SRI International

TABLOG: THE DEDUCTIVE-TABLEAU PROGRAMMING LANGUAGE

September 1984

Technical Note 328

By: Yonathan Malachi
Zohar Manna
Computer Science Department
Stanford University

Richard Waldinger
Artificial Intelligence Center
Computer Science and Technology Division

This research was supported in part by the National Science Foundation under Grants MCS-82-14523, MCS-81-11586, and MCS-81-05565, by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014, by DARPA under Contract N0039-82-C-0250, and by a grant from IBM Research, San Jose, California.

Presented at the ACM Symposium on LISP and Functional Programming, University of Texas at Austin, August 5-8, 1984.

SRI International®
333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Abstract

TABLOG (Tableau Logic Programming Language) is a language based on first-order predicate logic with equality that combines functional and logic programming. TABLOG incorporates advantages of LISP and PROLOG.

A program in TABLOG is a list of formulas in a first-order logic (including equality, negation, and equivalence) that is more general and more expressive than PROLOG's Horn clauses. Whereas PROLOG programs must be relational, TABLOG programs may define either relations or functions. While LISP programs yield results of a computation by returning a single output value, TABLOG programs can be relations and can produce several results simultaneously through their arguments.

TABLOG employs the Manna-Waldinger *deductive-tableau* proof system as an interpreter in the same way that PROLOG uses a resolution-based proof system. Unification is used by TABLOG to match a call with a line in the program and to bind arguments. The basic rules of deduction used for computing are nonclausal resolution and rewriting by means of equality and equivalence.

A pilot interpreter for the language has been implemented.

1. Introduction

Logic programming [Kowalski 79] attempts to improve programmer productivity by proposing logic, a human-oriented language, as a programming language. PROLOG, the flagship of logic-programming languages, based on a resolution proof system, has a restricted syntax. TABLOG is based on a more flexible theorem prover, the *deductive-tableau* proof system [Manna and Waldinger 80], which allows a more intuitive and a richer syntax. A TABLOG program is a list of *assertions* in [quantifier-free] first-order logic with equality. The execution of a program corresponds to the proof of a *goal*, which produces the desired output(s) as a side effect.

Since a particular procedure is specified by the programmer, and since the proof taking place is always a proof of a special case of a theorem—namely, the case for the given input—the program interpreter does not need all the deduction rules available in the original deductive-tableau proof system. The theorem prover can be more directed, efficient, and predictable than a theorem prover used for program synthesis or for any other general-purpose deduction.

2. TABLOG Syntax

Syntactic Objects

The language is that of the quantifier-free first-order predicate logic with equality, consisting of the following:

- truth values: *true, false.*
- connectives: $\wedge$, $\vee$, $\neg$, $\equiv$, $\rightarrow$ (*implies*), $\leftarrow$ (*if*), *if-then-else.*
- variables such as u, v, x_1, y_{25}.
- constants such as a, b, [], 5.
- predicates such as $=$, **prime**, $\in$, $\geq$.
- functions such as **gcd**, **append**, $+$.

The user must declare the variables, constants, functions, and predicates used in the program; some primitive constants, functions, and predicates (such as 0, [], $+$, $-$, $\geq$, **odd**) are predefined.

Note that we use the *if-then-else* construct, both as a connective for formulas

$$\text{if } u = [] \text{ then } \mathbf{empty}(u) \text{ else } \mathbf{sorted}(u)$$

and as an operator generating terms

$$\mathbf{gcd}(x, y) = \text{if } x \geq y \text{ then } \mathbf{gcd}(x-y, y) \text{ else } \mathbf{gcd}(x, y-x).$$

This, together with $\leftarrow$ (reverse implication), enables the programmer to write LISP-style as well as PROLOG-style programs.

Programs

A program is a list of *assertions* (formulas in [quantifier-free] first-order logic with equality), specifying the algorithm. Variables are implicitly universally quantified.

Here is a very simple program for appending two lists:

$$\begin{aligned}&\textbf{append}([\,],v) \;=\; v\\ &\textbf{append}(x\circ u,v) \;=\; x\circ\textbf{append}(u,v).\end{aligned}$$

The $\circ$ symbol denotes the list insertion (**cons** in LISP) operator, and $[\,]$ denotes the empty list (**nil** in LISP).

