiable*. The corresponding process is called *semantic unification*. By combining semantic unification with biased unification we obtain the notion of *biased semantic unification*. The lazy evaluation effect is achieved by requiring that a biased unification algorithm perform "reduction–by–need", that is, perform only those reductions that are necessary in order to make two terms semantically unifiable. Consider the following example which defines a generator g of an infinite stream of squares (of integers), and a function f.

Example 3.

g(N) = N*N ^ g(N + 1)
f(X,M ^ Y) = if X = 0 then M^ f(X–1,Y) else f(X + 1,Y)

One_Step_Reduction of the term f(1, g(5)) gives us 25 ^ f(0, g(6)) with Q_f = { X = 1, M = 25, Y = g(6) } and Q_g = { N = 5 }. Notice that g(5) needs to be reduced first in order to reduce f(1, g(5)).

The above example serves to provide some intuition about the notions of pattern–driven reduction and reduction–by–need. When the terms M ^ Y and g(5) were to match, g(5) was reduced once to yield 25 ^ g(6) which was then matched with M^ Y with M bound to 25 and Y to g(6). If we try to match, say, M^ N^ Y with g(5), then g(5) will be reduced twice to become 25 ^ 36^ g(7). Thus, such reductions of function terms are in fact driven by *patterns*, and this is what we mean by *pattern–driven reduction*. The actual number of reductions carried out is based on the *need* to match a pattern and this is what we mean by reduction–by–need. Notice that by adopting an equational model the notion of *lazy evaluation* extends to any algebraic data type in a very natural way without increasing language complexity.

The careful reader has probably noticed that the definition of the function if_then_else and its involvement in the reduction has been ignored. The function if_then_else can be defined (in prefix notation) by equations

if_then_else(true, X, Y) = X
if_then_else(false, X, Y) = Y

The reduction of a function term, for example,

if_then_else(f(a), g(b), h(c))

will cause f(a) to match against the boolean constant true or false, which in turn demands reduction of f(a). The function if_then_else defined in this fashion is non–strict, that is, neither g(b) nor h(c) is demanded in the current reduction.

2.3. A Biased Semantic Unification Algorithm

We now give the definition of reducibility of function terms below. A biased semantic unification algorithm is described in Figure 1 and the procedure One_Step_Reduction in Figure 2. Functions can have more than one set of parameters; for clarity of our exposition, however, we will write function terms with only one set of parameters.

Definition 1: *Reducibility of Function Terms*
Let $f(s_1,...,s_n)$ be a function term to be reduced. Let $g_1,...,g_p$,where $p \geq 0$, be function calls contained in $f(s_1,...,s_n)$. The function term $f(s_1,...,s_n)$ is *reducible* if there exists (1) a function term $f(s_1',...,s_n')$ obtained from $f(s_1,...,s_n)$ by performing zero or more reductions of each of $g_1,...,g_p$, (2) an equation whose left hand side is $f(t_1,...,t_n)$ and (3) a substitution Q such that $f(t_1,...,t_n)Q = f(s_1',...,s_n')$. □

Note that in the algorithm, we assume that common variables cannot appear on the left hand side of an equation. As a result, the equations in the set I will not be interrelated, thus eliminating the necessity of processing equations that are interrelated by common variables.

[2]Strictly speaking, all primitives can be defined by equations. Thus, the choice of primitives is not a crucial step in modeling.

[3]However, we will write, for example, 7 instead of 5 + 1 + 1 for convenience.

A Biased Semantic Unification Algorithm

```
Input: The left hand side of an equation
       f(t1,...,tn), a function
       term f(s1,...,sn) and EQN.

Output: A failure report or a substitution
        Q={x1=m1, x2=m2,...,xp=mp}, where
        xi's are variables and mi's are terms.
Begin
    I <- {t1 = s1, ...,tn = sn};
    Q <- ∅;
    Repeat Until I = ∅
      remove an equation ti = si from I;
      if ti is identical to si
              then do nothing
      else if ti is a variable
              then add ti = si to Q
      else if si is a constant or a variable
              then stop with failure
      else  let ti = g(t1',...,tp')
            and si = g'(s1',...,sq')
            if g = g' and p = q
            then add ti'=si' for all i to I
            else if si is reducible
                    then add
                    ti=One_Step_Reduction(si) to I
            else stop with failure;
      End Repeat;
      Output Q;
End.
```

