Nonclausal Logic Programming

by

Yonathan Malachi

Department of Computer Science

Stanford University
Stanford, CA 94305

NONCLAUSAL LOGIC PROGRAMMING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

by

Yonathan Malachi

March 1986

ABSTRACT

This dissertation proposes a new logic programming language, TABLOG, and studies its properties. A program in TABLOG is a list of formulas in (quantifier-free) first-order logic with equality, which usually amounts to a very natural description of a problem and a procedure for solving it.

The inclusion of equivalence, negation, conditionals, functions, and equality in TABLOG enables the programmer to combine functional and relational programming in a single framework. Unification is used as the binding mechanism and makes it convenient to pass unbound variables to a program and to manipulate partially computed objects.

The Manna-Waldinger *deductive-tableau* proof system is employed as an interpreter for the language in the same way that a resolution proof system serves as an interpreter for PROLOG. The basic rules of inference used in the system are non-clausal resolution, equational rewriting, and replacement of formulas by equivalent ones.

This dissertation describes the syntax and semantics of TABLOG and compares it with LISP and PROLOG. The examples and discussions scattered throughout the text demonstrate various programming styles in TABLOG and also point out its weak and strong points.

Some theoretical properties of TABLOG and the proof procedure used as its interpreter are studied and compared with the corresponding properties of PROLOG and LISP.

This research was supported in part by DARPA under contracts N00039-82-C-0250 and N00038-84-C-0211, by the National Science Foundation under grants MCS79-09495 and DCR82-14523, by the United States Air Force Office of Scientific Research under Contract AFOSR-81-0014. Graduate studies and work that preceded and served as the basis for this research were supported by a Rothschild

Possible ways to extend the procedural interpretation for parallel execution are proposed and related to models for parallel execution of PROLOG.

All the examples described throughout the discussion were executed on an implemented system; a user's guide for this system appears in an appendix.

Acknowledgments

I am deeply indebted to my advisor Zohar Manna for his personal, moral, and professional support and guidance throughout my years at Stanford. Zohar's encouragement and feedback were critical and essential in enabling me to get to the completion of this dissertation. I am also very grateful to Richard Waldinger whose help was beyond and over what is expected from a member of the reading committee. Richard was always there when I needed encouragement, advice or feedback. He also helped improving the presentation of the material in this dissertation by his careful reading and enlightening comments. Zohar and Richard also collaborated with me on [Malachi, Manna, and Waldinger 84] and [Malachi, Manna, and Waldinger 85] in which some of the material of chapters 2, 3, 4, 5, 6, and 7 appeared.

John McCarthy's comments and questions contributed significantly to the scope of this dissertation and in particular to Chapter 9 and the inclusion of Appendix A.

I also want to thank all my friends at Stanford that have made my stay here very enjoyable (and thus contributed to prolonging it); I know that if I try to list them all I will forget someone so if you think you belong in this list, you surely do.

Special thanks are due to those who helped with comments related to various pieces and versions of the research described here. These include Martin Abadi, Alex Bronstein, Bengt Jonsson, Yoram Moses, Eric Muller, Oren Patashnik, Chuck Restivo, Jon Traugott, Joe Weening, and Frank Yellin. If I forgot someone here, please accept my apologies and thanks.

The staff and faculty of the Computer Science Department were very helpful in creating a great environment for study and research.

Mark Stickel was very helpful with comments and prompt answers to all my questions and comments.

Directly related to this research, I found the PROLOG mailing list very stimulating. I want to thank Chuck Restivo for moderating the discussions and organiz-

I want to thank Prof. Bernard Widrow for his help during my first year at Stanford.

Many institutions and agencies supported me during my years at Stanford while and before working on the research described here. In particular, I want to thank Yad Avi Hayishuv, The American Society for The Technion, and IBM Corporation for their generous scholarships or fellowships.

Last, but certainly not least, I want to thank Sara, Guy, Ainat, and Saggi and my parents and parents-in-law for all their love and support.

To Sara

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation

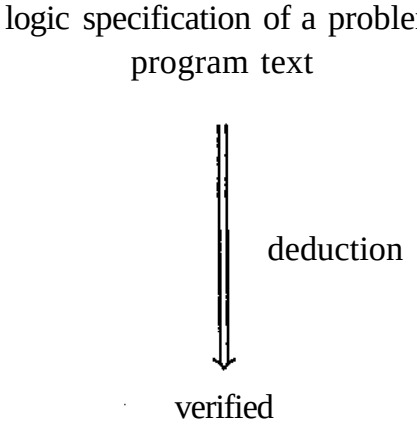
In the last two decades we have encountered the so-called "software crisis." The cost of writing and maintaining software grows as the cost of hardware drops. Software project deadlines have to be pushed and software packages are, in general, not free of errors. The two basic problems that create this software crisis are programmer productivity and program reliability. In this section I will describe one attack on these problems that was launched using mathematical logic and automated theorem-proving techniques. The other approach, taken by the school of *structured programming,* will not be described in this manuscript. Initially research in the field of logics of programs attacked only the reliability problem by developing methods for proving the correctness of programs. *Program verification* improves the reliability of programs but is not too helpful for productivity. *Program synthesis* tries to remedy this by going further and generating programs from specifications. This, unfortunately, is a problem too hard to solve in general, and research has yet to produce practical methods for program synthesis. An easier approach is to execute the specification of a problem directly. This is what *logic programming* is all about. In the rest of this section I will elaborate on the three approaches that use logic to help solve the software crisis.

1.1.1 Program verification

The standard approach to program verification (cf. [Manna 74, Chapter 3]) is based on the Floyd-Hoare method. The paradigm of this approach is that the programmer supplies the verification system with two descriptions of the program: the program text, and logic specification of the problem the program is expected to solve. The verification process proves that the program actually satisfies the specification.

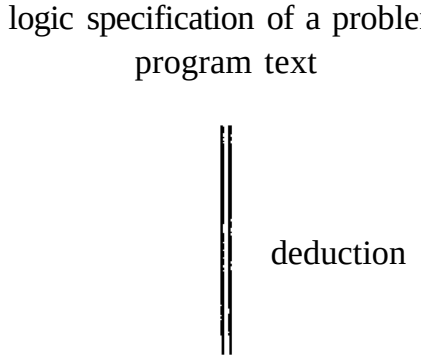
would imply the correctness of the program. Then a theorem prover is used to prove these verification conditions. Generally the user is expected to annotate the code with assertions at various points in the program to help both in the generation and proof of the verification conditions.

In most cases the data domain of the problem is too complex (usually undecidable) so the system should also be supplied with lemmas and axioms appropriate for the specific problem. If the programmer is lucky, the verifier runs for a while and then produces the message "verified," as illustrated below:



This is very nice—the program is now stamped as correct; productivity, however, is not improved because the verification process introduces extra work for the programmer.

Unfortunately, most (early versions of) programs are not correct; even when they are correct the system usually does not have an adequate amount of knowledge about the data domain or the program's intended behavior (as given by the programmer's annotations). In this case the scenario will be more like the one depicted in the following illustration:



A good verification system will also help the programmer to locate the bugs by printing the unprovable verification conditions; this will hint to the programmer what is wrong in the program or what lemmas about the data domain should be added.

1.1.2 Program synthesis

Research in *program synthesis* tries to solve the correctness problem and at the same time improve productivity. When following the deductive approach to program synthesis we see the following scenario:

logic specification of a problem
symbolic inputs

deduction

an applicative program

This is a very interesting and promising approach; the main difficulty, however, is that the general problem of synthesizing a program from specification is too hard for current theorem-proving technology. This is where *logic programming* enters the scene.

1.1.3 Logic programming

Logic programming promises to improve programmers productivity by replacing the standard machine-oriented programming languages by logic, a human-oriented language. This will be attractive only if a reasonable efficiency can be achieved while keeping the language intuitive to use.

This paradigm is exemplified by PROLOG, the most widely used logic programming language, but is not restricted to the use of the Horn-clause subset of logic. TABLOG, the language discussed throughout this dissertation, is a more general example of this approach.

1.2 Tablog

This work describes TABLOG, a new logic-programming language that incorporates a much broader subset of predicate logic than PROLOG and allows programs to be much clearer to read and write. The basic features of TABLOG are described and the various properties of the language are studied.

In addition to being less restricted in syntax than PROLOG, TABLOG is also cleaner from a logical point of view: predicate and function symbols are disjoint, negation is the standard logical negation, and a sound unification procedure is used. These properties are not generally true of PROLOG and its interpreters.

TABLOG incorporates advantages of two of the leading programming languages for symbolic manipulation—PROLOG and LISP. This is done by incorporating both relations and functions together and adding the power of unification as a binding mechanism.