A call to a program is a *goal* to be proved. Like the assertions, goals are formulas in logic, but variables are implicitly existentially quantified. The bindings of these variables are recorded throughout the proof and become the outputs of the program upon termination.

For example, a call to the **append** program above might be

$$z \;=\; \textbf{append}([1,2,3],\ [a,b]).$$

The output of the execution of this program call will be

$$[1,2,3,a,b],$$

as expected.

The list construct (e.g. $[1,2,3]$) is for convenience in expressing input and output, and denotes the term $1\circ(2\circ(3\circ[\,]))$.

3. Examples

The following examples demonstrate the basic features of TABLOG. The correctness of these programs does not depend on the order of assertions in the program. It is possible, however, to write programs that do take advantage of the known order of the interpreter's goal evaluation, as will be explained later.

In the examples, we use x and y (possibly with subscripts) for variables intended to be assigned atoms (integers in most of the examples); u and v (possibly with subscripts) are variables used for lists.

Deleting a List Element

The following program deletes all [top-level] occurrences of an element x from a list:

$$\begin{aligned}&\textbf{delete}(x,[\,]) \;=\; [\,]\\ &\textbf{delete}(x,\,y\circ u) \;=\; \big(\text{if } x=y \text{ then } \textbf{delete}(x,u)\\ &\qquad\qquad\qquad\qquad \text{else } y\circ\textbf{delete}(x,u)\big).\end{aligned}$$

This program demonstrates the use of equality, *if-then-else*, and recursive calls. For those who prefer the PROLOG style of programming, the last line could be replaced by assertions:

$\mathbf{delete}(x, x \circ u) = \mathbf{delete}(x, u)$
$x \neq y \ \rightarrow \ \mathbf{delete}(x, y \circ u) = y \circ \mathbf{delete}(x, u)$

To remove all occurrences of a from the list $[a, b, a, c]$ the goal

$$z = \mathbf{delete}(a, [a, b, a, c])$$

is given to the interpreter.

Set Union

The following example, a program to find the union of two sets represented by lists, demonstrates the use of negation, equivalence and *if-then-else*:

1. $\mathbf{union}([\,], v) = v$
2. $\mathbf{union}(x \circ u, v) \ =$ if $\mathbf{member}(x, v)$
   then $\mathbf{union}(u, v)$
   else $(x \circ \mathbf{union}(u, v))$
3. $\neg\mathbf{member}(x, [\,])$
4. $\mathbf{member}(x, y \circ u) \ \equiv \ ((x = y) \vee \mathbf{member}(x, u))$

Lines 1 and 2 define the **union** function. Line 1 defines the union of the empty set with another set, and line 2 asserts that the head x of the first set $x \circ u$ should be inserted into the union if it is not already in the second set v.

Lines 3 and 4 define the **member** relation. Line 3 specifies that no element is a member of the empty set, and line 4 defines how to test recursively membership in a nonempty set.

Factorial

The following program will compute the factorial of a nonnegative integer x:

$\mathbf{fact}(0) = 1$
$\mathbf{fact}(x) = x * \mathbf{fact}(x - 1) \ \leftarrow \ x \geq 1$

The corresponding PROLOG program will be

$\mathbf{factp}(0, 1)$
$\mathbf{factp}(x, z) \ \leftarrow \ x_1 \ \mathbf{is} \ x-1 \ \wedge \ \mathbf{factp}(x_1, y) \ \wedge \ z \ \mathbf{is} \ x * y.$

The **is** construct is used in PROLOG to force the evaluation of an arithmetic expression.

Quicksort

Here is a TABLOG program that uses quicksort to sort a list of numbers. It combines a PROLOG-style relational subprogram for partitioning with a LISP-style functional subprogram for sorting.