Figure 1: A biased semantic unification algorithm

```
Procedure One_Step_Reduction(f(s1,...,sn));
Begin
  find an equation f(t1,...,tn) = RHS in EQN;
  call the biased semantic unification algorithm
  with input terms f(s1,...,sn) and f(t1,...,tn)
  Return the result of applying Q to the RHS
         where Q is the substitution returned
         by the biased semantic unification
         algorithm;
End.
```

Figure 2: The procedure One_Step_Reduction

2.4. An Interpreter for Function Reductions

An interpreter for function reductions is presented in Figure 3. The top level driving process is called Reduce. An illustration of the reduction process followed by this interpreter is given in Example 4.

```
Reduce(term)
  Begin
     if term is an atomic term or not reducible
             then Return term;
     let term be f(t1, ..., tn)
     if f is a constructor
             then Return f(Reduce(t1),...,Reduce(tn))
     else Return Reduce(One_Step_Reduction(term))
  End
```

Figure 3: An Interpreter for Function Reductions

Example 4. *Computing any finite portion of an infinite sequence whose elements are pairs of integers and their squares, both increased by 1.*

```
pairs(N) = [N,N*N] ^ pairs(N+1)
first_N_pairs(N) = truncate(N, map(map(plus1))(pairs(1)))

plus1(N) = N+1
truncate(0, L) = [ ]
truncate(N, A ^ L) = if N > 0 then A^ truncate(N-1, L)
```

The function map is defined in Example 2. The sequence of reductions in interpreting the function term first—N—pairs(2) is shown in Figure 4.

```
first-N-pairs(2)
|
V
truncate(2, map(map(plus1))(pair(1)))
|
V
map(plus1)([1,1]) ^
            truncate((1, map(map(plus1))(pair(2)))
|
V
[2,2] ^map(plus1)([2,4]) ^
            truncate(0, map(map(plus1))(pair(1)))
|
V
[2,2] ^[3,5] ^[ ]
```

Figure 4: A sample run of Example 4

3. Computing with Infinite Data Structures

Before discussing applications of pattern-driven reduction in logic programming, it is relevant to motivate the need for allowing computations on infinite data structures and the difficulty of achieving this in logic programming languages.

The advantages of being able to compute with infinite data structures have been cogently argued for by several researchers e.g., [9, 10]. By computing with infinite data structures, we mean that the program can manipulate as a whole data objects that are conceptually infinite, even though the user may, in any given computation, only wish to obtain a finite truncation of the potentially infinite objects. This ability not only provides a new programming style in which one can separate the data processing aspect from possibly complex boundary conditions, but also provides a way to elegantly solve certain classes of problems.

In Prolog, data objects are dynamically built up as variable bindings that result from unification. Since infinite data structures can only be generated as a result of an infinite number of procedure calls, in order to generate an infinite data structure, there must be at least one procedure that can potentially *run forever* (and consequently, the corresponding formula can neither be proved true nor false). On the other hand, since only a *finite* portion of the infinite structure need be manifest during any particular execution, it should be possible to stop the program (i.e. for the formula to be proved true) at *any time*. Consider, for example, the generator of an infinite list of integers starting from N:

```
gen(M, L).
gen(N, N ^ L) :- N1 is N + 1, gen(N1, L).
```

where the relation *is*[4] is a built–in procedure in DEC–10 Prolog [14]. The second clause can be used to generate an infinite list of integers, while the first clause can stop the computation at any time, if invoked. The goal ?– gen(1, L) will have L bound to the infinite list of integers starting from 1 on the assumption that the first clause is *never* invoked. The drawback of this definition is that while defining an infinite data structure generator, the programmer must be concerned with the problem of generating the proper termination of the program. This is an inherent phenomenon for a logic programming language based on the resolution procedure and cannot be avoided, since every formula should be proved true in order for a computation to terminate meaningfully.