The execution of TABLOG programs is based on the deductive-tableau proof system of [Manna and Waldinger 80], which was developed for program synthesis and is described in Chapter 3.

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.

1.3 Structure of the dissertation

The background material in Chapter 2 gives an overview of logic programming and PROLOG; this chapter also covers the basic concepts of first-order logic used in the rest of the presentation. Virtually all the material in this chapter is well known so most readers can skip it. Chapter 3 describes the deductive-tableau proof system. The presentation is somewhat different from other presentations of this proof system and includes some modified inference rules so reading this chapter is recommended.

Chapter 4 introduces TABLOG, its syntax and logical interpretation; it is followed by a collection of examples in Chapter 5.

Chapter 6 is wholly devoted to the comparison of TABLOG to PROLOG and LISP. The comparisons and examples in this chapter are intended to shed more light on understanding the procedural semantics of TABLOG. Points of similarity and difference are also brought up throughout the other chapters.

The procedural interpretation of TABLOG is detailed in Chapter 7 followed by a short description of the TABLOG interpreter implementation in Chapter 8. A user manual for using this system is given in Appendix B.

Chapter 9 investigates the theory of computation of TABLOG programs and their interpretation.

Chapter 10 describes a model for the parallel execution of TABLOG programs.

Chapter 11 compares this work to related research efforts in functional and logic programming and their combination.

The conclusion chapter, Chapter 12, summarizes the advantages and disadvantages of the TABLOG experiment and suggests directions for future work.

Definitions, examples, and theorems are numbered using the same numbering sequence, so for example we might find Theorem 3.5 following Definition 3.4 and preceding example 3.6; in such a case there will be no Theorem 3.4 or Theorem 3.6.

CHAPTER 2

LOGIC PROGRAMMING

This chapter supplies the background material on first-order predicate logic together with an overview of logic programming in general and PROLOG in particular. Readers familiar with these subjects can skip this chapter.

2.1 Introduction

Though in many circles the term *logic programming* is considered synonymous with programming in PROLOG, this is a very narrow interpretation. A more general definition refers to the use of (some subset of) first-order predicate logic as a programming language. TABLOG falls under this definition. Once we follow this wider interpretation we can probably call PROLOG and its variants *relational-programming* languages given their emphasis on using predicates to describe computations.

An even broader definition of the concept of logic programming includes any programming language that is based on a formal logic system. Under this broader sense, (pure) LISP for example is also a logic programming language based on the λ-calculus, and so are the various languages based on equational logic. While LISP is so well known that it needs no introduction, I will describe some of the other approaches to logic programming in the broad sense in Chapter 11.

The idea of using first-order predicate logic as a language for problem solving has already been proposed by [Waldinger 69] and [Green 69]; both used mechanical theorem proving to search and discover solutions. The founders of (first-order based) logic programming however are considered to be Kowalski ([Kowalski 74]) and Colmerauer who proposed using Horn clauses as a programming language and experimented with their implementation. Their decision to adhere to this restricted form of logic helped logic programming to start gathering momentum. The implementation of an experimental system started around 1972 at the University of Marseille and is described in [Colmerauer, Kanoui, and van Caneghem 79]; this system and the demonstration of an efficient compiler for the language ([Warren

The research in logic programming in general and PROLOG in particular have attracted even more interest after the Japanese Fifth Generation Computer Project chose PROLOG as the programming language for knowledge-based applications, which are considered to be of major importance to this project. Before describing PROLOG we should first review first-order predicate logic, which is the language behind PROLOG and TABLOG.

2.2 First-order logic

First-order logic is the basis for PROLOG and many of its extensions as well as for TABLOG. In this section I will describe the syntax of the full first-order predicate logic with equality; later it will be shown which subsets are used for PROLOG and TABLOG. The syntax described here is somewhat richer than the standard language of first-order logic, but all the additional connectives can be easily defined using the classical ones.

2.2.1 Alphabet

The alphabet of the language of first-order logic consists of the following syntactic categories:

- Truth constants: *true* and *false*.
- Logical connectives: $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\equiv$ (equivalence), $\rightarrow$ (implication), $\leftarrow$ (reverse implication), and the conditional *if-then-else*.
- Individual variables denoted by symbols such as u, v, x_1, y_{25}.
  V will denote the set of all the variables in the language.
- Individual constants such as a, b, [], 5.
- Predicate symbols including $=$ (equality), **prime**, $\in$, $\geq$.
- Function symbols such as **gcd**, **append**, $+$.
- Quantifiers: $\forall$ (universal quantifier, for-all) and $\exists$ (existential quantifier, for-some).
- Punctuation: '(', ')', and ','.

The syntactic categories just mentioned are pairwise disjoint and every symbol of the language must belong to exactly one of them.

Although all the propositional connectives can be expressed in terms of $\vee$ and $\neg$, the procedural interpretation might be different for connectives which are logically equivalent. Therefore, we cannot, for example, consider $u \rightarrow v$ to be merely an abbreviation for $\neg u \vee v$ but as a different connective.

We consider basic arithmetic to be part of our language, so all the integers are constants, and the basic arithmetic predicates and functions such as $+$, $-$, $\geq$, **odd** are also included.

2.2.2 Terms and formulas

The language of logic uses *formulas* to describe properties of objects represented by *terms*. Terms and formulas are both (finite) strings over the alphabet described in the previous subsection.

Definition 2.1. *Terms*

- An individual constant c is a term.
- An individual variable v is a term.
- If $t_1, \ldots, t_n$ are terms and f is a function symbol, then

  $$f(t_1, \ldots, t_n)$$

  is a term. (Where n is the *arity* of the function symbol f.)
- If $\mathcal{F}$ is a (well-formed) formula (as defined below) and t_1 and t_2 are terms, then

  $$\text{if } \mathcal{F} \text{ then } t_1 \text{ else } t_2$$

  is a term. ▟

Σ will denote the set of all terms in language that can be generated using the definition above.

Definition 2.2. *Ground terms*

- A ground term is a term with no variables. ▟

In the variant of the language used here, the *if-then-else* construct can be used to build terms out of formulas and simpler terms. This operator makes the definition of terms and formulas mutually recursive. Note that we use this construct both as a logical connective for formulas, as will be seen below, and as an operator generating terms, as in the definition above. Although these are two different operators, we can overload the symbol since it is always clear from the context which of the two is meant.

Definition 2.3. *Formulas*

- The truth constants

  true and *false*

  are atomic formulas.
- If $t_1, \ldots, t_n$ are terms and p is a predicate symbol, then

  $$p(t_1, \ldots, t_n)$$

  is an atomic formula, where n is the *arity* of the predicate symbol p.
  In particular, $t_1 = t_2$ is a formula for any terms t_1 and t_2.
- If $\mathcal{F}_1, \mathcal{F}_2$, and $\mathcal{F}_3$ are atomic or compound formulas, then

  $$(\mathcal{F}_1 \wedge \mathcal{F}_2)$$
  $$(\mathcal{F}_1 \vee \mathcal{F}_2)$$
  $$(\mathcal{F}_1 \rightarrow \mathcal{F}_2)$$
  $$(\mathcal{F}_1 \leftarrow \mathcal{F}_2)$$
  $$(\mathcal{F}_1 \equiv \mathcal{F}_2)$$
  $$(\neg \mathcal{F}_1)$$
  $$(\text{if } \mathcal{F}_1 \text{ then } \mathcal{F}_2 \text{ else } \mathcal{F}_3)$$

  are all compound formulas.
- If $\mathcal{F}$ is a formula and v is a variable, then

  $$(\forall v)\mathcal{F} \quad \text{and} \quad (\exists v)\mathcal{F}$$

  are (quantified) compound formulas.
  $\mathcal{F}$ is the *scope* of the quantifier ($\forall v$ or $\exists v$) and it is also called the matrix of the formula. The free occurrences of v in $\mathcal{F}$ are *bound* by that quantifier. If a variable u is not bound by a quantifier $\forall u$ or $\exists u$, it is said to be a *free* variable. ⌟

Conventionally, formulas and terms involving certain functions and predicates are written in infix or postfix form (e.g., $3 + x$, $4!$, $z \leq 3$). For readability we will write our formulas this way, and will often use brackets instead of parentheses and will omit certain parentheses according to the usual conventions on the relative binding power of the functions, predicates, and connectives involved.

