

LOGIC PROGRAMMING *cum* APPLICATIVE PROGRAMMING*

Nachum Dershowitz
 Department of Computer Science
 University of Illinois
 Urbana, IL 61801
 U.S.A.

David A. Plaisted**
 Department of Computer Science
 University of North Carolina
 Chapel Hill, NC 27514
 U.S.A.

ABSTRACT

Conditional (directed) equations provide a paradigm of computation that combines the clean syntax and semantics of both PROLOG-like logic programming and (first-order) LISP-like applicative (functional) programming in a uniform manner. For applicative programming, equations are used as conditional rewrite rules; for logic programming, the same equations are employed for "conditional narrowing". Increased expressive power is obtainable by combining both paradigms in one program.

1. INTRODUCTION

An *applicative (functional) program* is a set of *directed* equations used for computing. For example, the following is a LISP-like [McCarthy, *etal.*-60] program for concatenating two lists of elements:

<i>Applicative List Append</i>
$append(U, V) \Leftarrow$ if $U = \mathbf{nil}$ then V else $\mathbf{car}(U) \cdot append(\mathbf{cdr}(U), V)$

where **nil** is the empty list and \mathbf{car} denotes the **cons** function. The symbol \Leftarrow has the declarative meaning "is equal", but operationally restricts the use of an equation to replacing instances of the left-hand side with the corresponding right-hand side. [Throughout this paper we follow the convention of using lower case for constants and upper case for free (universally quantified) variables. Bold-face is used for standard built-in functions and constants; italics for user-defined ones.] With the usual "call-by-value" semantics, an *innermost* occurrence of *append* in a term will be replaced by the *value* of the right-hand side. Furthermore, the condition in an **if ... then ... else ...** expression is evaluated before either the **then** or **else** branch is. Given, for example, the term

$$append(a \cdot b \cdot \mathbf{nil}, c \cdot d \cdot e \cdot \mathbf{nil}),$$

the above program will compute the result of appending the two-element list $a \cdot b \cdot \mathbf{nil}$ [with parentheses, that should be $a \cdot (b \cdot \mathbf{nil})$] to the front of the list $c \cdot d \cdot e \cdot \mathbf{nil}$.

Pattern-directed applicative languages include SASL [Turner-79], HOPE [Burstall, *etal.*-80], OBJ2 [Futatsugi, *etal.*-84], ML [Gordon, *etal.*-79], rewrite languages [Hoffmann-O'Donnell-82], and Planner-like [Sussman, *etal.*-70] languages. In these languages, the left-hand side of an equation need not be of the form $f(X_1, \dots, X_n)$, where f is a defined function and X_1, \dots, X_n are variables. More than one equation may be given for f , though some restrictions on the form of left-hand sides are often imposed. For example, the following version of *append* has two equations for the two list *constructor* functions **nil** and **cons**:

<i>Pattern-Directed Append</i>	
$append(\mathbf{nil}, V)$	$\Leftarrow V$
$append(A \cdot U, V)$	$\Leftarrow A \cdot append(U, V)$

A *logic program*, as described in [Kowalski-74], is a set of Horn clauses used as a pattern-directed program that *searches* for output terms satisfying a given goal for given input terms. In this paradigm, the *append* program could be expressed (in PROLOG [Clocksin-Mellish-84]) as the following two clauses:

<i>Logical List Append</i>	
$append(\mathbf{nil}, V, V)$	
$append(A \cdot U, V, A \cdot W)$	$:- append(U, V, W)$

Here, the symbol $:-$ has the declarative meaning "is implied by"; operationally such an implication is used to replace a goal of the same form as the left-hand side with the corresponding right-hand side subgoals. For example, given a goal

$$append(a \cdot b \cdot \mathbf{nil}, c \cdot d \cdot e \cdot \mathbf{nil}, Z),$$

this program generates the subgoals

$$append(b \cdot \mathbf{nil}, c \cdot d \cdot e \cdot \mathbf{nil}, Y)$$

$$append(\mathbf{nil}, c \cdot d \cdot e \cdot \mathbf{nil}, X),$$

and then returns the answer $Z = a \cdot Y = a \cdot b \cdot X = a \cdot b \cdot c \cdot d \cdot e \cdot \mathbf{nil}$.

*This research was supported in part by the National Science Foundation under Grant MCS 83-07755.

**On leave from the University of Illinois.

In previous research [Dershowitz-84], we have investigated the use of unconditional rewrite systems for logic programming. A *rewrite system* (see [Huet-Oppen-80]) is a set of *directed* equations (or equivalences) used as a *nondeterministic* pattern-directed program that returns as output a simplified term equal to a given input term. For example, the following is a three-rule rewrite system for *append*:

<i>List Append</i>		
$append(\mathbf{nil}, V)$	\rightarrow	V
$append(U, \mathbf{nil})$	\rightarrow	U
$append(A \cdot U, V)$	\rightarrow	$A \cdot append(U, V)$

Rules may be applied in any order to any matching subterm until no further applications are possible. Thus, applying the rules to the term

$$append(a \cdot \mathbf{nil}, \mathbf{nil}),$$

one gets either the rewrite sequence

$$append(a \cdot \mathbf{nil}, \mathbf{nil}) \Rightarrow a \cdot append(\mathbf{nil}, \mathbf{nil}) \Rightarrow a \cdot \mathbf{nil},$$

or, using only the second rule,

$$append(a \cdot \mathbf{nil}, \mathbf{nil}) \Rightarrow a \cdot \mathbf{nil}.$$

The same rules are also used to solve goals by a process called "narrowing". If the left-hand side of a rule unifies with any subterm of a goal, then the goal is *narrowed* by applying the unifying substitution to the goal and then applying the rule to rewrite that subterm. (For the use of narrowing for computation, see [Goguen-Meseguer-84, Dershowitz-84, Reddy-85, Retz, et al.-85].)

In this paper, we explore the possibility of using conditional equations to provide a uniform framework for combining LISP-like applicative programming with PROLOG-like logic programming. A *conditional equation* is a formula of the form

$$p \supset l = r,$$

meaning that the term l is equal to the term r when the condition p holds. In general, there may be variables X, Y , etc. in p, l , and/or r , in which case the conditional equation is meant to hold for *all* terms X, Y , etc. If any term containing an instance of l is "less defined" than the corresponding term with r in place of l , then the conditional equation may be *directed*. (See Section 5.1.) A directed equation is used to "substitute equals for equals" only from left to right, and we write it as a *conditional rule*:

$$l \text{ :- } p \rightarrow r.$$

Either (or both) of the rule parts $\text{:-}p$ and $\rightarrow r$ may be omitted, in which case it is taken to be **true**. (Variables on the right-hand side should also appear in one of the other two parts, and no left-hand side should be the term **true**.) A rule with only a left-hand side l is called an *assertion*; one with no condition p is a *rewrite rule*; one with no right-hand side r is a *logic rule*.