In order to obtain only finite portions of this infinite list of integers, we may want to define a procedure for finite truncation. One such procedure may be defined as follows:

```
truncate(0, Infinite, [ ]).
truncate(N, A ^ Infinite, A^ Finite)
        :- N1 is N–1, truncate(N1, Infinite, Finite).
```

To define a procedure that generates the first N integers starting from 1, we can then write the following clause:

```
first_N_ints(N, L) :- truncate(N, L1, L), gen(1, L1).
```

Although this program is "logically" correct, specifying an appropriate control strategy for the program may not be an easy task. To properly terminate the program and to gain efficiency, control facilities of the underlying logic programming language should support the following control strategy:

1. The execution of the goal should be coroutine–triggered (demand–driven) where the process *gen* is the designated producer and the process *truncate* the consumer.
2. The second clause for *gen* (amongst two clauses defining the process *gen*) should always be chosen when the consumer process *truncate* has not been terminated (resolved to an empty clause). The first clause must be chosen after the consumer *truncate* has been terminated.

The key to this control strategy is to choose the right clause for *gen* at the right time, depending on whether or not there exist consumer processes. For cyclic infinite streams (e.g., the solution to Hamming's problem in [3]), however, this strategy no longer works, because it is not clear when to terminate those consumer processes that are involved in the cycle(s), especially in the case that cycles are interconnected to form a complex network. It is true that, because of the completeness of the resolution procedure, there always exists a proof for a "logically" correct program, which means there always exists a way to terminate the program. However, due to the control complexity, as far as we know, no existing Prolog systems based on Robinson's resolution procedure [16] provide control features which can be used to terminate the computation of a goal like, say, ?– first_N_ints(100, L).

[4] *Z is X* means that the integer expression *X* is evaluated and the result is unified with *Z*. *Z is X* fails if *X* is not an integer expression.

4. Pattern–Driven Reduction in a Combined Environment of Logic and Functions

We now show how pattern–driven reduction can be employed in a computing environment that affords both logic and functions. Doing this introduces the notion of "executable functions" in relations, that is, the concept of function evaluation is introduced in Horn clauses. To properly embed functions into relations, we need to modify the conventional unification algorithm to deal with executable functions. The modified procedure is called *(general) semantic unification* in this paper.

4.1. A Semantic Unification Algorithm

The process of unifying two terms can be viewed as one of solving simultaneous equations [13]. For example, the problem of unifying $f(s_1, ..., s_n)$ with $f(t_1, ..., t_n)$ is equivalent to solving the set of equations $\{s_1 = t_1, s_2 = t_2, ..., s_n = t_n\}$. The solution of these equations, if one exists, is of the form $\{x_1 = p_1, x_2 = p_2, ..., x_m = p_m\}$ where the x_i's are variables and the p_i's are terms.

In conventional unification, an equation $g(t_1,...,t_p) = g'(s_1,...s_q)$ is not solvable if g and g' are not identical or if $p \neq q$, since g and g' are both treated as constructors. In contrast, semantic unification allows either g or g' or both (or any function symbol in the equation) to be defined functions. Two terms are semantically unifiable (i.e., the corresponding equation is solvable in this context) if there exist semantically equivalent forms (obtained by reduction(s) on any reducible term in the equation) which are unifiable. The lazy evaluation effect can be achieved by requiring that a semantic unification perform reduction–by–need, that is, perform only those reductions that are necessary in order to make an equation solvable. Presented in Appendix I is a formalization of this extension to the conventional unification algorithm, e.g., the one described in [13]. The algorithm is non–deterministic (in the sense that the transformation rules can be applied in an arbitrary order) and abstract (in the sense no concrete implementation strategy is supplied). Note that the algorithm ignores "occur checks"[5].

As an exercise, the reader may work through the sequence of changes to the equation {30*factorial(X) = factorial(6)}, where *factorial* is a defined function and the multiplication operation * is assumed to be associative. As expected, the solution to this equation is {X = 4}. *"Conventional" unification will fail in this example.* The following example illustrates how lazy reduction is achieved in this combined environment.