1. $\mathbf{qsort}([\,]) = [\,]$
2. $\mathbf{qsort}(x \circ u) = \mathbf{append}(\mathbf{qsort}(u_1),\ x \circ \mathbf{qsort}(u_2))$
   $\leftarrow\ \mathbf{partition}(x, u, u_1, u_2)$
3. $\mathbf{partition}(x, [\,], [\,], [\,])$
4. $\mathbf{partition}(x, y \circ u, y \circ u_1, u_2)$
   $\leftarrow\ y \leq x\ \wedge\ \mathbf{partition}(x, u, u_1, u_2)$
5. $\mathbf{partition}(x, y \circ u, u_1, y \circ u_2)$
   $\leftarrow\ y > x\ \wedge\ \mathbf{partition}(x, u, u_1, u_2)$

The assertions in lines 1 and 2 form the sorting subprogram. Line 1 asserts that the empty list is already sorted. Line 2 specifies that, to sort a list $x \circ u$, with head x and tail u, one should append the sorted version of two sublists of u, u_1 and u_2, and insert the element x between them; the two sublists u_1 and u_2 are determined by the subprogram **partition** to be the elements of u less than or equal to x and greater than x, respectively.

The assertions in lines 3 to 5 specify how to partition a list according to a partition element x. Line 3 discusses the partitioning of the empty list, while lines 4 and 5 treat the case in which the list is of the form $y \circ u$. Line 4 is for the case in which y, the head of the list, is less than or equal to x; therefore, y should be inserted into the list u_1 of elements not greater than x. Line 5 is for the alternative case.

The **append** function for appending two lists was defined earlier.

4. Comparison with PROLOG

Functions and Equality

While PROLOG programs must be relations, TABLOG programs can be either relations or functions. The availability of functions and equality makes it possible to write programs more naturally. The functional style of programs frees the programmer from the need to introduce many auxiliary variables.

We can compare the PROLOG and TABLOG programs for quicksort. In TABLOG, the program uses the unary function **qsort** to produce a value, whereas a PROLOG program is a binary relation **qsortp**; the second argument is needed to hold the output.

The second assertion in the TABLOG program is

$\mathbf{qsort}(x \circ u) = \mathbf{append}(\mathbf{qsort}(u_1), x \circ \mathbf{qsort}(u_2))$
$\leftarrow\ \mathbf{partition}(x, u, u_1, u_2)$

The corresponding clause in the PROLOG program will be something like

$$\begin{aligned}\mathbf{qsortp}(x \circ u, z) \leftarrow\ & \mathbf{partition}(x, u, u_1, u_2) \wedge \\ & \mathbf{qsortp}(u_1, z_1) \wedge \\ & \mathbf{qsortp}(u_2, z_2) \wedge \\ & \mathbf{appendp}(z_1, x \circ z_2, z).\end{aligned}$$

The additional variables z_1 and z_2 are required to store the results of sorting u_1 and u_2. This demonstrates the advantage of having functions and equality in the language. Note that, although function symbols exist in PROLOG, they are used only for constructing data structures (like TABLOG's primitive functions) and are not reduced.

Negation and Equivalence

In PROLOG, negation is not available directly; it is simulated by finite failure. To prove $not(P)$, PROLOG attempts to prove P; $not(P)$ succeeds if and only if the proof of P fails. In TABLOG, negation is treated like any other connective of logic. Therefore, we can prove formulas such as $\neg\mathbf{member}(1, [2, 3])$.

The TABLOG **union** program, described earlier, uses both equivalence and negation:

$$\mathbf{union}([\,], v) = v$$

$$\begin{aligned}\mathbf{union}(x \circ u, v) = \text{if } & \mathbf{member}(x, v) \\ & \text{then } \mathbf{union}(u, v) \\ & \text{else } (x \circ \mathbf{union}(u, v))\end{aligned}$$

$$\neg\mathbf{member}(x, [\,])$$

$$\mathbf{member}(x, y \circ u) \equiv (x = y) \vee \mathbf{member}(x, u).$$

Here is a possible PROLOG implementation of the same algorithm:

$$\mathbf{unionp}(x \circ u, v, z) \leftarrow \mathbf{memberp}(x, v) \wedge \mathbf{unionp}(u, v, z)$$

$$\mathbf{unionp}(x \circ u, v, x \circ z) \leftarrow \mathbf{unionp}(u, v, z)$$

$$\mathbf{unionp}([\,], v, v)$$

$$\mathbf{memberp}(x, x \circ u)$$

$$\mathbf{memberp}(x, y \circ u) \leftarrow \mathbf{memberp}(x, u).$$

Changing the order of the first two clauses in the PROLOG program will result in an incorrect output; the second clause is correct only for the case in which x is not a member of v. The TABLOG assertions can be freely rearranged; this suggests that all of them can be matched against the current goal in parallel, if desired.