A (*conditional*) *rewrite program* is a system of such conditional, directed equations. Each equation may be used in two distinct ways: it can be used to *simplify* a subterm that matches its left-hand side, and it can be used to *narrow* a subterm that unifies with its left-hand side. Thus, a rewrite system can be used to compute by repeatedly substituting equal terms in a given term, until the simplest form possible is obtained. A system can also be used to compute by deriving consequences of given equations until the desired output values are obtained. As we will see, simplifications are irrevocable, while narrowings are provisional.

In the next section we consider applicative programming using equations for simplification and rewriting, and in the section that follows it we consider logic programming using equations for unification and narrowing. Section 4 shows some of the benefits that can be obtained by combining both programming forms in one program. Correctness issues are addressed in Section 5 and implementation issues in Section 6. They are followed by a brief comparison with related work.

2. APPLICATIVE PROGRAMMING

A (*conditional*) *rewrite system* \mathcal{R} is a finite set of rewrite rules, each of the form

$$l[\bar{X}] \text{ :- } p[\bar{X}] \rightarrow r[\bar{X}],$$

where l and r are terms and p is a predicate. Terms in general contain variables (these are the \bar{X}), but the *right-hand side* r and the *condition* p should only contain variables that appear in the *left-hand side* l . (This restriction will be relaxed in the next section.) Such a rule may be applied to a term t if a subterm s of t *matches* (by "one-sided" unification) the left-hand side l with some substitution σ of terms for the variables in l , and if the corresponding condition $p\sigma$ is true, where $p\sigma$ denotes the term p after making the substitution σ for its variables. The rule is applied by replacing the subterm $s\sigma=l\sigma$ in t with the right-hand side $r\sigma$. The

choice of which rule to apply where is made nondeterministically from amongst all possibilities. We write $t \Rightarrow t'$ to indicate that a term t' is *derivable* from the term t by a single application of some rule in \mathcal{R} . When we said above that $p\sigma$ must be true for the rule to be applied, we meant $p\sigma \Rightarrow \dots \Rightarrow \mathbf{true}$, i.e. that $p\sigma$ reduces to the constant **true** via zero or more rule applications. There is no backtracking over reductions. (There is backtracking over deductions, as we will see in the next section.)

An applicative program definition of the form

$$f(\bar{X}) \Leftarrow \text{if } p[\bar{X}] \text{ then } r[\bar{X}] \text{ else } s[\bar{X}]$$

can be translated into an unconditional system

$$\begin{aligned} f(\bar{X}) &\rightarrow f'(p[\bar{X}], \bar{X}) \\ f'(\mathbf{true}, \bar{X}) &\rightarrow r[\bar{X}] \\ f'(\mathbf{false}, \bar{X}) &\rightarrow s[\bar{X}], \end{aligned}$$

where f' is a new function symbol. This has the effect of ensuring that the condition is evaluated only once before either branch is explored. If the condition $p[\bar{X}]$ evaluates to **true** for the given values of \bar{X} , the term $f(\bar{X})$ gets eventually replaced by $r[\bar{X}]$; if $p[\bar{X}]$ evaluates to **false**, then $f(\bar{X})$ is replaced by $s[\bar{X}]$. The one-line rule

$$f(\bar{X}) \rightarrow \text{if } p[\bar{X}] \text{ then } r[\bar{X}] \text{ else } s[\bar{X}]$$

may be considered, then, as an abbreviation for the above three rules.

An alternative rewrite program for conditional expressions, using conditional rules and no new symbols, would be

$$\begin{aligned} f(\bar{X}) &::= p[\bar{X}] &\rightarrow r[\bar{X}] \\ f(\bar{X}) &::= \mathbf{not}(p[\bar{X}]) &\rightarrow s[\bar{X}], \end{aligned}$$

where **not(false)** evaluates to **true**. Often, conditions may instead be expressed as left-hand side patterns. Consider, for example, the following program for computing the union of two sets (of numbers, say) represented as lists without repetitions. That is, given two lists X and Y , it returns a list $\mathit{union}(X, Y)$, containing those elements that appear in at least one of the input lists. The program is

<i>List Union</i>	
$\mathit{union}(\mathbf{nil}, Y)$	$\rightarrow Y$
$\mathit{union}(X, \mathbf{nil})$	$\rightarrow X$
$\mathit{union}(A \cdot X, Y)$	$::= \mathit{member}(A, Y)$
	$\rightarrow \mathit{union}(X, Y)$
$\mathit{union}(A \cdot X, Y)$	$::= \mathbf{not}(\mathit{member}(A, Y))$
	$\rightarrow A \cdot (\mathit{union}(X, Y))$
$\mathit{member}(A, \mathbf{nil})$	$\rightarrow \mathbf{false}$
$\mathit{member}(A, A \cdot Y)$	$\rightarrow \mathbf{true}$
$\mathit{member}(A, B \cdot Y)$	$::= \mathbf{not}(A=B)$
	$\rightarrow \mathit{member}(A, Y)$
$I = J$	$::= \mathbf{number}(I, J)$
	$\rightarrow \mathbf{eq}(I, J)$

where union is the function being defined, member is an auxiliary predicate testing for membership of an element in a list, **eq** is a built-in predicate that tests for equality of numbers, and **number** is a built-in predicate that returns **true** if all its argument are numbers (and **false** otherwise). In place of the logic rule

$$\mathit{member}(A, Y) ::= A = \mathbf{car}(Y),$$

this program has the pattern-directed assertion

$$\mathit{member}(A, A \cdot Y).$$

The condition **number** is necessary in

$$I = J ::= \mathbf{number}(I, J) \rightarrow \mathbf{eq}(I, J),$$

since **eq** is only intended to work for built-in data-types. Note that we must have the **false** case for member for the fourth union rule to work, since negation is not being handled "by failure" (cf. [Clark-78]).

3. LOGIC PROGRAMMING

Rewrite systems may be used as "logic programs" [Kowalski-74], in addition to their straightforward use for computation by rewriting, illustrated in the previous section. The programming paradigm described below allows for the advantageous combination of both computing modes. The result is a PROLOG-like programming language, the main differences being that rewrite rules are conditional equivalences, rather than implications in Horn-clause form. Any (pure) PROLOG statement may be directly translated into a rewrite rule: the clause

$$p :- q, r$$

corresponds to the identical rule. A rule of the form

$$p :- q \rightarrow r$$

is stronger than the above Horn clause and means that $q \supset (p \equiv r)$. A rule like

$$p \rightarrow q \& r$$

is even stronger; it has p true if, and only if, q and r both hold.