Example 5. *Indeterminately choosing a "good" character.*

```
filter(P, [ ]) = [ ]
filter(P, A ^ L) = if P(A) then A^ filter(P, L) else filter(P, L)

good(N) = if (N = a) or (N = b) then true else false

choose(First ^ Rest, First).
choose(First ^ Rest, Second) :- choose(Rest, Second)
```

Applying the semantic unification algorithm described above to

[5] "Occur checks" refer to checks to see if a variable that is being substituted for by a term occurs in the term itself, e.g. as in x = f(x). It has been argued that cycles can arise only if assertions and queries are formulated in certain abnormal ways [17]. Most Prolog systems do not perform "occur checks" in order to improve efficiency of the unification procedure.

the equation

choose(filter(good, [c,b,f,a]), Good) = choose(First ^ Rest, First)

we can get the unifier Q = {Good = First, First = b, Rest = filter(good, [f,a])}. The function term filter(good, [f,a]) is a part of the result of reducing filter(good, [c,b,f,a]) twice. The reductions were driven by the pattern First ^ Rest.

4.2. A Sample Execution in the Combined Environment

The underlying execution model for the combined environment can be envisioned as consisting of two processes: a *driving process* and an *answer–collector*. The driving process invokes the resolution procedure which is based on semantic unification while the process answer–collector does "back substitutions" and demands complete reductions of function terms "thrown out" by the driving process. Notice that without the process answer–collector, the corresponding execution can be incomplete, that is, there can be some function terms left unevaluated.

We would like to re–emphasize the fact that there is no notion of backtracking in function reductions. This means that two goals like

?– p(X, gen(1))
?– p(X, 1 ^ gen(2)),

where gen(N) = N ^ gen(N + 1), are semantically equivalent, since the sign "=", meaning "can be replaced by", is actually an equivalence–preserving transformation. Therefore, there is no need to go back from the second goal to the first.

Example 6. *Evaluating a polynomial of order n at a series of x values in a given range. (Such evaluations are used in plotting curves.)*

A polynomial of order n is represented by a list of n coefficients. For example, $2*x^2 + 4*x - 5$ is represented by the list [–5,4,2].

```
poly(X, An ^ L) = X*poly(X, L) + An
poly(X, [ ]) = 0
curve(X, Inc, L) = poly(X, L) ^ curve(X + Inc, Inc, L)

curve_in_range(Start, End, Inc, Coeffs, Y_values)
 :– range(Start, End, Inc, curve(Start,Inc,Coeffs), Y_values).
range(Current, End, Inc, L, [ ]) :– Current > End.
range(Current, End, Inc, A ^ L1, A^ L2)
          :– range(Current + Inc, End, Inc, L1, L2).
```

In defining the function *curve*, one does not have to worry about boundary conditions, that is, the engineering of how the function terminates. This makes it easier to program and the resulting program is somewhat more flexible.

Several steps by the driving process in executing the goal

?– curve_in_range(–1, 1, 0.1, [2,5], L)

is sketched in Figure 5. The actions of the process answer–collector are included as comments. The sub_i's are used to show the bindings of interests.

Driving Process

```
?–  curve_in_range(–1, 1, 0.1, [2,5], L)
 |
 |  sub1 = {L = Y_values}
 |
 |  /* the process answer–collector is created   */
 |  /* and waits for the binding of Y_values */
 V
?–  range(–1, 1, 0.1, curve(–1, 0.1, [2,5]), Y_values)
 |
 |  sub2 = {A = poly(–1, [2,5])
 |          L1 = curve(–0.9, 0.1, [2,5])
 |          Y_values = A ^L2
 |
 |  /* answer–collector demands reduction of      */
 |  /* poly(–1, [2,5]) and waits for the binding */
 |  /* of L2                                      */
 V
?–  range(–0.9, 1, 0.1, curve(–0.9, 0.1, [2,5]), L2)
    ......
```

Figure 5: A sample execution of Example 6

5. A Solution to the Problem of Finding the First N Lucky Numbers

Example 7 *Finding the First N Lucky Numbers.*
A well–known number–theoretic computation involves the computation of lucky numbers which proceeds as follows: from the list of numbers 1,2,3,4,... we remove every second number, leaving the list 1,3,5,7,9,.... Since 3 is the first number (except for 1) that has not been used in sifting, we remove every third number from the remaining numbers, obtaining 1,3,7,9,13,15,19,21,.... Now every seventh number is removed, leaving 1,3,7,9,13,15,21,.... and so on. The numbers that are never removed from the list are the lucky numbers. The original description of the problem can be found in [15].