Unification

The unification procedure built into PROLOG is not really unification (e.g., as defined in [Robinson 65]); it does not fail in matching an expression against one of its proper

subexpressions since it lacks an *occur-check*. When a theorem prover is used as a program interpreter, the omission of the occur-check makes it possible to generate cyclic expressions that may not correspond to any concrete objects.

The unification used by the TABLOG interpreter does include an occur-check, so that only theorems can indeed be proved.

5. Comparison with LISP

LISP programs are functions, each returning one value; the arguments of a function must be bound before the function is called. In TABLOG, on the other hand, programs can be either relations or functions, and the arguments need not be bound; these arguments will later be bound by unification.

We can illustrate this with the quicksort program again, concentrating on the partition subprogram. In TABLOG, we have seen how to achieve the partition by a predicate with four arguments, two for input and two for output:

1. $\mathbf{partition}(x, [\,], [\,], [\,])$
2. $\mathbf{partition}(x, y \circ u, y \circ u_1, u_2)$
   $\leftarrow\ y \leq x\ \wedge\ \mathbf{partition}(x, u, u_1, u_2)$
3. $\mathbf{partition}(x, y \circ u, u_1, y \circ u_2)$
   $\leftarrow\ y > x\ \wedge\ \mathbf{partition}(x, u, u_1, u_2)$

The definition of the program **partition** is much shorter and cleaner than the corresponding LISP program:

$\mathbf{highpart}(x, u)\ \Leftarrow$
    if $\mathbf{null}(u)$ then **nil**
    else-if $x \geq \mathbf{car}(u)$ then $\mathbf{highpart}(x, \mathbf{cdr}(u))$
    else $\mathbf{cons}(\mathbf{car}(u), \mathbf{highpart}(x, \mathbf{cdr}(u)))$

$\mathbf{lowpart}(x, u)\ \Leftarrow$
    if $\mathbf{null}(u)$ then **nil**
    else-if $x \geq \mathbf{car}(u)$
        then $\mathbf{cons}(\mathbf{car}(\mathrm{u}), \mathbf{lowpart}(\mathrm{x}, \mathbf{cdr}(\mathrm{u})))$
    else $\mathbf{lowpart}(x, \mathbf{cdr}(u))$.

We can generate the two sublists in LISP simultaneously, but this will require even more pairing and decomposition.

Note that unification also gives us "free" decomposition of the list argument into its head and tail; in the LISP program, this decomposition requires explicit calls to the functions **car** and **cdr**.

6. The Deductive-Tableau Proof System

In this section, we give a brief summary of the Manna-Waldinger deductive-tableau proof system [Manna and Waldinger 80 and 82]. This proof system is used as the TABLOG interpreter. We describe only the deduction rules actually employed in it.

A *deductive tableau* consists of rows, each containing either an *assertion* or a *goal*. The assertions and goals (both of which we refer to by the generic name *entries*) are first-order logic formulas; the theorem is proved by manipulating them. The declarative or logical meaning of a tableau is that, if every instance of all the assertions is true, then some instance of at least one of the goals is true. The assertions in the tableau are like clauses in a standard resolution theorem prover—but they can be arbitrary first-order formulas, not just disjunctions of literals.

The theorem to be proved is entered as the initial goal. A proof is constructed by adding new goals to the tableau, using deduction rules, in such a way that the final tableau is semantically equivalent to the original one. The proof is complete when we have generated the goal *true*.

Deduction Rules

The basic rules used for the program execution task are the following:

- *Nonclausal Resolution*: This generalized resolution rule allows removal of a subformula P from a goal $\mathcal{G}[P]$ by means of an appropriate assertion $\mathcal{A}[\hat{P}]$. Resolving the goal

  $$\mathcal{G}[P]$$

  with the assertion

  $$\mathcal{A}[\hat{P}],$$

  provided that P and $\hat{P}$ are unifiable, i.e., $P\theta = \hat{P}\theta$ for some (most-general) unifier θ, we get the new goal

  $$not(\mathcal{A}'[false]) \ \wedge \ \mathcal{G}'[true],$$

  where $\mathcal{A}'[false]$ is $\mathcal{A}\theta$ after all occurrences of $P\theta$ have been replaced by *false*, and similarly for $\mathcal{G}'[true]$. This deduction rule can be justified by case analysis.