The following, for example, is a program $div(a,b,q,r)$ to compute the quotient q and remainder r of nonnegative integer a and positive integer b :

Integer Division	
$div(X,Y,Q+1,R)$	$:- X \geq Y$ $\rightarrow div(X-Y,Y,Q,R)$
$div(X,Y,0,X)$	$:- Y > X$
$div(X,Y,0,R)$	$:- X \geq Y$ $\rightarrow \text{false}$
$I > J$	$:- \text{number}(I,J)$ $\rightarrow \text{greater}(I,J)$
$I \geq J$	$:- \text{number}(I,J)$ $\rightarrow \text{not}(\text{less}(J,I))$
$I - J$	$:- \text{number}(I,J)$ $\rightarrow \text{diff}(I,J)$

The first rule is the recursive case; the second is the base case; the third covers false cases; the remainder apply built-in functions to numbers.

The *resolution procedure*, developed by J. A. Robinson [Robinson-63] in the early 1960's, derives consequences of logical formulas written in "clausal" form. Resolving a clause of the form

$$p \vee q$$

with another clause

$$\neg r \vee s,$$

when p and r are unifiable by σ results in the new clause

$$q\sigma \vee s\sigma.$$

The *completion procedure*, developed by D. E. Knuth and P. Bendix [Knuth-Bendix-70] in the late 1960's, was introduced as a means of deriving canonical term-rewriting systems to serve as decision procedures for given unconditional equational theories. More recently, it has been applied to other aspects of equational

reasoning (see, for example, [Dershowitz-82b]), and in [Dershowitz-85, Dershowitz-Josephson-84] it has been applied to logic programming, as well. Completion is, in a sense, an extension of resolution in that it allows unification at subterms. That is, a rule

$$l \rightarrow r$$

may be *overlapped* on a rule of the form

$$u[s] \rightarrow t,$$

whose left-hand side contains a subterm s that is unifiable with l via substitution σ . The result is one of the two rules:

$$u[r]\sigma \rightarrow t\sigma$$

or

$$t\sigma \rightarrow u[r]\sigma.$$

Which orientation is chosen depends on a well-founded ordering $>$ supplied to the procedure. If $u[r]\sigma > t\sigma$ in that ordering, the former is chosen; if $t\sigma > u[r]\sigma$, the latter is. (If the ordering is partial, it may be that neither orientation works.) *Narrowing* [Slagle-74] is a "linear" restriction on completion, analogous to "linear input resolution". That is, program rules $l \rightarrow r$ are only overlapped on goal rules, not program rules on program rules, nor goal rules on goal rules. The orientation of subgoal rules is fixed, so no ordering is needed.

Formally, a *rewrite program* is a set of rewrite rules of the form

$$l[\bar{X}] :- p[\bar{X},\bar{Y}] \rightarrow r[\bar{X},\bar{Y}],$$

where the condition may contain variables \bar{Y} not also on the left-hand side. To execute a rewrite program, we must adapt the narrowing process to rules with conditions; we call this adaptation *conditional narrowing*. In the next few paragraphs, we describe the details of rewrite program interpretation.

To begin a computation with a rewrite program, a *goal rule* is added to the system. Goal rules are of the form

$$g[\bar{x},\bar{Z}] \rightarrow \text{answer}(\bar{Z}),$$

where g is the *calling term* containing input values (i.e. irreducible ground terms) \bar{x} and output variables \bar{Z} , and *answer* is the predicate symbol that will store the result. PROLOG goals of the form

$$:- q, r$$

correspond to goal rules

$$\mathbf{true} \text{ :- } q, r \rightarrow \mathit{answer}(\bar{Z}),$$

where \bar{Z} are the variables in q and r .

At each point in the computation, the current subgoal is of the general form

$$h \text{ :- } q_1, \dots, q_n \rightarrow \mathit{answer}(\bar{s}),$$

meaning that the answer is \bar{s} if the subgoals q_1, \dots, q_n , and h are achieved (in that order). Given such a subgoal, and a rule

$$l \text{ :- } p \rightarrow r$$

whose left-hand side l can be unified with a (nonvariable) subterm of q_1 via most general unifier σ , i.e. $q_1\sigma = t[l\sigma]$ for some context t , the subgoal is *conditionally narrowed* to

$$h\sigma \text{ :- } p\sigma, t[r\sigma], q_2\sigma, \dots, q_n\sigma \rightarrow \mathit{answer}(\bar{s}\sigma).$$

At each such step, all possible simplifications (as in the previous section) are applied throughout. That is, if a left-hand side *matches* any subterm of a subgoal, that subterm is reduced. (Recall that for a conditional rule to apply, the condition must reduce to **true**. If the condition reduces to anything else, the rule is not applied. If it reduces to a term containing variables, rather than to **true** or **false**, then the potential simplification will be considered as a possible narrowing.) Simplifying gives a new subgoal

$$h' \text{ :- } p', q_1', q_2', \dots, q_n' \rightarrow \mathit{answer}(\bar{s}').$$

where h', p', q_1' , etc. are all irreducible. If any of these conditions reduced to the term **false**, then the whole subgoal may be abandoned. Only when all the conditions become **true**, and the subgoal is of the unconditional form

$$h' \rightarrow \mathit{answer}(\bar{s}'),$$

are narrowing unifications attempted within h' . Computation ends when a *solution rule*

$$\mathbf{true} \rightarrow \mathit{answer}(\bar{t})$$

is generated, giving an answer \bar{t} such that

$$g[\bar{x}, \bar{t}]$$

holds. Since, in general, there may be many ways to achieve a subgoal, alternative narrowing computations must be attempted, either in parallel (until one succeeds) or sequentially (by backtracking upon failure).

Conditions, when separated by commas, are executed from left-to-right, and must all be true before the left-hand side is replaced by the right-hand side. Conditions separated by the symbol \mathcal{E} , on the other hand, may be executed in any order. Such commas are just "syntactic sugar" in that a rule

$$l \text{ :- } p, q \rightarrow r$$

can always be replaced by two rules,

$$\begin{aligned} l \text{ :- } p &\rightarrow \mathit{if}(q, r) \\ \mathit{if}(Q, R) \text{ :- } Q &\rightarrow R, \end{aligned}$$

having only one condition each.