The difficulty with computational frameworks that do not support computing with infinite data structures is that it may require complex number–theoretic calculations to estimate *a priori* how many integers should be produced in order to get the first N lucky numbers. One can, of course, use a conservative range of integers to search for the first N lucky numbers — in which case some of the computations will be redundant.

A program to compute the first N lucky numbers without any redundant computations is given below:

```
gen(N) = N ^ gen(N + 2)
sift(M ^ L, N) = M^ sift(sieve(L, M, N), N + 1)
sieve(A ^ L, M, N) = if M = N then sieve(L, M, 1)
                     else A ^ sieve(L, M, N + 1)

truncate(0, Infinite, [ ]).
truncate(N, A ^ Infinite, A^ Finite)
          :– truncate(N–1, Infinite, Finite).
first_N_lucky_numbers(N,L)
                    :– truncate(N, 1 ^ sift(gen(3), 3), L).
```

The function *gen* is used to generate all odd numbers (since no even number can be a lucky number). The function *sift* incrementally outputs the first number M in the current list as a lucky number, calls on the function *sieve* to remove every Mth

number in the list, and then starts the same process again on the remaining list. The variable N is used to count the position of each number in the list in order to determine whether the number should be removed or not.

6. Related Work and Discussion

We have described a novel lazy evaluation mechanism, pattern–driven lazy reduction, that provides a unifying evaluation mechanism for functional and logic programs. This enhances the logic computing environment by adding the ability to gracefully terminate computations on infinite data structures and extends the functional computing environment by providing a knowledge–based inferencing capability. We have shown how pattern–driven reduction can serve as a basis for cleanly integrating these two computing environments into a more powerful one.

The conventional lazy evaluation mechanism is described elsewhere, e.g., in [6] and [9], and has been incorporated into several languages with equational flavors, e.g., those described in [18] and [2]. The pattern–driven lazy reduction mechanism advocated in this paper uses pattern matching as the "demand" for achieving laziness. The advantages of pattern–driven lazy evaluation are: (1) simplicity (only an extended unification algorithm is needed); (2) generality (higher order functions and lazy evaluation on arbitrary algebraic data types are both supported without increasing language complexity); (3) eliminating the need of specifications from the user (e.g., *delay* and *force* in [9]); and (4) provision of a clean and natural mechanism for integrating functions and logic.

A demand–driven lazy evaluation rule is provided in a logic programming language described in [8] to achieve the ability to terminate computations on infinite data structures. This language is based on natural deduction and provides a functional notation. However, the rule they give only deals with acyclic computations, that is, for infinite data structures built up cyclically the rule is inadequate. In addition, it is the user's responsibility to explicitly specify that the laziness is required. Further, it is not feasible to incorporate facilities that deal with higher order functions in this language due to the first order nature of the deduction system used.

An interpreter for function term reductions has been implemented in PSL [7]. We are currently developing an implementation–oriented general semantic unification algorithm. The development of a precise computational model based on the pattern–driven reduction mechanism and a corresponding interpreter is also under way. The report on this work will appear in a future paper.

7. References

1. Kenneth A.Bowen. Prolog. ACM national conference, 1979, pp. 14–23.

2. R. M.Burstall and D.T. Sannella. *HOPE User's Manual.* Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1980.

3. K.L. Clark and Steve Gregory. A relational language for parallel programming. Functional programming and computer architecture, October, 1981, pp. 171–179.

4. W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer–Verlag, 1981.

5. M. H.van Emden. Programming with resolution logic. Machine Intelligence 8, London, 1978, pp. 266–299.

6. D. P.Friedman and D.S. Wise. CONS should not Evaluate Its Arguments. Automata, Languages and Programming, Edinburgh, 1976.

7. The Utah Symbolic Computation Group. *The Portable Standard Lisp Users Manual.* 3.1 edition, Dept. of Computer Science, University of Utah, 1983.

8. A. Hansson, S.Haridi and S.–A. Tarnlund. Properties of a logic programming language. Logic programming, New York, 1982, pp. 267–280.