  The choice of the unified subformulas is governed by the *polarity strategy* [Murray 82]. A subformula has *positive* polarity if it occurs within an even number of (explicit or implicit) negations, and has *negative* polarity if it occurs within an odd number of negations. (An assertion has an implicit negation applied to it.) A subformula can occur both positively and negatively in a formula. According to the polarity strategy, the subformula P will be replaced by *false* only if it occurs with negative polarity and the subformula Q will be replaced by *true* only if it occurs with positive polarity.

- *Equality Rule*: An asserted [possibly conditional] equality of two terms can be used to replace one of the terms with the other in a goal. If the asserted equality is conditional, the conditions are added to the resulting goal as conjuncts.

Thus, suppose the assertion is of the form

$$\mathcal{A}[s = t],$$

and the goal is

$$\mathcal{G}[\hat{s}],$$

where s and $\hat{s}$ are unifiable, i.e., $s\theta = \hat{s}\theta$ for some unifier θ. Then we get the new goal

$$not(\mathcal{A}'[false]) \ \wedge \ \mathcal{G}'[t'],$$

where $\mathcal{A}'[false]$ is $\mathcal{A}\theta$ after all occurrences of the equality $s\theta = t\theta$ (which should occur with negative polarity) have been replaced by *false*, and where $\mathcal{G}'[t']$ is $\mathcal{G}\theta$ after the replacement of all occurrences of the term $s\theta$ by $t\theta$.

The reflexivity axiom for equality $x = x$ is implicitly included among the assertions of every tableau.

- *Equivalence Rule*: The replacement of one subformula by another asserted to be equivalent to it. This is completely analogous to the equality rule except that we replace atomic formulas rather than terms, using equivalence rather then equality.
- *Simplification*: The replacement of a formula by an equivalent but simpler formula. Both propositional and arithmetic simplification are performed automatically by the TABLOG interpreter.

While nonclausal resolution and the equivalence rule can be performed unifying arbitrary subformulas, the TABLOG interpreter applies these deduction rules unifying atomic subformulas only.

7. Program Semantics

The logical interpretation of a tableau containing a TABLOG program and a call to it is the logical sentence associated with the tableau: the conjunction of the universal closures of the assertions implies the existential closure of the goal.

The desired goal is reduced to *true* by means of the assertions and the deduction rules. The variables are bound when subexpressions of the goal (or derived subgoals) are unified with subexpressions of the assertions. The order of the reduction is explained in the next section. The output of the program is the final binding of the variables of the original goal.

We distinguish between *defined* functions, whose semantics is defined by the user program, and *primitive* functions, which are either data constructors (e.g., $\circ$), or are built-in and have their semantics defined by attached procedures in the simplifier; for example, an expression like $(2 + x + 5) \circ [\,]$ is considered primitive and will be automatically simplified to $(x + 7) \circ [\,]$.

As in PROLOG, variables are local to the assertion or goal in which they appear. Renaming of variables is done automatically by the interpreter when there is a collision of names between the goal and assertion involved in a derivation step.

The variables of the original goal are the output variables. The interpreter keeps their binding throughout the derivation; the same variable name can be used for a different purpose in other assertions or goals.

8. Program Execution

Every line in a program is an assertion in the tableau; a call to the program is a goal in the same tableau.

The tableau system provides us with deduction rules but with no specific order in which to apply them. To use it as a programing language, we have to specify the order of application both for predictability and for efficiency.

The proof system is used to execute programs in a way analogous to the inversion of a matrix by linear operations on its rows, where we simultaneously apply the same transformations to the matrix to be inverted and to the identity matrix. In the program execution process, we start with a tableau containing the assertions of the program and a goal calling this program; we apply the same substitutions (obtained by unification) to the current subgoal and to the binding of the output variables. A matrix inversion is complete when we reduce the original matrix to the identity matrix; in TABLOG we are done when we have reduced the original goal to *true*. At this point, the result of the computation is the final binding of the output variables.

Although in the declarative (logical) semantics of the tableau the order of entries is immaterial, the procedural interpretation of the tableau as a program takes this order into account; changing the order of two assertions or changing the order of the conjuncts or disjuncts in an assertion or a goal may produce different computations.