To compute, for example, the quotient and remainder of two nonnegative numbers a and b with the above program, the rule

$$\mathit{div}(a, b, Q, R) \rightarrow \mathit{answer}(Q, R)$$

is added, meaning that Q and R are the answer if and only if they are the quotient and remainder, respectively, of a and b . The interpreter then generates a rule

$$\mathbf{true} \rightarrow \mathit{answer}(c, d),$$

containing the answer values c and d for Q and R . For example, to compute the quotient and remainder of 7 and 3, the rule

$$\mathit{div}(7, 3, Q, R) \rightarrow \mathit{answer}(Q, R)$$

is added. Narrowing generates

$$\mathit{div}(7-3, 3, U, R) \rightarrow \mathit{answer}(U+1, R),$$

by applying the first program rule, which simplifies to

$$\mathit{div}(4, 3, U, R) \rightarrow \mathit{answer}(U+1, R),$$

applying the last rule for built-in subtraction. Using the first rule again gives

$$\mathit{div}(1, 3, V, R) \rightarrow \mathit{answer}(V+1+1, R).$$

Now the second rule yields the answer

$$\mathbf{true} \rightarrow \mathit{answer}(0+1+1, 1).$$

Note that this program can test whether or not two numbers have the given quotient and remainder. It is not, however, in a form that would allow computing the first argument, say, from the other three, unless the built-in **number**(I, J) generates all instances of I and J that are numbers. A goal like

$$\text{div}(X,3,2,1) \rightarrow \text{answer}(X)$$

generates the subgoal

$$\text{div}(X-3,3,2,1) \text{ :- } X \geq 3 \rightarrow \text{answer}(X),$$

but we gave no rules for reducing $X-3$ or solving $X \geq 3$ when X is not a number satisfying $\text{number}(X,3)$. (See Section 6.1.)

4. FEATURES

The two paradigms of computation illustrated in the preceding sections, viz. simplification and narrowing, can be combined in a single rewrite program. Every narrowing step is followed by as much simplification as possible. Simplification steps employ pattern matching, while narrowing involves unification. Simplifications are irrevocable, while narrowing steps are subject to backtracking.

As an example of the utility of applying rules at more than one level of a goal, consider the following situation:

<i>List Generator</i>	
$\text{listp}(\text{nil})$	
$\text{listp}(A \cdot Y)$	$\rightarrow \text{listp}(Y)$
$\text{length}(\text{nil})$	$\rightarrow 0$
$\text{length}(A \cdot Y)$	$\rightarrow \text{length}(Y) + 1$
$I < J$	$\text{:- } \text{number}(I, J)$
	$\rightarrow \text{less}(I, J)$
$I + 1 < J$	$\text{:- } \text{number}(J)$
	$\rightarrow I < \text{sub1}(J)$
$I < 0$	$\rightarrow \text{false}$
$P \& \text{false}$	$\rightarrow \text{false}$

A subgoal $\text{listp}(Z)$ can, by repeating the second rule, generate arbitrarily large lists $Z = A_1 \cdot A_2 \cdot \dots \cdot A_n \cdot \text{nil}$. But when combined with a test for length, as in

$$\text{listp}(Z) \& \text{length}(Z) < 10,$$

it will *reduce* to **false** after ten narrowings. At that point the second subgoal becomes **false**, thereby pruning an otherwise potentially infinite computation path.

4.1. Solving equations

The applicative *union* program in Section 2 can be used, for example, in a logic program to find a list Z

and element A such that

$$\{1\} \cup Z = \{A, 1\} \cup \{2, 3\}.$$

That goal can be expressed as

$$\text{union}(1 \cdot \text{nil}, Z) = \text{union}(A \cdot 1 \cdot \text{nil}, 2 \cdot 3 \cdot \text{nil}) \& \text{not}(A=1) \rightarrow \text{answer}(A, Z),$$

where the condition $\text{not}(A=1)$ is needed to ensure that the list $A \cdot 1 \cdot \text{nil}$ is a proper encoding of a set. To solve equations, we will also need an additional assertion

$$U = U.$$

The computation could then proceed as follows: Beginning with the goal

$$\text{union}(1 \cdot \text{nil}, Z) = \text{union}(A \cdot 1 \cdot \text{nil}, 2 \cdot 3 \cdot \text{nil}) \& \text{not}(A=1) \rightarrow \text{answer}(A, Z),$$

the third rule for *union* is applied, giving

$$\begin{aligned} \text{union}(1 \cdot \text{nil}, Z) &= 1 \cdot 2 \cdot 3 \cdot \text{nil} \& \text{not}(A=1) \\ \text{:- } \text{member}(A, 2 \cdot 3 \cdot \text{nil}) &\rightarrow \text{answer}(A, Z). \end{aligned}$$

(The first rule for *union* cannot be used, since nil does not unify with $1 \cdot \text{nil}$; the second rule makes $Z = \text{nil}$, but then the equality fails; the fourth rule leads to other solutions.) Now the condition $\text{member}(A, 2 \cdot 3 \cdot \text{nil})$ needs to be solved before anything else. One way to solve it is by letting $A=2$, yielding

$$\text{union}(1 \cdot \text{nil}, Z) = 1 \cdot 2 \cdot 3 \cdot \text{nil} \rightarrow \text{answer}(2, Z)$$

after simplification. Using the third rule again, gives

$$Z = 1 \cdot 2 \cdot 3 \cdot \text{nil} \text{ :- } \text{member}(1, Z) \rightarrow \text{answer}(2, Z).$$

If $Z = 1 \cdot Y$, the condition is satisfied and it remains to solve

$$1 \cdot Y = 1 \cdot 2 \cdot 3 \cdot \text{nil} \rightarrow \text{answer}(2, 1 \cdot Y).$$

Unifying the two sides of the equality, using the assertion $U=U$, solves the original goal:

$$\text{true} \rightarrow \text{answer}(2, 2 \cdot 3 \cdot \text{nil}).$$

This computation yields as an answer, $A=2$ and $Z=2 \cdot 3 \cdot \text{nil}$, one solution out of many.

The *union* program can also be extended with rules like

$$\begin{aligned} \text{member}(A, \text{union}(X, Y)) \\ \rightarrow \text{member}(A, X) \vee \text{member}(A, Y) \end{aligned}$$

to help find, say, an X such that A is (or is not) a member of $\text{union}(X, Y)$, given A and Y .

4.2. Assignment

The conditional part can be used for generalized assignment (subsuming `setq` in LISP and `is` in PROLOG) in the following manner:

Insertion Sort	
<code>sort(nil)</code>	\rightarrow <code>nil</code>
<code>sort(A·nil)</code>	\rightarrow <code>A·nil</code>
<code>sort(A·Y)</code>	$\text{:- } Z \doteq \text{sort}(Y)$ \rightarrow <code>insert(A,Z)</code>
<code>insert(A,nil)</code>	\rightarrow <code>A·nil</code>
<code>insert(A,B·Z)</code>	$\text{:- not(greater(A,B))}$ \rightarrow <code>A·B·Z</code>
<code>insert(A,B·Z)</code>	:- not(less(A,B)) \rightarrow <code>B·insert(A,Z)</code>
<code>nil</code>	\doteq <code>nil</code>
<code>A·Z</code>	\doteq <code>A·Z</code>
<code>nil</code>	\doteq <code>A·Z</code> \rightarrow <code>false</code>
<code>A·Z</code>	\doteq <code>nil</code> \rightarrow <code>false</code>

The purpose of the condition $Z \doteq \text{sort}(Y)$ is to assign the sorted list to Z . Only when $\text{sort}(Y)$ is partially evaluated to a list of the form $A·Y$ can the rules for \doteq be applied; the term $\text{sort}(Y)$ itself cannot be assigned to Z , as would be the case were $=$ used. (This has an effect similar to that of the "read-only" function in Concurrent Prolog [Shapiro-83]. In general, one would want to have built-in assignment rules of this form for each built-in data-type.)