Definition 2.4. *Expressions and subexpressions*

- An *expression* is either a term or a formula.
- The *subexpressions* of an expression are all the terms and formulas in it. ⌟

Example 2.5.
If gcd is **a** binary function symbol and | is an infix binary predicate, then the compound formula

$$gcd(x,y)\backslash x \text{ A } gcd(z,y)|y \text{ A } (Vz)[(z\backslash x \text{ A } z\backslash y) \text{ -> } z\backslash gcd(x,y)]$$

has two free variables, *x* and y, and a bound variable, *z.* The terms in this formula are '#', 'y?, 'z\ and gcd(rp, *y)*\ The atomic formulas are **'gcd(;r,** *y)\x* **'gcd(#,** *y)\y* *'z\x\ 'z\y* and **C2|gcd(#,** *y)*\ The whole formula is a conjunction and, although the conjunction connective A is a binary infix connective, we can use associativity to eliminate the parentheses that group the conjuncts in pairs. The formula

$$(z\backslash x \text{ A } z\backslash y) \text{ -• } 2|\text{gcd(a;,y)}$$

in the scope of the quantifier *Wz* is an implication with *antecedent* $^{L}z\backslash x$ A $z\backslash y^{y}$ and *consequent 'z\gcd(x,y)*\ The subexpressions of the formula include all the terms and atomic formulas mentioned above in addition to the formula itself and other compound formulas like the antecedent of the implication just described. ▟

When an associative construct, like the conjunction connective A, is used in an expression of the form

$$\text{Pi A P}_2 \text{ A } \bullet\bullet\bullet \text{ A } P_n$$

our convention will be to regard it as an rc-ary connective (operator in general) and the subexpressions will be $\text{Pi}, \text{P2}, \ldots, P_n$ and their subexpressions in addition to the whole conjunction itself. Other conventions, which enumerate all possible ways to write the conjunction as composition of binary ones, will result in a much bigger set of subexpressions; in the context of this dissertation we will not need this bigger set.

When we use logic to describe programs in TABLOG and PROLOG, we do not specify the arity of the function and predicate symbols, so the same symbol can actually be used to represent more than one function or relation and the arity (and the appropriate function or relation) can be inferred from the context.

2.2.3 Clausal form

Once logic is used to describe problems, we want to have a mechanical way to solve these problems. [Robinson 65] introduced the resolution principle that can be used to prove any valid sentence of first-order logic by refuting its negation. This classical resolution requires that formulas be converted to clausal form before they can be refuted. Clausal form is a conjunctive normal form representing a sentence as a conjunction of a set of disjunctions.

Definition 2.6. *Clauses and clausal sentences*

- *Clausal sentence:* a set of *clauses.*
- *Clause:* a set of *literals.*
- *Literal:* an atomic formula or its negation. ▟

Example 2.7.

The sentence (in clausal form)

$$\{\{p(a), \neg q(a)\}, \{\neg p(x)\}, \{q(y), \neg p(a)\}\}$$

can also be written as

$$b(a)V\text{-}_9(a)] \ A \ hp(x)] \ A \ [q(y) \ V \ ^>(a)]$$

where each conjunct corresponds to a clause.

Another way to write clauses is to collect all the *positive* literals together and all the *negative* literals together and to write an arrow between them. For example the clause

$$\text{-ip(a)} \ V \ ^q(x) \ V \ r(v) \ V \ \text{-»r(2)} \ V \ p(b)$$

which is equivalent to

$$p(a) \ Aq(x) \ Ar(2) \ ^ \ r(v)V \ p(b)$$

is written in arrow notation as

$$r(v),p(b) \ <\text{-} \ p(a),_9(ar), r(2).$$

If the left-hand side of the clause in this notation is empty, it is taken (as an empty disjunction) to be *false.* If the right-hand side is empty, it is taken (as an empty conjunction) to be *true.* The empty clause, representing *true* —> *false,* is taken to be *false.*

There is a mechanical procedure to convert any first-order logic formula into clausal form. In the process of this conversion (e.g., [Manna 74] Chapter 2) all the quantifiers are removed by means of *skolemization.* The variables in each clause of a sentence should be disjoint from those of the other clauses.

2.2.4 Horn clauses

The subset of logic that was chosen as the language of PROLOG is the class of *Horn clauses* (also called *definite clauses*). Horn clauses have exactly one *positive* literal so when they are written using the arrow notation there will be exactly one literal on the left-hand side of the arrow.

Definition 2.8. *Horn sentence*

- *Horn (definite) sentence*: a set of Horn clauses.
- *Horn (definite) clause*:

$$L \leftarrow L_1, \ldots, L_m$$

where $m \geq 0$ and L and all the L_i's are atomic formulas.
When $m = 0$ the clause is

$$L \leftarrow \mathit{true}$$

which can be written simply as L.
When $L = \mathit{false}$ the clause

$$\mathit{false} \leftarrow L_1, \ldots, L_m$$

represent a negated conjunction and is called a *goal.* and can be written as

$$\leftarrow L_1, \ldots, L_m.$$ ∎

While every sentence in logic can be translated into clausal form, this is not true for Horn clauses. The language of Horn clauses is not as expressive as first-order logic.

2.3 Unification

Unification is a generalization of pattern matching and is used as the binding mechanism for both PROLOG and TABLOG. It is the process of making two (or more) expressions equal by substituting terms for variables. A unification algorithm is described in [Robinson 65] together with the introduction of resolution. Because unification is essential for any resolution-based theorem proving, the unification problem has been a center of intensive study. [Wegman and Paterson 78] and independently [Martelli and Montanari 82] designed linear- (and almost-linear-) time algorithms for finding a unifying substitution (unifier). A simple unification algorithm implemented in TABLOG is given in Section 5.7. Recently it was proved in [Dwork, Kanellakis, and Mitchell 84] that unification is complete for polynomial time (under log-space reductions), which means theoretically that unification is sequential in nature, and the amount of parallelism we can hope to introduce into algorithms for finding a unifier is limited. This result may not have practical importance because, for example, it does not exclude any constant speed-up of unification using a parallel algorithm.

Recall that we denote by V the set of all the variable symbols in the language, and by E the set of terms.

Definition 2.9. ***Substitutions and applications***

- A *substitution* is a function $9 : V \to$ E.
  Most commonly, *6* maps all but a finite number of variables to themselves. In this case *6* can be represented as a set

  $$\{(v_i, <!), \ldots, (v_m, <_m\text{»},$$

  with $V\{ \wedge Vj$ if $i \wedge j$ and $U \wedge v_t$ for all i, *j*.
- The *application* of the substitution *0* to the expression *E* is denoted by *E6* and is defined recursively:
  - for a variable v, *vO* is *0(v).*
    If we use the set notation, it means that if *{v,t)* £ *0* then *v9 = t.* Also for a variable *v* such that for no term J, (v,t) G *0* we have ***v6 = v.***
  - for a constant c, *cO* = c.
  - for an expression $L(e_i, \ldots, e_n)$ where *L* is a function, predicate or a connective and each e* is an expression:

$$L(e_1, \ldots, e_n)\theta = L(e_1\theta, \ldots, e_n\theta).$$ ∎

Definition 2.10. *Unifier*
A substitution θ is the *unifier* of the set of expressions $\{e_1, \ldots, e_m\}$ if

$$e_1\theta = e_2\theta = \cdots = e_m\theta.$$

We will sometimes refer to a unifier as a *unifying substitution.* In general a unifier (when it exists) is not unique. We are interested in a *most general unifier*:

Definition 2.11. *Most general unifier*
A unifying substitution θ is a *most general unifier* of $\{e_1, \ldots, e_m\}$ if it is a unifier for this set of expressions and, for any unifier λ of $\{e_1, \ldots, e_m\}$, there exists a substitution ρ such that $\theta\rho = \lambda$, i.e., for any expression $\mathcal{E}$, $(\mathcal{E}\theta)\rho = \mathcal{E}\lambda$.

Example 2.12.

- Two most general unifiers of

  $$f(x, y) \text{ and } f(g(u, v), u)$$

  are

  $$\{x \leftarrow g(u, v),\ y \leftarrow u\}$$
  and
  $$\{x \leftarrow g(w, z),\ u \leftarrow w,\ y \leftarrow w,\ v \leftarrow z\}$$

  The unifier

  $$\{x \leftarrow g(u, 2),\ y \leftarrow u,\ v \leftarrow 2\}$$

  is not most general. Whenever two expressions are unifiable there always exists a most general unifier.

- The terms

  $$f(x, g(y, x), y) \text{ and } f(g(u, v), v, u)$$

  are not unifiable because, to make the two expressions equal, x must be replaced by an expression containing x which, in turn, makes the two expression different again. The check for this case, which is called the occur-check, is discussed in Section 6.4.