The user for his part, has to specify an algorithm by employing the predefined order of evaluation of the tableau. At each step of the execution, one *basic expression* (a nonvariable term or an atomic formula) of the current goal is reduced. The expression to be reduced is selected by scanning the goal from left to right. The first (leftmost) basic expression that has only primitive arguments (i.e., that contain only variables, constants, and primitive functions) is chosen and reduced, if possible. Matching the selected expression against assertions is done in order of appearance.

This is best explained with an example:

To sort the list $[2,1,4,3]$ using quicksort, we write the goal

$$z = \mathbf{qsort}([2,1,4,3]).$$

To execute this goal, the expression chosen for reduction will be the term $\mathbf{qsort}([2,1,4,3])$, i.e., $\mathbf{qsort}(2 \circ [1,4,3])$. This term unifies with the leftmost term $\mathbf{qsort}(x \circ u)$ in the second assertion of the quicksort program,

$$\begin{aligned} \mathbf{qsort}(x \circ u) &= \mathbf{append}(\mathbf{qsort}(u_1), x \circ \mathbf{qsort}(u_2)) \\ &\leftarrow \mathbf{partition}(x, u, u_1, u_2). \end{aligned}$$

According to the equality rule, it will be replaced by the corresponding instance of the right-hand side of the equality; this is done only after the unifier

$$\{x \leftarrow 2,\ u \leftarrow [1,4,3]\}$$

is applied to both the goal and the assertion. The occurrence of the equality

$$\mathbf{qsort}(2 \circ [1,4,3]) = \mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2))$$

is replaced by *false* in the [modified] assertion, the occurrence of the term

$$\mathbf{qsort}(2 \circ [1,4,3])$$

is replaced by the term

$$\mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2))$$

in the (modified) goal, and a conjunction is formed, obtaining

$$\begin{array}{l} not(false \leftarrow \mathbf{partition}(2, [1,4,3], u_1, u_2) \wedge \\ z = \mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2)). \end{array}$$

This formula can be reduced by the simplifications

$$(false \leftarrow P) \Rightarrow not\ P$$

and

$$not(not\ P) \Rightarrow P$$

to obtain the new goal

$$\begin{array}{l} \mathbf{partition}(2, [1,4,3], u_1, u_2) \wedge \\ z = \mathbf{append}(\mathbf{qsort}(u_1), 2 \circ \mathbf{qsort}(u_2)). \end{array}$$

Continuing with this example, we now have a case in which the expression to be reduced is an atomic formula, namely,

$$\mathbf{partition}(2, [1,4,3], u_1, u_2).$$

This atomic formula is unifiable with a subformula in the second assertion of the **partition** subprogram (with variables renamed to resolve collisions)

$$\begin{array}{l} \mathbf{partition}(x, y \circ u, y \circ u_3, u_4) \\ \quad \leftarrow\ y \leq x\ \wedge\ \mathbf{partition}(x, u, u_3, u_4). \end{array}$$

Nonclausal resolution is now performed to further reduce the current goal. The unifier

$$\{x \leftarrow 2,\ y \leftarrow 1,\ u \leftarrow [4,3],\ u_1 \leftarrow 1 \circ u_3,\ u_2 \leftarrow u_4\}$$

is applied to both the assertion and the goal; the formula

$$\mathbf{partition}(2, [1,4,3], 1 \circ u_3, u_4)$$

is replaced by *false* in the [modified] assertion and by *true* in the goal. Once again a conjunction is formed and the new goal generated (after simplification) is

$$\mathbf{partition}(2, [4,3], u_3, u_4) \wedge$$
$$z = \mathbf{append}(\mathbf{qsort}(1 \circ u_3),\ 2 \circ \mathbf{qsort}(u_4)).$$

Eventually we reach the subgoal

$$z = [1,2,3,4],$$

where the right-hand side of the equality contains only primitive functions and constants. The execution then terminates and the desired output is

$$[1,2,3,4].$$

Note that some functions and predicates (e.g., $\circ$ in this example) are predefined to be primitive; an expression in which such a symbol is the main operator is never selected to be reduced, although its subexpressions may be reduced.

Backtracking

If the selected expression cannot be reduced, the search for other possible reductions is done by backtracking.

In PROLOG each goal is a conjunction, so all the conjuncts must be proved; this means that, when facing a dead end, we have to undo the most recent binding and try other assertions.