4.3. Functionality and negation

The two main uses of "cuts" in PROLOG are to avoid backtracking in the presence of "functional dependencies" and to handle negation (see [Clocksin-Mellish-84]). With rewrite rules, functions can be handled directly by simplification which does not allow for backtracking. Thus, a goal of the form

$$p(f(X),Z) \rightarrow \text{answer}(Z)$$

will first have $f(X)$ simplified, before continuing with the subgoal p . If p then fails, the computation of $f(X)$ is not undone.

The second use of "cuts", negation, can be handled within the logical framework, since programs can evaluate false cases as well as true ones. For example, the following program computes the first prime numbers up to (but not including) N :

Prime Number Generator	
<code>prime(N)</code>	\rightarrow <code>sift(integers(2,N))</code>
<code>integers(K,N)</code>	$\text{:- less}(K,N)$ \rightarrow <code>K·integers(K+1,N)</code>
<code>integers(K,N)</code>	$\text{:- not(less}(K,N))$ \rightarrow <code>nil</code>
<code>sift(N·L)</code>	\rightarrow <code>N·sift(filter(N,L))</code>
<code>sift(nil)</code>	\rightarrow <code>nil</code>
<code>filter(M,nil)</code>	\rightarrow <code>nil</code>
<code>filter(M,N·L)</code>	$\text{:- divides}(M,N)$ \rightarrow <code>filter(M,L)</code>
<code>filter(M,N·L)</code>	$\text{:- not(divides}(M,N))$ \rightarrow <code>N·filter(M,L)</code>
<code>divides(M,N)</code>	$\text{:- div}(N,M,Q,0)$

where div is the division program given earlier. Since $\text{divides}(M,N)$ returns **false** when a does not divide N , the program can test for $\text{not}(\text{divides}(M,N))$ without recourse to a new predicate "not-divides" or to "negation by failure".

Any "closed-world" assumption can be made explicit, using the "if-and-only-if" meaning of \rightarrow , as in the following example:

Adam and Eve	
<code>female(X)</code>	$\text{:- person}(X)$ \rightarrow $\neg \text{male}(X)$
<code>male(X)</code>	$\text{:- person}(X)$ \rightarrow <code>X = adam</code>
<code>person(adam)</code> <code>person(eve)</code>	
<code>X = Y</code>	$\text{:- person}(X) \& \text{person}(Y)$ \rightarrow <code>eq(X,Y)</code>
<code>true & P</code>	\rightarrow <code>P</code>
<code>P & true</code>	\rightarrow <code>P</code>
	$\neg \text{false}$

Here, \neg is *not* the built-in negation function and `eq` works for people, too. The goal `female(Y)` results in the following computation:

$$\begin{aligned} \text{female}(Y) &\rightarrow \text{answer}(Y) \\ \neg \text{male}(Y) &\text{:- } \text{person}(Y) \\ &\rightarrow \text{answer}(Y) \\ \text{true} &\rightarrow \text{answer}(\text{eve}) \end{aligned}$$

This works since it is explicitly given that a person is

female if and only she is non-male, and that *adam* is the only male in the (primeval) world. The first step replaces *female*(*Y*) with \neg *male*(*Y*). Then the condition *person*(*Y*) is solved. Letting $Y=adam$ fails, since $\neg male(adam)$ reduces to **false**; letting $Y=eve$ succeeds.

It is this ability to express logical negation, and say that some goal is **false**, which allows for the pruning of fruitless paths. The following example illustrates this:

<i>Slow Sort</i>	
<i>sort</i> (nil)	\rightarrow nil
<i>sort</i> (<i>Y</i>)	\vdash <i>perm</i> (<i>Y,Z</i>), <i>ordered</i> (<i>Z</i>) \rightarrow <i>Z</i>
<i>ordered</i> (nil)	
<i>ordered</i> (<i>A</i> · nil)	
<i>ordered</i> (<i>A</i> · <i>B</i> · <i>Z</i>)	\vdash not (<i>less</i> (<i>B,A</i>)) \rightarrow <i>ordered</i> (<i>B</i> · <i>Z</i>)
<i>ordered</i> (<i>A</i> · <i>B</i> · <i>Z</i>)	\vdash <i>less</i> (<i>B,A</i>) \rightarrow false
<i>perm</i> (<i>Y,B</i> · <i>Z</i>)	\vdash <i>append</i> (<i>U,B</i> · <i>V</i>) \doteq <i>Y</i> \rightarrow <i>perm</i> (<i>append</i> (<i>U,V</i>), <i>Z</i>)
$A \cdot Y \doteq B \cdot Z$	\rightarrow $A=B \ \& \ Y=Z$
$A \cdot Y \doteq \text{nil}$	\rightarrow false

which uses the previous program for *append*. Any permutation being generated by *perm* is pruned by *ordered* as soon as it contains two inverted elements. The definition for \doteq prunes impossibly long instances of *append*(*U,B*·*V*).

A more general logical facility than negation that works for all boolean combinations of predicates is provided by the following rewrite system:

<i>Propositional Calculus</i>	
$\neg U$	\rightarrow $U = \text{false}$
$U \vee V$	\rightarrow $(U \& V) = U = V$
$U \supset V$	\rightarrow $(U \& V) = U$
$U \& \text{true}$	\rightarrow U
$U \& \text{false}$	\rightarrow false
$U \& U$	\rightarrow U
$U = \text{true}$	\rightarrow U
$U = U$	\rightarrow true
$(U=V) \& W$	\rightarrow $(U \& W) = (V \& W) = W$

Using these rules requires associative-commutative unification [Stickel-81, Fages-83] for $\&$ and $=$ (equivalence). The advantage is that any propositional

formula has a *unique* irreducible form. Propositionally valid formulae reduce to **true**; propositionally unsatisfiable ones reduce to **false**. (See [Dershowitz,etal.-83])

Note that negation makes the ordering of subgoals less crucial, since pruning can be used to guarantee termination.