While unification is a syntactic process, depending only on the set of expression that we are trying to unify, it can be extended to depend on a given set of equations. Such an extension allows the expressions to be rewritten using the equations to make them unifiable. This extended procedure is sometimes called *semantic unification.* Some extensions of PROLOG use this process for the introduction of equality with explicit assertions to specify the properties of equality.

2.4 Resolution

Resolution was introduced in [Robinson 65] as a complete proof rule for first-order predicate logic that is suitable for mechanical deduction. While the next chapter describes, as part of the deductive-tableau proof system, a generalized resolution rule that can be applied to any sentence in logic, classical resolution works only on sentences in clausal form. The resolution rule is used in a refutation procedure where the theorem to be proved is negated before it is converted to clausal form; then the rule is applied until the empty clause is derived. When the empty clause (which is the same as the empty disjunction, i.e., *false*) is derived, the sentence is refuted and the proof is complete.

The resolution rule is applied to two clauses in a sentence; it eliminates a literal that occurs in both clauses and produces a new clause to be added to the sentence. The occurrences of the literal must be of opposite polarity, i.e., negated in one clause and unnegated in the other.

Definition 2.13. *Resolution*
If a clausal sentence contains the two clauses

$$\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k, \mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_m\},$$

and

$$\{\neg\hat{\mathcal{P}}_1, \neg\hat{\mathcal{P}}_2, \ldots, \neg\hat{\mathcal{P}}_\ell, \mathcal{Q}'_1, \mathcal{Q}'_2, \ldots, \mathcal{Q}'_n\},$$

and if $\{\mathcal{P}_1, \ldots, \mathcal{P}_k, \hat{\mathcal{P}}_1, \ldots, \hat{\mathcal{P}}_\ell\}$ are unifiable with a most general unifier θ, we can add to the sentence the new clause

$$\{\mathcal{Q}_1\theta, \mathcal{Q}_2\theta, \ldots, \mathcal{Q}_m\theta, \mathcal{Q}'_1\theta, \mathcal{Q}'_2\theta, \ldots, \mathcal{Q}'_n\theta\}.$$

This new clause is the *resolvent* of the two given clauses. ⌟

The soundness of the resolution principle is the fact that the original sentence is unsatisfiable if and only if the sentence that we get after the addition of the resolvent is.

Example 2.14.
Given
$\{\{p(x), \neg q(b)\},$
$\{q(y), q(u), \neg q(a)\}\}$
we can use the unifier $\{y \leftarrow b\}$ to resolve the two clauses and add the clause $\{p(x), q(u), \neg q(a)\}$. ⌟

This example is an instance of *binary resolution* as we have matched one literal from each clause. Using the general form of the rule, we can get the new clause

$$\{p(x), \neg q(a)\}$$

as both positive q literals in the second clause can participate in the resolution.

2.4.1 SLD resolution

SLD resolution is a refinement of the general resolution principle for the language of definite sentences. The name SLD stands for linear resolution with a selection function for definite sentences.

Since a definite sentence has exactly one positive literal in every clause the resolution rule becomes much simpler.

In SLD resolution we apply the resolution to a goal and a clause to get a new goal.

Given a goal of the form

$$\{\neg Q, \neg P_1, \neg P_2, \ldots, \neg P_m\}$$

and a rule clause of the form

$$\{Q, \neg Q_1, \neg Q_2, \ldots, \neg Q_n\}$$

we can derive the new goal clause

$$\{\neg Q_1, \neg Q_2, \ldots, \neg Q_n, \neg P_1, \neg P_2, \ldots, \neg P_m\}$$

The S in the name of the proof procedure is for the *selection function* that chooses the literal in the goal to be resolved upon (i.e., the one specified as Q in the description above). Each of the rule clauses has exactly one positive literal in it (the clause head) so once the clause is selected this literal is always chosen. The resolution always starts from the goal and keeps generating new goals until the empty goal is derived.

2.5 Prolog

PROLOG was the first logic-programming language to be widely recognized as such. It is based on Horn clauses, which were described in Section 2.2. PROLOG uses an SLD resolution theorem prover as the computation engine. This language is based on the work of Kowalski and his colleagues on linear resolution, connection graphs, and related subjects, and the first implementation was realized in Marseille. Today PROLOG is a popular programming language and is used by people having no previous contact with logic. [Clocksin and Mellish 1981] is the standard introductory text for PROLOG while [Kowalski 79] offers a more general treatment of (relational) logic programming. [Lloyd 84] studies PROLOG but in a more theoretical manner than any of the other books.

The language of PROLOG consist of three types of Horn clauses:

- *Rules* are clauses of the form

$$P \leftarrow Q_1, \ldots, Q_m$$

  with *head* P, and *body* Qi, ..., Q_m.

- *Facts* are Horn clauses with empty body

$$P \leftarrow$$

  and are written as

  P.

- *Goals* axe Horn clauses with no head

$$\leftarrow G_1, \ldots, G_m.$$

  Goals are negated conjunctions and are also called *queries;* they are simply written as

  d A... AG_m.

  The context (and usually the indentation) will tell when a formula is a goal.

While a definite sentence was defined earlier to *be & set* of Horn clauses, a PROLOG program is a *list* of (fact and rule) clauses. The order of the clauses in a program is important to the procedural interpretation of the language.

Each clause in a PROLOG program defines the predicate in the head of the clause using the body of that clause. If the same predicate occurs in the head of more

than one clause, the different clauses are considered alternative definitions. A call to a PROLOG program is a goal clause which is a query to be answered; this is done by showing (using SLD resolution) that the goal (i.e., the negated conjunction) is inconsistent with the clauses of the program. The proof proceeds by unifying a conjunct in the goal against the heads of the program clauses; if the unification succeeds, the conjunct is replaced in the goal by the body of the corresponding clause. The clauses defining each predicate are used according to the order of their appearance in the program. Alternative definitions will be used only if the computation fails because the leftmost conjunct of the new goal does not match any clause head. The PROLOG interpreter uses depth-first search to explore the proof tree that consists of the possible applications of the inference rule. When a dead end is encountered, the interpreter backtracks to the last branching point where there was a choice of the clause to use.

Example 2.15. *Append*
The following classical PROLOG example is a program for concatenating two lists, rewritten here in the syntax used throughout this document.

appendp([ ], v, v).
appendp($x \circ u$, v, $x \circ w$) $\leftarrow$ **appendp**(u, v, w).

The first clause states that the **appendp** relation holds between the empty list [], an arbitrary list v and this same v. This represents the fact that when appending the empty list in front of any other list v, the result is v.

The second clause states that the relation **appendp** holds for three lists $x \circ u$, v, and $x \circ w$, if the first and last lists have the same head (represented by x here), and in addition, the **appendp** relation holds for u, the tail of the first list, v, the second list, and w the tail of the third list.

Typically the third arguments of the **appendp** predicate is intended to be the output, and the program above prescribes an algorithm to append two arbitrary lists. This use of the arguments is not specified and this same program can be used in different ways to decompose lists rather than to append them.

More discussion of PROLOG and its features can be found in Chapter 6 as part of the comparison between TABLOG and PROLOG. The syntax of PROLOG is limited; it allows only Horn clauses and thus eliminates the use of connectives like negation, disjunction, and equivalence. Equality with its standard meaning is not included in the language and the use of functions is limited to data structures. In recent years we have seen a growing number of research efforts to extend PROLOG and eliminate some of these shortcomings. Some of these approaches are described in Chapter 11.

CHAPTER 3

THE DEDUCTIVE-TABLEAU PROOF SYSTEM

This chapter gives a summary of the Manna-Waldinger deductive-tableau proof system [Manna and Waldinger 80 and 86]. Only the deduction rules actually employed in the proof system when it is used as the TABLOG interpreter are detailed here. Other rules are just briefly mentioned; refer to the original articles for more detail. For a thorough treatment of the proof system, wait for Volume 2 of [Manna and Waldinger 85].

The deductive-tableau proof system, which is used as a program interpreter for TABLOG, was originally developed for program synthesis. There are LISP and PROLOG implementations of the proof system in interactive systems for program synthesis, as described in [Malachi 82], [Bronstein 83], and [Yellin 83]. [Stickel 82] combines the nonclausal resolution rule of this proof system with connection graphs to yield an automatic theorem prover that has been incorporated into a natural-language understanding system.

3.1 The deductive tableau

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. These formulas can contain all the logical connectives and quantifiers but as will be seen below may contain free variables which will have implicit quantification.

Here is the general form of a deductive tableau

Assertions	Goals
$\mathcal{A}_1(\bar{u}_1)$	
$A_2(\bar{u}_2)$	
	0i(t>i)

.
.
.