In TABLOG the situation is more complex: each goal (and each assertion) is an arbitrary formula, so it is possible to satisfy it without satisfying all its atomic subformulas. Therefore, when the TABLOG interpreter fails to find an assertion that reduces some basic expression, it tries to reduce the next expression that can allow the proof to proceed. In the case in which the expression that cannot be reduced is "essential" (for example, a conjunct in a conjunctive goal), no other subexpression will be attempted and backtracking will occur.

During backtracking, the goal from which the current goal was derived becomes the new current goal, but the next plausible assertion is used. This is similar to the backtracking used in PROLOG.

The Implementation

A prototype interpreter for TABLOG is implemented in MACLISP. The implemented system serves as a program editor, debugger, and interpreter. All the examples mentioned in this paper have been executed on this interpreter.

The backtracking mechanism provides a simple way of changing the interpreter so that lazy evaluation can be employed—i.e., so that attempts can be made to evaluate expressions even if they have nonprimitive arguments.

Because the interpreter is built on top of a versatile theorem-proving system, the execution of programs is relatively slow. The interpreter now handles complicated cases that might arise in a more general theorem-proving task, but will never occur in TABLOG. We hope that performance will be improved considerably by tuning the simplifier and utilizing tricks from PROLOG implementations to make the binding of variables faster.

9. Related Research

Logic programming has become a fashionable research topic in recent years. Most of the research relates to PROLOG and its extensions. We mention here some of the work that has been done independently of TABLOG to generate languages similar to TABLOG in their intention and capabilities.

While the deductive-tableau theorem prover used for TABLOG execution is based on a generalized resolution inference rule, [Haridi 81], [Haridi and Sahlin 83], and [Hansson, Haridi, and Tärnlund 82] describe a programming language based on a natural-deduction proof system. They do allow quantifiers and other connectives in the language but the syntax of their assertions is somewhat restricted.

[Kornfeld 83] extends PROLOG to include equality; asserting equality between two objects in his language causes the system to unify these objects when regular unification fails. This makes it possible to unify objects that differ syntactically. Kornfeld treats only Horn clauses and does not introduce any substitution rule either for equality or for equivalence.

[Tamaki 84] extends PROLOG by introducing a reducibility predicate, denoted by $\triangleright$. This predicate has semantics similar to the way TABLOG uses equality for rewriting terms. This work also includes *f-symbols* and *d-symbols* that are analogous to TABLOG's distinction between defined and primitive functions. The possible nesting of terms is restricted and programs must be in Horn clause form.

OBJ [Goguen, Meseguer, and Plaisted 82] is also related to logic programming. It is based, however, on the algebraic semantics of abstract data types and equational theory rather than on [resolution-based] theorem proving in first-order logic. OBJ1 is an advanced implementation of the language that allows parameterized and hierarchical programming. OBJ1 includes system features for convenience and efficiency; it uses one-way pattern matching to apply rewrite rules rather than two-way unification. [Goguen and Meseguer 84] describes EQLOG, the extension of OBJ to include unification and Horn clauses.

There are PROLOG systems, such as LOGLISP [Robinson and Sibert 82] and QLOG [Komorowski 79 and 82] that are implemented within LISP systems. These systems allow the user to invoke the PROLOG interpreter from within a LISP program and vice versa. In TABLOG, however, LISP-like features and PROLOG-like features coexist peacefully in the same framework and are processed by the same deductive engine.

10. Conclusions and Discussion

The TABLOG language is a new approach to logic programming: instead of patching up PROLOG with new constructs to eliminate its shortcomings, we suggest a more powerful deductive engine.

The combination in TABLOG of unification as a binding mechanism, equality for specifying functions, and first-order logic for specifying predicates creates a rich language that is clean from a logical point of view. As a consequence, programs correspond to our intuition and are easier to write, read, and modify. We can mix LISP-style and PROLOG-style programming and use whichever is more convenient for the problem or subproblem.

By restricting the general-purpose deductive-tableau theorem prover and forcing it to follow a specific search order, we have made it suitable to serve as a program interpreter; the specific search order makes it both more predictable and more efficient than attempting to apply the deduction rules arbitrarily.