4.4. Streams

The following modification of the prime number program illustrates the use of "streams":

<i>Prime Number Stream</i>	
<i>prefix</i> (<i>N</i> · <i>K,N</i> · <i>L</i>)	\rightarrow <i>prefix</i> (<i>K,L</i>)
<i>prefix</i> (nil , <i>L</i>)	
<i>integers</i> (<i>K</i>)	\rightarrow <i>K</i> · <i>integers'</i> (<i>K</i> +1)
<i>sift</i> (<i>N</i> · <i>L</i>)	\rightarrow <i>N</i> · <i>sift'</i> (<i>filter</i> (<i>N,L</i>))
<i>filter</i> (<i>M,N</i> · <i>L</i>)	\vdash <i>divides</i> (<i>M,N</i>) \rightarrow <i>filter</i> (<i>M,L</i>)
<i>filter</i> (<i>M,N</i> · <i>L</i>)	\vdash not (<i>divides</i> (<i>M,N</i>)) \rightarrow <i>N</i> · <i>filter'</i> (<i>M,L</i>)
<i>divides</i> (<i>M,N</i>)	\vdash <i>div</i> (<i>N,M,Q,0</i>)
<i>filter</i> (<i>M,integers'</i> (<i>K</i>))	\rightarrow <i>filter</i> (<i>M,integers</i> (<i>K</i>))
<i>sift</i> (<i>filter'</i> (<i>N,L</i>))	\rightarrow <i>sift</i> (<i>filter</i> (<i>N,L</i>))
<i>filter</i> (<i>M,filter'</i> (<i>N,L</i>))	\rightarrow <i>filter</i> (<i>M,filter</i> (<i>N,L</i>))
<i>prefix</i> (<i>Z,sift'</i> (<i>L</i>))	\rightarrow <i>prefix</i> (<i>Z,sift</i> (<i>L</i>))

Invoking

$$prefix(Z,sift(integers(2))) \rightarrow answer(Z)$$

generates arbitrarily long sequences of prime numbers. Notice how new (primed) function names (e.g. *integers'*) are used for otherwise infinite "streams"; a "call-by-need" effect is obtained with the last set of rules for reinstating the original (unprimed) function names. (Cf. [Tamaki-Sato-83].) In this way, simplification will always terminate for correct programs; any possible nontermination should be confined to the narrowing process.

5. CORRECTNESS

In this section we look at what it means for rewrite programs to be correct. Some related issues are considered in [Remy-Zhang-84, Kaplan-84, Bergstra-Klop-82].

5.1. Termination

As mentioned earlier, a correct rewrite system must not allow an infinite sequence of simplifications. Formally, we require for each rule

$$l \text{ :- } p \rightarrow r$$

we have

$$p \text{ implies } l \succ r$$

and

$$l \succ p$$

for some well-founded ordering \succ . The ordering \succ must be *monotonic* in the sense that reducing a subterm reduces any superterm containing it, i.e.

$$s \succ t \text{ implies } u[s\sigma] \succ u[t\sigma]$$

for all terms s and t , all contexts u , and all substitutions σ . See [Dershowitz-85] for a survey of orderings for term-rewriting systems.

This requirement does not, however, preclude there being an infinite *narrowing* sequence. For example, the program

$$\begin{aligned} & \text{listp}(\text{nil}) \\ \text{listp}(A \cdot Y) & \text{ :- listp}(Y), \end{aligned}$$

though terminating whenever used to simplify, goes on forever generating solutions for the goal

$$\text{listp}(Z).$$

5.2. Completeness

An interpreter is said to be *complete* for a logic-programming language, if for every *logically* satisfiable goal there is a successful computation path. Analogous to the unconditional case in [Dershowitz-84], one can show that if there exists a solution to a goal whose correctness follows *equationally* (substituting "equals for equals") from a terminating system \mathcal{R} , then the interpreter will find a solution (by narrowing and simplification). More precisely, given a term g and

ground substitution σ , if $g\sigma$ reduces to **true** (and to no other irreducible term), then there is a *narrowing* sequence such that the goal

$$g \rightarrow \text{answer}$$

generates an answer

$$\text{true} \rightarrow \text{answer}\mu,$$

where μ is a substitution that is at least as general as σ . (For a similar result, see [Kaplan-84]. If conditions contain new variables, they can be existentially quantified away.)

It remains to determine under what conditions any logically satisfiable goal is also equationally satisfiable. A terminating rewrite system is said to be *ground confluent* if each ground term reduces to a *unique* irreducible term. Terminating LISP-like programs (with mutually exclusive conditions that do not introduce terms with new variables) are ground confluent, as are PROLOG-like programs (with only the term **true** for right-hand sides). For "syntactic" methods of demonstrating confluence of conditional systems, see [Kaplan-84]; for "semantic" methods, see [Plaisted-85]. We say that a theory is *disjunctively complete* for a rewrite system \mathcal{R} if for all valid disjunctions $C_1 \vee \dots \vee C_n$ of instances of conditions appearing in rules, at least one of the C_i is itself valid. In particular, this condition holds if the theory has an initial model and all conditions are non-negated literals. It also holds if all ground instances of conditions are provably true or provably false. We can show that if \mathcal{R} is sound, ground confluent, terminating, disjunctively complete, and equalities appear only in purely *conjunctive* conditions, then there is a successful narrowing for any satisfiable goal. These conditions are satisfied by correct LISP-like and PROLOG-like programs.

6. IMPLEMENTATION

Currently, we have an experimental implementation in FRANZ LISP [Foderaro, *et al.*-84], based on the facilities of the RRL [Kapur-Sivakumar-83] rewrite-system environment. In this section, we briefly touch on some of the engineering issues that are being addressed.

6.1. Built-in functions

With full functional notation, one can make full use of built-in functions (cf. [Futatsugi,etal.-84]) An example is the following binary-search program:

<i>Binary Search</i>	
$bin(Z,X,Z,1)$	$\rightarrow answer(Z)$
$bin(P,X,Z,Y)$	$:- less(X, f(mid(Z,Y)))$ $\rightarrow bin(P,X,Z, half(Y))$
$bin(P,X,Z,Y)$	$:- even(Y), not(less(Z, f(mid(Z,Y))))$ $\rightarrow bin(P,X, mid(Z,Y), half(Y))$
$bin(P,X,Z,Y)$	$:- odd(Y), not(less(Z, f(mid(Z,Y))))$ $\rightarrow bin(P,X, mid(Z,Y), half(Y)+1)$
$mid(I,J)$	$:- number(I,J)$ $\rightarrow plus(I, half(J))$
$half(J)$	$:- number(J)$ $\rightarrow quotient(J,2)$
$I+1$	$:- number(I)$ $\rightarrow add1(I)$

Given a goal $bin(Z,X,A,N)$ and a nondecreasing function f —this program searches for a Z among the positions $A, A+1, \dots, A+N-1$, such that $f(Z) \leq X < f(Z+1)$. To use this program, a program for computing f must also be provided.