$A_m(\bar{u}_m)$	
	$Qn(\bar{V}n)$

where $\bar{U}i$ is used as an abbreviation for ti»u • • • ?^{u}%ki •

This tableau is semanticaJly equivalent to the sentence

$$(\forall \bar{u}_1)\mathcal{A}_1(\bar{u}_1) \wedge \cdots \wedge (\forall \bar{u}_m)\mathcal{A}_m(\bar{u}_m) \rightarrow (\exists \bar{v}_1)\mathcal{G}_1(\bar{v}_1) \vee \cdots \vee (\exists \bar{v}_n)\mathcal{G}_n(\bar{v}_n)$$

which is also equivalent to

$$(\exists \bar{u}_1, \ldots, \bar{u}_m, \bar{v}_1, \ldots, \bar{v}_n)\,[\neg\mathcal{A}_1(\bar{u}_1) \vee \cdots \vee \neg\mathcal{A}_m(\bar{u}_m) \vee \mathcal{G}_1(\bar{v}_1) \vee \cdots \vee \mathcal{G}_n(\bar{v}_n)].$$

The last formula exhibits the duality between goals and assertions in the tableau. If we have the formula Q as a goal in the tableau, we can replace it with the assertion -*Q without changing the meaning of the tableau. Equivalently, we can replace an assertion A with the goal -*A.

The declarative or logical meaning of a tableau (which is formally defined by the sentence above) 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.

3.2 Proofs

A proof of a theorem using this system starts with the initial tableau, which contains the theorem as the initial goal, and any assumptions needed for the theorem to hold as assertions.

A proof is constructed by using deduction rules to add new goals to the tableau in such a way that, at each stage, the tableau is valid if and only if the initial tableau is. The inference rules that are used in the deduction are described in the rest of this chapter.

In the version of the tableau used for the TABLOG interpreter, the proof is complete when we have generated the goal *true;* this is an affirmation procedure. In the general scheme, we can alternatively derive the assertion *false,* which is equivalent to getting the empty clause in standard resolution; this is a refutation procedure.

When employing the tableau proof system as an interpreter for a TABLOG program we are interested in the binding of the *output variables,* (a subset of) the variables of the goal to be proved by the deduction. The final binding of these output variables is called the *answer substitution.* In order to record the binding throughout the deduction, we add to the tableau a third column, the *output column.* Using this notation in the description of the inference rules, it will be clear what effect each rule has on the binding.

In a TABLOG program, the output column is empty for assertions in the program, and it contains the output variable(s) in the goal calling the program. In the subgoals derived during the computation the output column contains the current binding of the output variables. If a substitution is made for an output variable as part of a derivation step, the newly introduced term appears in the output column of the derived goal.

Although in the declarative (and logical) semantics of the tableau the order of entries is immaterial, a proof procedure may take this order into account when chosing which instance of the deduction rules to apply. For example, the procedural interpretation of the tableau as a TABLOG program (to be described later) utilizes the order of the assertions in the program. This implies that changing the order of two assertions, or changing the order of the subformulas, in an assertion or goal, may produce different computations.

3.3 Nonclausal resolution

Before we study the details of this rule we should become familiar with the notation used in the description of the various inference rules. The square brackets are used to denote that an expression occurs as a subexpression of another. So when we write *A[P]* it is understood that the formula *P* occurs as a subexpression in the formula *A.* The notation is context sensitive; in the conclusion of the rule (below the double line) *A0[false]* represents the formula that is the result of applying the substitution *0* to *A* and then replacing *all* the occurrences of *P9* with *false;* note that the expression that was replaced is specified by one of the premises of the inference rule. Square brackets denote substituting all occurrences, and angle brackets will represent substituting *some* occurrences.

The nonclausal resolution rule allows removal of a subformula *V* from a goal *G[P]* by means of an appropriate assertion ^[$\hat{P}$]. If *0* is a (most general) unifier of *V* and $\hat{V}$, i.e.,

$$ve = \hat{v}e,$$

then a new goal can be added as illustrated below.

Assertions	Goals	Outputs
A[P)		
	G[$\hat{P}$]	***9***
	not A6\false] and ge[true]	***go***

The new output *gd* is *g* after the application of the unifier 0, and the conjunct *Q9[true]* represents *Q9* with all occurrences of $\hat{V}$*9* replaced by *true.*

Before applying the nonclausal resolution rule we must make sure that the two entries do not have any variables in common; we may have to rename some variables to ensure this. We shall assume this for all the inference rules that involve two entries.

The version of the rule just described is called *goal-assertion resolution;* a slightly different form of the rule is used when the matching subformula is replaced with *true* in the goal and with *false* in the assertion. This version will be called *assertion-goal resolution.* Later we will discuss how to chose the appropriate form based on the *polarity* of the subformula.

Thus the assertion-goal nonclausal resolution rule is

assertions	goals	outputs
$\mathcal{A}[\mathcal{P}]$		
	$\mathcal{G}[\hat{\mathcal{P}}]$	g
	$not\ \mathcal{A}\theta[true]$ and $\mathcal{G}\theta[false]$	$g\theta$

The soundness of both versions of the rule is proved by showing that the tableau is valid after the addition of the new goal if and only if it is valid before this inference. Such a proof can be done by case analysis, distinguishing between the interpretations for which $\mathcal{P}\theta$ is *true* and those for which it is *false*.

Given an assertion and a goal there are usually many ways to apply the resolution rule to them; even if there is only one matching subformula, we can replace it with *false* in the assertions and with *true* in the goal or vice versa. The choice of the unified subformulas is governed by the *polarity strategy*. The polarity strategy eliminates some of the fruitless goals that result from resolution with bad choice of the replacement.

Before stating this strategy, we must define the concept of polarity of an occurrence of a formula in a tableau.

An occurrence of formula 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. Assertions are considered positive, and because of duality every goal has an implicit negation applied to it. A subformula can occur both positively and negatively in a formula; this can happen either by having more than one occurrence of the formula or by having the formula within the scope of a connective that implies both polarities as will be shown below.

Formally, for a tableau $\mathbf{T}$ we will assign to every occurrence of a formula as an entry or as a subformula in an entry a polarity in $\mathbf{T}$ according to the following definition. To make the definition shorter the notion of *opposite polarities* is used; negative and positive polarities are the opposite of each other.

Definition 3.1. *Polarity of occurrences*

- An assertion $\mathcal{F}$ has positive polarity in $\mathbf{T}$
- A goal $\mathcal{F}$ has negative polarity in $\mathbf{T}$

- If a formula $\mathcal{E}$ is of form

$$\neg\mathcal{F},$$

then $\mathcal{F}$ has polarity in **T** opposite to that of $\mathcal{E}$;

- if a formula $\mathcal{E}$ is of form

$$\mathcal{F}_1 \wedge \mathcal{F}_2$$

or

$$\mathcal{F}_1 \vee \mathcal{F}_2,$$

then both $\mathcal{F}_1$ and $\mathcal{F}_2$ have the same polarity in **T** as $\mathcal{E}$;

- If a formula $\mathcal{E}$ is of form

$$\mathcal{F}_1 \rightarrow \mathcal{F}_2,$$

then the consequent $\mathcal{F}_2$ has the same polarity in **T** as $\mathcal{E}$, but the antecedent $\mathcal{F}_1$ has polarity in **T** opposite to that of $\mathcal{E}$;

- If a formula $\mathcal{E}$ is of form

$$\mathcal{F}_1 \equiv \mathcal{F}_2,$$

then $\mathcal{F}_1$ and $\mathcal{F}_2$ have both positive and negative polarities in **T** (independent of the polarity of $\mathcal{E}$ in **T**);

- If a formula $\mathcal{E}$ is of form

$$\text{if } \mathcal{F} \text{ then } \mathcal{G}_1 \text{ else } \mathcal{G}_2,$$

then the *then*-clause $\mathcal{G}_1$ and the *else*-clause $\mathcal{G}_2$ have the same polarity as $\mathcal{E}$ in **T**, while the *if*-clause $\mathcal{F}$ has both positive and negative polarities in **T**. ∎

Once we know the polarity of a formula in **T**, we can deduce the polarity of each of its components by applying one of the above rules. In particular we will be interested in applying this process to find the polarity of the atomic formulas in a tableau.

Example 3.2.
If **T** is the tableau

Assertions	Goals
$P_1 \rightarrow P_2 \vee P_3$	
	$[P_1 \wedge \neg P_2] \vee [P_4 \equiv P_5]$

then

both occurrences of P_1 have negative polarity in **T**,
the single occurrence of P_3 and both occurrences of P_2
have positive occurrence in **T**,

and

P_4 and P_5 both have both polarities in **T**.