9. P. Henderson. *Functional programming.* Prentice–Hall, 1980.

10. Robert M.Keller. Semantics and applications of function graphs. Tech. Rept. UUCS–8–112, University of Utah, Dept. of Computer Science, October, 1980.

11. Bill Kornfeld. Equality for Prolog. 8th International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, August, 1983.

12. R. A.Kowalski. Predicate logic as a programming language. Proceeding of IFIP 74, 74, pp. 556–574.

13. Alberto Martelli and Ugo Montanari. "An efficient unification algorithm." *ACM Transaction on Programming Languages and Systems 4*, 2 (April 1982), 258–282.

14. Luis Moniz Pereira, Fernando C N Pereira and David H.D. Warren. *User's Guide to DECsystem–10 Prolog.* 1978.

15. Edward M.Reingold, Jurg Nievergelt and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice–Hall, 1977.

16. J. A.Robinson. "A Machine Oriented Logic Based on the Resolution." *J. ACM 12*, 1 (January 1965), 23–41.

17. J. A.Robinson and E.E. Sibert. Logic programming in Lisp. School of Computer and Information Science, Syracuse University, December, 1980.

18. D. A.Turner. *SASL Language Manual.* University of St. Andrews, 1979.

19. D. H.D. Warren, F.Pereira and L.M. Pereira. "PROLOG: the language and its implementation compared with LISP." *SIGPLAN Notices 12*, 8 (1977), 109–115.

8. Appendix I: A Semantic Unification Algorithm

Global variables and auxiliary procedures used in the algorithm:

- QUEUE: a first–in–first–out queue whose elements are *sets* of equations. Initially it is empty.
- C: the current set of equations.
- Add(SET, QUEUE): inserts the set (of equations) SET into QUEUE.
- Add_all(E, C, QUEUE): for each reducible term t in the equation E, let

New_set = C–E ∪ E[t/One _step _Reduction(t)]

do Add(New_set, QUEUE), where * e[t/t'] denotes e with t replaced by t', and C–E denotes set difference.

– Front(QUEUE): returns the front set in QUEUE. The set returned is deleted from QUEUE.

A Semantic Unification Algorithm

```
Input: Two relation terms p(s1,...,sn),
       p(t1,...,tn) and EQN.

Output: A failure report or a set of equations
        {x1 = m1, x2 = m2, ..., xp = mp}
        where xi's are variables
        and mi's are terms

Begin
  Add({t1 = s1, ...,tn = sn}, QUEUE);
  Repeat Until QUEUE = ∅ or success is reported
    C <- Front(QUEUE);
    Repeat Until a failure is reported or no
                 transformation applies, in which
                 case success is reported.

    /* Repeatedly attempt to apply any of the */
    /* following transformations to each      */
    /* equation in C                          */

    1. Eliminate any equations of the form x=x.
    2. If x and y are distinct variables, x=y is
       in C and x  has other occurrences in C,
       then replace these occurrences of x by
       occurrences of y.
    3. Replace t=x by x=t if x is a variable
       and t is not.
    4. If x is a variable and t1 and t2 are not,
       replace x=t1 and x=t2 by x=t1 and t1=t2.
    5. In case an equation E is of the form
       g(t1',...,tp') = g'(s1',...,sq')
       where p > 0 and q > 0

       case (a) p = q and g = g' and both
                g and g' are constructors
           C <- C–E ∪ {t1'=s1',...,tp'=sq'};
       case (b) p = q and g = g' and both
                g and g' are defined functions
           C <- C–E ∪ {t1'=s1',...,tp'=sq'};
           if there exist reducible terms
                    then Add_all(E, C, QUEUE);
       case (c) one of g and g' is a defined
                function, say g,  and the
                corresponding term g(t1',...,tp')
                is reducible, and the other,
                i.e., g', is a constructor
           let LHS = g(t1',...,tp')
           C <- C–E ∪
                  {One_Step_Reduction(LHS)
                   = g'(s1',...,sq')};
       case (d) (p =/  q or g =/  g')
                and there exist reducible terms
           Add_all(E, C, QUEUE);
           C <- Front(QUEUE);
       case (e) none of the above applies
           report a failure
    End Repeat
  End Repeat;
  if success has been reported
        then the semantic unification
                succeeds with output C
  else the semantic unification fails
End.
```