Since there is no backtracking over simplifications, arrays can be handled cheaply by destructive assignments. For example, in applying the rules

```
zap(A,I,N) :- array(A), less(I,N)
            → zap(assign(A,I,0),add1(I),N)
zap(A,N,N) :- array(A)
            → assign(A,N,0)
```

to the term

$zap(a,0,9)$,

there is no need to preserve old values of a , since, after rewriting, a will only appear within the built-in array assignment function.

Built-in functions are not narrowable, i.e. they only work when given constructor arguments. Note that one can mix the use of built-in functions (when the arguments are fully evaluated) and defined functions (that can be applied even to nonground terms) as in:

```
0 + J → J
I + 0 → I
I + J :- number(I,J)
        → plus(I,J)
I + J = K :- number(J,K)
           → I = diff(K,J)
I + J = K :- number(I,K)
           → J = diff(K,I)
```

6.2. Queuing

The effect of a single rule application on future applications is localized: the only *new* possibilities for applications are within the new subterm introduced by the right-hand side of a rule and at nearby, enclosing function symbols. Thus, in most cases only an area bounded by the size of the rules in the program needs to be examined at each step. This suggests maintaining a queue of positions at which rules can be applied, rather than searching through the whole term again and again. Congruence-closure algorithms [Nelson-Oppen-80] (which work on ground terms) go a step further and speed things up by remembering partial matches, obviating the need to re-examine enclosing function symbols.

There is a space-time tradeoff in deciding whether an implementation should save the results of previous simplifications. OBJ2 [Futatsugi,etal.-84], for example, maintains a hash-table of terms and their fully simplified forms.

6.3. Parallelism

Confluency guarantees that the order in which simplifications are applied is immaterial, making simplification of non-overlapping subterms a natural candidate for concurrent execution. Alternative narrowings, on the other hand, can lead to success or failure; to guarantee that an existing answer will be found (see Section 5.2) requires that no possible narrowing be "starved". Ensuring that by breadth-first search, however, would, in general, make heavy storage demands.

There are cases when certain potential narrowings are sure to be redundant and can be eliminated. Particularly with parallel execution, it would be desirable to prune such unproductive paths. By including rules for

false cases, as outlined above, goals that are *unsatisfiable* will not be pursued. By not just narrowing goals, but also comparing one with another, duplicate goals can be pruned. In particular, given two rules for some subgoal g , one of the unconditional form

$$g \rightarrow \text{answer}(\bar{s}),$$

and the other of the more restrictive form

$$g\sigma \text{ :- } p \rightarrow \text{answer}(\bar{t}),$$

where σ is any substitution, the latter rule can be ignored (assuming any solution suffices).

Similarly, using program rules to overlap assertions, as well as goals, provides a "forward reasoning" capability, generating new facts from old ones.

7. DISCUSSION

The approach outlined here is an attempt to combine features of applicative and logic programming in a unified way. The result is an extension of functional programming in that the given equations are also used for narrowing; thus, solutions to a given goal can be sought. Logic programming has been extended with a full functional notation, so that unification can apply at subterms, as well as at goal literals.

Various systems have been designed that add features of one paradigm to the other, e.g. HCPRVR [Chester-80], FPL [Bellia,etal.-82], QLOG [Komorowski-82], LOGLISP [Robinson-Sibert-82], [Kornfeld-83], LEAF [Barbutti,etal.-84], [Carlsson-84], [Lloyd-Topor-84], and AT1 [Newton 1985 AT1]. Other language proposals, which provide facilities for solving equations, include the following:

- HASL [Abramson-84], which adds unification to a lazy pattern-directed functional language;
- [Fribourg-84], which uses Horn-clauses composed of directed equality literals applied at the innermost overlap;
- EQLOG [Goguen-Meseguer-84], which uses narrowing to solve equations, interleaving it with simplification;
- TABLOG [Malachi,etal.-84], which uses a nonclausal theorem prover and allows for equalities, but has no programmer-defined notion of "simplification";

- [Reddy-85], which uses "lazy narrowing", similar in effect to our "conditional narrowing", has restrictions on the form of left-hand sides, and makes no distinction between narrowing and simplification;
- FUNLOG [Subrahmanyam-You-84], in which terms are lazily evaluated before unification is attempted;
- [Tamaki-84], which simulates narrowing, in PROLOG, by decomposing terms (see also [Plaisted-Greenbaum-84]);
- FGL+LV [Lindstrom-85], in which terms are simplified before unification is attempted, but unifications are not backtrackable.

Our proposal differs from most of these in its insistence on user-defined, unbacktrackable, nonlazy simplification prior to each backtrackable narrowing step.

ACKNOWLEDGEMENT

We thank Alan Josephson for his suggestions and for his implementation.

REFERENCES

1. H. Abramson. "A prological definition of HASL, a purely functional language with unification based on conditional binding expressions," *New Generation Computing*, Vol. 2, No. 1 (1984), pp. 3-35.
2. R. Barbutti, M. Bellia, G. Levi and M. Martelli. "On the integration of logic programming and functional programming," *Proceedings of the IEEE International Symposium on Logic Programming*, Atlantic City, CA (February 1984), pp. 160-166.
3. M. Bellia, P. Degano and G. Levi. "Call by name semantics of a clause language with functions," In: *Logic Programming*, K. L. Clark and S.-A. Tärnlund, eds. Academic Press, New York, 1982.
4. J. A. Bergstra and J. W. Klop. "Conditional rewrite rules: Confluence and termination", Report IW 198/82 MEI, Mathematische Centrum, Amsterdam, 1982.