While the theorem prover supports reasoning with quantified formulas [Manna and Waldinger 82; Bronstein 83], the ramifications of including quantifiers in the language are still under investigation. Quantifiers would certainly enhance the expressive power of TABLOG, but we believe that they are more suited to a specification language than a programming language.

It seems very natural to extend TABLOG to parallel computation. The inclusion of real negation makes it possible to write programs that do not depend on the order of assertions.

The extension of TABLOG to support concurrent programs is being pursued. If the conditions of the assertions are disjoint, several assertions can be matched against the current subgoal in parallel. In addition, disjunctive goals can be split between processes. If there are no common variables, conjuncts can be solved in parallel; otherwise some form of communication is required.

The *or-parallelism* and *and-parallelism* suggested for PROLOG are applicable for TABLOG as well. The or-parallelism of PROLOG relates to matching against many assertions; in TABLOG or-parallelism is possible within every goal, since, for example, goals can be disjunctive. In TABLOG can other forms of parallelism can be applied to nested function calls.

Acknowledgments

Thanks are due to Martin Abadi, Yoram Moses, Oren Patashnik, Jon Traugott, and Joe Weening for comments on various versions of this paper. We are especially indebted to Bengt Jonsson and Frank Yellin for reading many versions of the manuscript and providing insightful comments and suggestions.

References

[Bronstein 83] .
A. Bronstein, "Full quantification and special relations in a first-order logic theorem prover," programming project, Computer Science Department, Stanford University, 1983.

[Clark and Tärnlund 82]
K. L. Clark and S.-Å. Tärnlund (editors), *Logic Programming*, Academic Press (1982). A.P.I.C. Studies in Data Processing No. 16.

[Goguen and Meseguer 84]
J. Goguen and J. Meseguer, "Equality, types, modules and generics for logic programming," in *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 2-6, 1984.

[Goguen, Meseguer, and Plaisted 82]
J. Goguen, J. Meseguer, and D. Plaisted, "Programming with parameterized abstract objects in OBJ," in *Theory and Practice of Software Technology*, edited by D. Ferrari, M. Bolognani, and J. Goguen, North-Holland, 1982.

[Hansson, Haridi, and Tärnlund 82]
Å. Hansson, S. Haridi, and S.-Å. Tärnlund, "Properties of a Logic Programming Language," in [Clark and Tärnlund 82].

[Haridi 81]
S. Haridi, "Logic programming based on a natural deduction system," Ph.D. Thesis, Department of Telecommunication Systems and Computer Science, The Royal Institute of Technology, Stockholm, Sweden, 1981.

[Haridi and Sahlin 83]
S. Haridi and D. Sahlin, "Evaluation of logic programs based on natural deduction," Technical report RITA-CS-8305 B, Department of Telecommunication Systems and Computer Science, The Royal Institute of Technology, Stockholm, Sweden, 1983.

[Komorowski 79]
H. J. Komorowski, "The QLOG Interactive Environment," Technical Report LITH-MAR-R-79-19, Informatics Lab, Linkopping University, Sweden, August 1979.

[Komorowski 82]
H. J. Komorowski, "QLOG The Programming Environment for Prolog in LISP," in [Clark and Tärnlund 82]

[Kornfeld 83]
W. Kornfeld, "Equality for Prolog," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, August 1983.

[Kowalski 79]
R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.

[Manna and Waldinger 80]
Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, pp. 92–121, January 1980.

[Manna and Waldinger 82]
Z. Manna and R. Waldinger, "Special relations in program-synthetic deduction," Department of Computer Science, Technical Report No. STAN-CS-82-902, Stanford University. To appear in *Journal of the ACM.*

[Murray 82]
N. V. Murray, "Completely nonclausal theorem proving," *Artificial Intelligence*, Vol. 18, No. 1, pp. 67–85.

[Robinson 65]
J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, Vol. 12, No. 1, Jan 1965, pp. 23–41.

[Robinson and Sibert 82]
J. A. Robinson and E. E. Sibert, "LOGLISP: and alternative to PROLOG," in *Machine Intelligence 10*, J. E. Hayes, D. Michie, and Y-H Pao editors, Ellis Horwood Ltd., Chichester, 1982.

[Tamaki 84]
H. Tamaki, "Semantics of a logic programming language with a reducibility predicate," *Proceedings of the IEEE Logic Programming Conference*, Atlantic City, February 1984.