Sometimes we may explicitly denote the polarity of occurrences in a tableau; using this notation the same tableau will be written as

Assertions	Goals
$P_1^- \rightarrow P_2^+ \vee P_3^+$	
	$[P_1^- \wedge \neg P_2^+] \vee [P_4^{\pm} \equiv P_5^{\pm}]$

Note that this notation for polarity is opposite to the one used in [Manna and Waldinger 80] but it conforms to the notation used in standard resolution theorem proving and by [Manna and Waldinger 86] and [Murray 82].

While the soundness of the nonclausal resolution rule (in both versions mentioned above) does not depend on the *polarity* of the subformulas $\mathcal{P}$ and $\hat{\mathcal{P}}$ being resolved, many useless applications of the rule can be eliminated if we follow the polarity strategy.

Definition 3.3. *Polarity strategy*

When applying the nonclausal resolution rule, an instance $\hat{\mathcal{P}}\theta$ of the subformula $\hat{\mathcal{P}}$ of $\mathcal{F}$ will be replaced by *false* in $\mathcal{F}\theta$ only if $\hat{\mathcal{P}}$ occurs in $\mathcal{F}$ with positive polarity in the tableau.

> Dually, (an instance $\mathcal{P}\theta$ of) the subformula $\mathcal{P}$ of $\mathcal{G}$ will be replaced by *true* in $\mathcal{G}\theta$ only if $\mathcal{P}$ occurs in $\mathcal{G}$ with negative polarity in the tableau. ⌟

[Murray 82] proves that nonclausal resolution system is complete for first order logic even under the restriction of the polarity strategy. The version used by TABLOG is not complete because the proof procedure does not use the versions of the rule that match two assertions or two goals. The version used here always unifies a pair of subformulas while the general rule allows unifying sets of subformulas (and thus takes care of factoring). The (in)completeness of the proof system used by TABLOG is discussed in detail in Chapter 9.

Note that the original formulation of the resolution rule, as described above, requires replacing all occurrences of $\mathcal{P}$ by *true* or by *false*, but for efficiency reasons and for parallel execution of proofs we might want to replace only some of them. The relaxed form of the rule is:

Assertions	Goals	Outputs
$\mathcal{A}\langle\mathcal{P}\rangle$		
	$\mathcal{G}\langle\hat{\mathcal{P}}\rangle$	g
	not $\mathcal{A}\theta\langle$*false*$\rangle$ *and* $\mathcal{G}\theta\langle$*true*$\rangle$	$g\theta$

where everything is the same as for the original rule except that $\mathcal{G}\theta\langle$*true*$\rangle$ is $\mathcal{G}\theta$ with *some* occurrences of $\hat{\mathcal{P}}\theta$ replaced by *true*, and $\mathcal{A}\theta\langle$*false*$\rangle$ is obtained by replacing some occurrences of $\mathcal{P}\theta$ by *false*.

The soundness of this relaxed version of the rule can also be proved in using case analysis.

3.4 Equality rule

An asserted (possibly conditional or otherwise embedded in a larger formula) 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. The general form of the rule is:

assertions	goals	outputs
$\mathcal{A}[s = t]$		
	$\mathcal{G}\langle\hat{s}\rangle$	g
	not $\mathcal{A}\theta[\mathit{false}]$ *and* $\mathcal{G}\theta\langle t\theta\rangle$	$g\theta$

where θ is a unifier of s and $\hat{s}$, i.e.,

$$s\theta = \hat{s}\theta,$$

and $\mathcal{A}\theta[\mathit{false}]$ is $\mathcal{A}\theta$ after all occurrences of the equality $s\theta = t\theta$ (which, by the polarity strategy, can be required to occur with positive polarity) have been replaced by *false*, and where $\mathcal{G}\theta\langle t\theta\rangle$ is $\mathcal{G}\theta$ after the replacement of some occurrences of the term $s\theta$ by $t\theta$.

In the original tableau proof system we can also use the same rule to replace a term by another if the replaced term matches the left-hand side of an equality; allowing these two versions of the rule makes it a generalization of paramodulation.

3.5 Equivalence rule

This rule allows 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.

assertions	goals	outputs
$\mathcal{F}[\mathcal{P} \equiv \mathcal{Q}]$		
	$\mathcal{G}\langle\hat{\mathcal{P}}\rangle$	g
	not $\mathcal{F}\theta[\mathit{false}]$ *and* $\mathcal{G}\theta\langle\mathcal{Q}\theta\rangle$	$g\theta$

where θ is a unifier of $\mathcal{P}$ and $\hat{\mathcal{P}}$, i.e., $\mathcal{P}\theta = \hat{\mathcal{P}}\theta$.

3.6 Splitting

It is possible to split assertions and goals in some cases and get simpler ones without changing the meaning of the tableau. A conjunctive assertion is equivalent to a collection of assertions (goals) each containing one conjunct (disjunct), while an implicative goal can be split into an assertion and a goal.

For instance,

assertions	goals
$\mathcal{F}_1 \wedge \mathcal{F}_2 \wedge \mathcal{F}_3$	
$\mathcal{F}_1$	
$\mathcal{F}_2$	
$\mathcal{F}_3$	

Dually, a disjunctive goal can be split to produce goals containing the individual disjuncts. This splitting rule is not utilized by the sequential TABLOG interpreter but is useful for the parallel interpretation of TABLOG programs.

assertions	goals	outputs
	$\mathcal{F}_1 \vee \mathcal{F}_2 \vee \mathcal{F}_3$	g
	$\mathcal{F}_1$	g
	$\mathcal{F}_2$	g
	$\mathcal{F}_3$	g

Another possible splitting rule (not mentioned in the other descriptions of the system) involves splitting an implicative assertion with a conjunctive consequent,

assertions	goals	outputs
$\mathcal{A} \rightarrow [\mathcal{F}_1 \wedge \mathcal{F}_2 \wedge \mathcal{F}_3]$		g
$\mathcal{A} \rightarrow \mathcal{F}_1$		g
$\mathcal{A} \rightarrow \mathcal{F}_2$		g
$\mathcal{A} \rightarrow \mathcal{F}_3$		g

As will be seen later, this rule is useful to split the joint definition of two or more functions or predicates in TABLOG into separate definitions.

3.7 Other rules

The other rules that are included in the deductive-tableau framework but not described here (as they are not used in TABLOG), are:

Mathematical induction based on the complete induction principle using problem-dependent well-founded ordering. *Relation-matching* rules generalize the nonclausal resolution to cases where the expressions in the two entries involved are not unifiable but have some relation between them. Heuristics must be used to determine when to apply these powerful and general rules.

Transformation rules allow replacing terms or formulas by equivalent ones under given conditions. These rules can be used for propositional *true-false* simplifications that are built into the TABLOG system. In most other cases the effect of these rules can instead be achieved using the equality and equivalence rules.

Simplification involves performing simple valid transformations on a formula to get an equivalent but simpler formula. Both propositional and arithmetic simplification are performed automatically by the TABLOG interpreter after each of the other inference rules.

While nonclausal resolution and the equivalence rule can be generally performed by unifying arbitrary subformulas, the TABLOG interpreter applies these deduction rules by unifying atomic subformulas only. This restriction does not affect completeness while it makes automatic selection of a subformula to use in the deduction simpler. Since TABLOG is a programming language and not a general problem-solving system it also makes sense to directly reduce functions and predicates but not complicated formulas.

CHAPTER 4

TABLOG

4.1 Syntax

The syntax of TABLOG is that of the quantifier-free first-order predicate logic. This syntax consists of all the constructs described in Section 2.2 except for the quantifiers 3 and V. This includes of course the equality predicate (=), which is used heavily in functional programs.

The predefined function symbol o stands for the list insertion operator **(cons** in LISP) and serves as the main data constructor of TABLOG; the empty list is represented by []. Lists can be written using the convenient square brackets construct, as in [1,2,3], but this is considered a shorthand for lo(2o(3o[])).

All the integers are considered predefined constants, as are basic arithmetic predicates and functions such as +, —, ≥, **odd.**

As was mentioned already, the *if-then-else* construct is used both as a connective for formulas and as an operator generating terms; it will be demonstrated later how this construct is very useful in writing LISP-style programs.

4.1.1 Defining formulas

Although the basic syntax of TABLOG allows arbitrary formulas in logic in order to define functions and relations in a meaningful way, we have to restrict the class of formulas that are allowed in the assertions of a program. For this purpose we will introduce the notion of *defining formulas.* We will also prescribe the functions and relations *defined* by each such formula. The set of functional terms defined by the formula *T* is denoted by *Df\T* the atomic formulas defined by *T* are in D_T\T\. Essentially each member of *Df[T]* U *DriP]* can be regarded as the head of a procedure defined by the formula. More details on the way the assertions are used procedurally are given in chapters 5, 7, and 8.