5. R. M. Burstall, D. B. MacQueen and D. T. Sannella. "HOPE: An experimental applicative language," *Conference Record of the 1980 LISP Conference*, Stanford, CA (1980), pp. 136-143.
6. M. Carlsson. "On implementing Prolog in functional programming," *Proceedings of the IEEE International Symposium on Logic Programming*, Atlantic City, CA (February 1984), pp. 259-264.
7. D. Chester. "HCPRVR: An interpreter for logic programs," *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford, CA (1980).
8. K. L. Clark. "Negation as failure," In: *Logic in Data Bases*, H. Gallaire and J. Minker, eds. Plenum Press, New York, 1978, pp. 292-322.
9. W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1984 (Second Edition).
10. N. Dershowitz. "Applications of the Knuth-Bendix completion procedure," *Proceedings of the Seminaire d'Informatique Theorique*, Paris, France (December 1982).
11. N. Dershowitz. "Equations as programming language," *Proceedings of the Fourth Jerusalem Conference on Information Technology* (May 1984), pp. 114-123.
12. N. Dershowitz. "Computing with rewrite systems," *Information and Control* (1985) (to appear).
13. N. Dershowitz. "Termination," *Proceedings of the International Conference on Rewriting Techniques and Applications*, Dijon, France (May 1985).
14. N. Dershowitz, J. Hsiang, N. A. Josephson and D. A. Plaisted. "Associative-commutative rewriting," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany (August 1983), pp. 940-944.
15. N. Dershowitz and N. A. Josephson. "Logic programming by completion," *Proceedings of the Second International Conference on Logic Programming*, Uppsala, Sweden (July 1984), pp. 313-320.
16. F. Fages. "Formes canoniques dans les algèbres booléennes, et application à la démonstration automatique en logique de premier ordre", Thèse, Université de Paris VI, Paris, France, June 1983.
17. J. K. Foderaro, K. L. Sklower and K. Layer. "The FRANZ LISP manual," In: *Unix Programmer's Manual: Supplementary Documents*, M. J. Karels and S. J. Leffler, eds. University of California, Berkeley, CA, 1984.
18. L. Fribourg. "Oriented equational clauses as a programming language," *Proceedings of the Eleventh EATCS Colloquium on Automata, Languages and Programming*, Antwerp, Belgium (July 1984).
19. K. Futatsugi, J. A. Goguen, J. P. Jouannaud and J. Meseguer. "Principles of OBJ2", Centre de Recherche en Informatique de Nancy, Nancy, France, 1984.
20. J. A. Goguen and J. Meseguer. "Equality, types, modules and (why not?) generics for logic programming," *Logic Programming*, Vol. 1, No. 2 (1984), pp. 179-210.
21. M. Gordon, R. Milner and C. Wadsworth. "Edinburgh LCF," *Lecture Notes in Computer Science*, Vol. 78 (1979).
22. C. M. Hoffmann and M. J. O'Donnell. "Programming with equations," *Transactions on Programming Languages and Systems*, Vol. 4, No. 1 (January 1982), pp. 83-112.
23. G. Huet and D. C. Oppen. "Equations and rewrite rules: A survey," In: *Formal Language Theory: Perspectives and Open Problems*, R. Book, ed. Academic Press, New York, 1980, pp. 349-405.
24. S. Kaplan. "Fair conditional term rewriting systems: Unification, termination and confluency", Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, November 1984.
25. D. Kapur and G. Sivakumar. "Experiments with and architecture of RRL, a rewrite rule laboratory," *Proceedings of an NSF Workshop on the Rewrite Rule Laboratory*, Schenectady, NY (September 1983), pp. 33-56 (Available as Report 84GEN008, General Electric Research and Development [April 1984]).
26. D. E. Knuth and P. B. Bendix. "Simple word problems in universal algebras," In: *Computational Problems in Abstract Algebra*, J. Leech, ed. Pergamon Press, 1970, pp. 263-297.
27. H. J. Komorowski. "QLOG—The programming environment for PROLOG in LISP," In: *Logic Programming*, K. L. Clark and S.-A. Tärnlund, eds. Academic Press, 1982, pp. 315-322.
28. W. Kornfeld. "Equality for Prolog," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany (August 1983), pp. 514-519.
29. R. A. Kowalski. "Predicate logic as programming language," *Proceedings of the IFIP Congress*, Amsterdam, The Netherlands (1974), pp. 569-574.

30. G. Lindstrom. "Functional programming and the logical variable," *Proceedings Annual ACM Symposium on Principles of Programming Languages* (January 1985).
31. J. Lloyd and R. Topor. "Making Prolog more expressive," *J. of Logic Programming*, Vol. 1, No. 3 (1984), pp. 225-240.
32. J. McCarthy, *et al.* "LISP 1 programmer's manual", Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, 1960.
33. Y. Malachi, Z. Manna and R. J. Waldinger. "TABLOG: The deductive tableau programming language," *Proceedings of the ACM Lisp and Functional Programming Conference* (1984), pp. 323-330.
34. C. G. Nelson and D. C. Oppen. "Fast decision procedures based on congruence closure," *J. of the Association for Computing Machinery*, Vol. 27, No. 2 (1980), pp. 356-364.
35. M. O. Newton. "A combined logical and functional programming language", Technical Report 5172:TR:85, Computer Science, California Institute of Technology, Pasadena, CA, 1985.
36. D. A. Plaisted. "Semantic confluence tests and completion methods," *Information and Control* (1985) (to appear).
37. D. A. Plaisted and S. Greenbaum. "Problem representations for backchaining and equality in resolution theorem proving," *Proceedings First Annual AI Applications Conference*, Denver, CO (December 1984).
38. U. Reddy. "Narrowing as the operational semantics of functional languages", Unpublished report, Department of Computer Science, University of Utah, Salt Lake City, UT, 1985 (submitted).
39. J. L. Rémy and H. Zhang. "Reveur4: a system for validating conditional algebraic specifications of abstract data types", Rapport 54506, Centre de Recherche en Informatique de Nancy, France, 1984.
40. P. Réty, C. Kirchner, H. Kirchner and P. Lescanne. "NARROWER: A new algorithm for unification and its application to logic programming," *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France (May 1985) (to appear).
41. J. Robinson. "Theorem proving on the computer," *J. of the Association for Computing Machinery* (1963), pp. 163-174.
42. J. A. Robinson and E. E. Sibert. "LOGLISP: An alternative to PROLOG," In: *Machine Intelligence 10*, J. E. Hayes, D. Michie and Y-H Pao, eds. Ellis Horwood, 1982, pp. 399-419.
43. E. Y. Shapiro. "A subset of Concurrent Prolog and its interpreter", Institute for New Generation Computer Technology, Tokyo, February 1983.
44. J. R. Slagle. "Automated theorem-proving for theories with simplifiers, commutativity, and associativity," *J. of the Association for Computing Machinery*, Vol. 21, No. 4 (1974), pp. 622-642.
45. M. E. Stickel. "A unification algorithm for associative-commutative functions," *J. of the Association for Computing Machinery*, Vol. 28, No. 3 (1981), pp. 423-434.
46. P. A. Subrahmanyam and J. H. You. "FUNLOG = functions + logic: A computational model integrating functional and logic programming," *Proceedings of the IEEE International Symposium on Logic Programming*, Atlantic City, NJ (February 1984), pp. 144-153.
47. G. J. Sussman, *et al.* "Micro-planner reference manual", Report 203, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1970.
48. H. Tamaki. "Semantics of a logic programming language with a reducibility predicate," *Proceedings of the IEEE International Symposium on Logic Programming*, Atlantic City, NJ (February 1984), pp. 259-264.
49. H. Tamaki and T. Sato. "Program transformation through meta-shifting," *New Generation Computing*, Vol. 1 (1983), pp. 93-98.
50. D. A. Turner. "SASL language manual", Department of Computational Science, University of St. Andrews, 1979.