Definition 4.1. *Defining formulas*

- If p is a nonprimitive predicate symbol (but not equality), *i,<2» • • • >*n are terms, and *T* is an arbitrary formula, then

$$p(t_1, \ldots, t_n),$$
$$\neg p(t_1, \ldots, t_n),$$
$$p(t_1, \ldots, t_n) \equiv \mathcal{F},$$

and

$$\neg p(t_1, \ldots, t_n) \equiv \mathcal{F},$$

are all *defining formulas* for the predicate symbol p, with

$$D_r[p(t_1, \ldots, t_n)] =$$
$$D_r[\neg p(t_1, \ldots, t_n)] =$$
$$D_r[p(t_1, \ldots, t_n) \equiv \mathcal{F}] =$$
$$D_r[\neg p(t_1, \ldots, t_n) \equiv \mathcal{F}] =$$
$$\{p(t_1, \ldots, t_n)\},$$

and

$$D_f[p(t_u \ldots, t_n)] =$$
$$D_f[\neg p(t_1, \ldots, t_n)] =$$
$$D_f\backslash p(t_u \ldots, t_n) \equiv f\backslash =$$
$$D_f[\neg p(t_1, \ldots, t_n) \equiv \mathcal{F}] =$$
$$\{\}.$$

- If # is a nonprimitive function symbol and $i_i, \ldots, t_n$, and <' are terms, then

$$g(t_1, \ldots, t_n) = t'$$

is a defining formula for the function symbol #, with

$$D_f[g(t_1, \ldots, t_n) = t'] = \{g(t_1, \ldots, t_n)\},$$

and

$$D_r[g(t_u \ldots, t_n) = t^{\#}] = \{\}.$$

• **If V_1 and $Z\geq 2$ are defining formulas and T is an arbitrary formula, then**

$$\mathcal{D}_1 \wedge \mathcal{D}_2,$$
$$\mathcal{D}_1 \vee \mathcal{D}_2,$$
$$\mathcal{F} \rightarrow \mathcal{D}_1,$$
$$\mathcal{D}_1 \leftarrow \mathcal{F},$$

and

$$\textit{if } T \textit{ then } V_1 \textit{ else } V_2$$

are all defining formulas as well with

$$D_f[D_x \wedge V_2] = D_f[V_1 \vee V_2] = D_f[\textit{if } T \textit{ then } V_x \textit{ else } V_2] = \mathcal{D}_f[I_1] \cup D_f$$

and

$$\mathcal{D}_r[2?_1 \wedge V_2] = P_r[2_1 \vee \mathcal{D}_2] = \neg D_r[\neg V_1 \wedge \neg V_x \textit{ else } V_2] = D_r[I_1] \cup D_r$$

while

$$D_f[\mathcal{F} \rightarrow \mathcal{D}_1] = D_f[\mathcal{D}_1 \leftarrow \mathcal{F}] = D_f[\mathcal{I}_1], \text{ and}$$
$$D_r[\mathcal{F} \rightarrow \mathcal{D}_1] = D_r[\mathcal{D}_1 \leftarrow \mathcal{F}] = D_r[\mathcal{D}_1].$$ ∎

Note that the formulas T appearing in the definitions above are unrestricted formulas (including negation, equivalence and equality).

Example 4.2.

• The formula

$$T_1: \quad f(g(x, y)) = [\text{if } V = K(x) \text{ then } f(x) \text{ else } f(y)]$$

is a defining formula for f but not for g or for h. Hence, for this formula

$$D_f[\mathcal{F}_1] = \{f(g(x, y))\},$$

and

$$D_r[T_1] = \{\}.$$

Note that the equality on the right-hand side of the defining formula is used in an unrestricted and undirectional form.

• The two (equivalent) formulas

$$T_2: \quad \neg p(x) \leftarrow q(x),$$

and

$$\mathcal{F}_3 : \qquad \neg q(x) \leftarrow p(x)$$

are both defining formulas but

$$D_r[\mathcal{F}_2] = \{p(x)\}, \text{ while } D_r[\mathcal{F}_3] = \{q(x)\}$$

- The formula

$$p(x) \vee q(x) \equiv r(x)$$

is not a defining formula because the left-hand side of the equivalence is not an atomic formula or its negation. ∎

The important restriction on the syntax of the *defining formulas* as formally defined above is that the literals used to define predicates and the equations used to define functions must occur in a *positive* way in the defining formulas. This restriction as well as the restriction on the use of equality and equivalence in the defining formulas are necessary to make the procedural interpretation clear. The directionality of $=$ and $\equiv$ when used as defining constructs is important to make sure that the programmer will use them only to define rewriting of simple formulas or simple terms. While it is easy to allow a broader class of formulas, doing so might cause assertions to be used in the computation in a way which differs from the programmer's intentions or not to be used at all.

The restrictions on the syntax are intended to remind the user that we are dealing with a programming language rather than with a general theorem prover.

The directionality of the implication in the defining formulas as constructed above depends on the logical direction of the implication rather than the order used to write it down; an alternative plausible approach is to make the orientation be determined by the left-to-right order in the same way it is done for equality and equivalence. This alternative approach will somewhat increase the expressive power of the language but will force the programmer to use the PROLOG style of writing conditionals, while the orientation as chosen above gives the flexibility of using either form.

4.1.2 Programs

Now that the formulas allowed in TABLOG are defined, we can describe programs and how to call them.

A program is a list of *defining formulas* specifying the algorithm. Variables appearing in these formulas are implicitly universally quantified.

A call to a program is a *goal* to be proved. Goals are formulas in logic as defined in the previous section. The variables appearing here are implicitly existentially quantified.

Here is a very simple program for appending two lists:

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

The $\circ$ symbol denotes the list insertion operator (**cons** in LISP), [] denotes the empty list (**nil** in LISP), and **append** is a function symbol whose semantics is defined by this program.

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

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

The output of the execution of this program call will be

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

as expected.

4.1.3 Notation

The example in the previous subsection follows the conventions used in all the examples in this text. Generally I will use the following notation:

- Variables are typeset in italics (x, y, w_2). Usually letters from the end of the alphabet are used but longer names will be encountered as well.
- Function and predicate symbols are typeset in bold face (**append**, **empty**). Shorter, graphic notations are also used when customary or convenient ($\leq, \circ, \in$).
- Constants are typeset in roman face (a, Bob). When representing a proper name they are capitalized.

4.2 Logical interpretation

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

For example, consider the append program

append([],u) = *u*.
append(# ou,t)) = a;o append(u, *v).*

and the call

s = append([l,2],[3]).

We have the initial tableau

assertions	goals
append([],u) = *u*	
append(#ou, *v)* = a;oappend(u,v)	
	* = append([l,2],[3])

which represents the logic sentence

r(V.)a$_{pp}$end([ ], u) = « *Af* *] - > (3z)(z =* **append([l, 2], [3])).**

It is important to note that the variables (being bound in the sentence) are dummy and can be arbitrarily renamed; in particular the variable *u* in the first conjunct is considered to be different from the one in the second conjunct.

As far as the logical interpretation is concerned, it is clear from this sentence that the order of the assertions (or conjuncts in the sentence) is not important and each equation gives the same information about both of its sides.

If the sentence describing the logical interpretation is valid (as in the example above), we can hope that the proof system will be able to prove it. The procedural interpretation of programs described in Chapter 7 is based on the proof procedure used to prove this theorem.

CHAPTER 5

EXAMPLES

5.1 Introduction

The examples in this chapter demonstrate the basic features of TABLOG. The correctness of most of these programs does not depend on the order of their assertions except for the last example, unification. In general when we write programs that do take advantage of the left-to-right, outside-in order of the interpreter's goal evaluation, we can get more efficient programs that avoid useless backtracking. An intuitive understanding of the order suffices for following the examples. The next chapter will give more details for a deeper understanding.

In the examples, I 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.

Additional examples can be found in the next chapter as part of the comparison of TABLOG to LISP and PROLOG. All the examples were actually tested on the existing TABLOG interpreter.

5.2 Simple examples

5.2.1 Deleting a list element

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

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

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 the 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.

5.2.2 Set union

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

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

Assertions 1 and 2 define the **union** function. Assertion 1 defines the union of the empty set with another set, and assertion 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.

Assertions 3 and 4 define the **member** relation. Assertion 3 specifies that no element is a member of the empty set, and assertion 4 defines how to test membership in a nonempty set recursively. Both $\mathbf{member}(x, u)$ and $\neg\mathbf{member}(x, u)$ can be proved using this program segment.

5.2.3 Factorial

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

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

Alternatively, using the postfix operator !, we can define factorial in a nicer form:

$0! = 1.$
$x! = x * (x-1)! \leftarrow x \geq 1.$

The following program, while mathematically defining yet the same function, cannot be used to evaluate factorial in TABLOG.

$0! = 1.$
$(x+1)! = (x+1) * x! \leftarrow x \geq 0.$

The problem is that TABLOG's unification procedure will not be able to match, for example, 5 with $x+1$, binding x to 4.

5.3 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. $\mathbf{partition}(x, y \circ u, u_1, y \circ u_2)$
   $\leftarrow y > x \ \wedge \ \mathbf{partition}(x, u, u_1, u_2).$

Assertions 1 and 2 form the sorting subprogram. Line 1 asserts that the empty list is already sorted. Assertion 2 specifies that, to sort a list $x \circ u$, with head x and tail u, one could append the sorted versions 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 has been defined earlier.

This example is discussed further in Chapter 7 on TABLOG's procedural interpretation.

Note that since the *linear list* is the basic data structure of TABLOG (like LISP and PROLOG) this quicksort program is not as fast as it should be; the composition of the solutions for the two sublist into a solution for the whole list takes linear time (as required by the append functions). The same composition takes constant time if the data to be sorted is stored in an array.

The following version of the quicksort program is more efficient although harder to understand. It is adopted from similar PROLOG programs that appear in [Coelho, Cotta, Pereira 80] (and attributed to M.H. van Emden) and in [Shapiro 83]. The idea of the PROLOG program is to represent lists using *difference lists* that can be concatenated in constant time. Here I take advantage of the availability of functions in TABLOG to write the procedure in a more readable form:

1. $\mathbf{quicksort}(u) = \mathbf{qsort}(u, [\,])$.
2. $\mathbf{qsort}(x \circ u, r) = \mathbf{qsort}(u_1, x \circ \mathbf{qsort}(u_2, r))$
   $\leftarrow \mathbf{partition}(x, u, u_1, u_2)$.
3. $\mathbf{qsort}([\,], r) = r$.

The **partition** subprogram is the same as in the previous version.

In the functional form of the program we use the second argument as a tool to push the rest of the computation on the recursion stack. The computation is done by breaking the first argument until we get a singleton list and then inserting its head on the result of the (yet to be computed) second argument. The auxiliary function **qsort** satisfies

$$\mathbf{qsort}(x, y) = \mathbf{append}(\mathbf{quicksort}(x), y).$$

Although the program above was written as a functional version of a PROLOG program that uses difference lists, it can actually be viewed as a direct application of McCarthy's idea that is utilized for the **flatten** function (cf. [McCarthy and Talcott 80, Chapter 3]). "

5.4 Computing with infinite streams

By using *lazy evaluation,* TABLOG can deal with infinite sequences as long as only a finite prefix is actually needed for the answer. The following program for generating the sequence of the first *n* prime numbers demonstrates this feature.

Example 5.1• *Prime numbers*

1. primes(n) = truncate(n, sift(integers(2))).

2. **truncate(0,** u) = [].

3. *n* > 0 —> truncate(n,iou) == iotruncate(n—l,u).

4. integers(i) = iointegers(i + 1).

5. sift(zou) = iosift(filter(i,u)).

6. **filter(p,** now) = if *p\n* then **filter(p,** *u)*
else *n* o **filter**(p, u).

7. *p\n* = (n = *p* * (n/p)).

The functions **integers, sift,** and **filter** manipulate infinite streams of numbers. A finite segment of the list of primes is created by truncating an infinite list. When using lazy evaluation the infinite streams are generated incrementally as the elements are consumed. The **truncate** function, defined in assertions 2 and 3 takes two arguments, a nonnegative number n and a list v, and returns the n first elements of *v.* A call to this function can be immediately evaluated if its second argument is of the form #ow, or if the first argument is zero; only if this is not the case will an attempt to further evaluate the arguments occur. In the context of the prime generation program above, this will cause the evaluation of the second argument of truncate until the next element in the infinite list of primes is generated.

5.5 Alpine Club puzzle

The following problem is taken from [Manna 74, page 160]:

> **Tony, Mike and John belong to the Alpine Club. Every member of the Alpine Club is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Mike dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow.**
> **Is there a member of the Alpine Club who is a mountain climber but not a skier?**

Recently this puzzle appeared on the PROLOG digest ([Restivo 85]) and several solutions in PROLOG where proposed; all of them require tricks. One of these solutions will be presented in the next chapter as part of the comparison of TABLOG and PROLOG.

The syntax of TABLOG makes solving this puzzle much more straightforward:

1. **alpinist(John) A alpinist(Tony) A alpinist(Mike).**
2. **skier(u) V climber(w) <— alpinist(u).**
3. **-iclimber(ti) «— likes(w,rain).**
4. **-iskier(ti) <— -»likes(u,snow).**
5. **likes(Tony, rain) A likes(Tony, snow).**
6. **likes(Mike, *x)* ⩵ -»likes(Tony, *x).***

To get the answer to the puzzle we should give TABLOG the goal

alpinist(z) A climber(^) A -»skier(2).

The solution produced by the interpreter is *z* = Mike.

Note that in line 4 of the program, the procedural interpretation of TABLOG forced us to use the contrapositive

¬skier(*u*) <—«likes(w,snow)

which is a defining formula for skier, rather than the direct form

skier(u) —• likes(u, snow)

which would be regarded as a definition for likes. This is the only transformation applied to the original specification. Had we chosen the alternative approach to

the use of implication in defining formulas (as discussed in Section 4.1), this transformation would have not been required and the direct form could be used in the program.

The disjunction in the consequent of line 2 of the program

$$\mathbf{skier}(u) \vee \mathbf{climber}(u) \leftarrow \mathbf{alpinist}(u)$$

is used to reduce

$$\mathbf{climber}(z)$$

to

$$\mathbf{alpinist}(z) \wedge \neg\mathbf{skier}(z).$$

Note that this is the only positive fact about **skier** and **climber** in the original puzzle. O'Keefe coded it as:

$$\mathbf{climber}(x) \leftarrow \mathbf{alpinist}(x) \ \wedge \ \mathbf{nonskier}(x)$$

which is correct for the given query but will not work for the query

$$\mathbf{skier}(z)$$

for which TABLOG will generate the answer $z =$ Tony; Actually the predicate **skier** does not even appear in O'Keefe's program.

Since TABLOG does not use negation as failure there is no way to deduce anything about John (except that he is an alpinist) while O'Keefe's solution can deduce that John is not a climber, which is not a consequence of the statement of the puzzle.

5.6 Map coloring

Map coloring is a famous problem that fits nicely as an example of declarative programming. The goal is to color a map such that each region's color is distinct from those of all its neighbors. The following program is for the case of a five-region map with the regions represented by the variables A, B, C, D, and E, and each should be assigned one of the colors red, blue, green, or black.

1. $\mathbf{colormap}([A, B, C, D, E]) \leftarrow$
$\mathbf{next}(A, B) \wedge \mathbf{next}(C, D) \wedge \mathbf{next}(A, C) \wedge$
$\mathbf{next}(A, D) \wedge \mathbf{next}(B, C) \wedge \mathbf{next}(B, E) \wedge$
$\mathbf{next}(C, E) \wedge \mathbf{next}(D, E).$

This assertion describes the topology of the map. The **next** predicate specifies that its two arguments should be legally colored as adjacent regions.

2. $\mathbf{next}(x, y) \leftarrow \mathbf{color}(x) \wedge \mathbf{color}(y) \wedge x \neq y.$

This is the definition of legal coloring of two neighbors, each should be assigned a color and the colors should be different.

3. $\mathbf{color}(x) \equiv [(x = \text{red}) \vee (x = \text{blue}) \vee (x = \text{green}) \vee (x = \text{black})].$

This last assertion lists the available colors.

There is no need to assert that the colors are distinct; this fact is a consequence of the way TABLOG treats distinct constants.

This program should be called by the goal

$$\mathbf{colormap}(z).$$

and the result will be binding z to list of colors assigned to the regions designated by the logical variables A, B, C, D, and E. In searching for the solution of this problem the TABLOG interpreter will depend heavily on backtracking; alternative solutions are attempted until a legal assignment of colors is found. The use of depth-first search for the solutions and the standard PROLOG backtracking will also characterize the execution of this TABLOG program. [McCarthy 82] gives this problem as an example of the inadequacy of the PROLOG's implicit control and suggests how to solve the map coloring problem in a more efficient way.

5,7 Unification

The TABLOG program below is an implementation of a unification algorithm. The algorithm coded here is very similar to the one derived in [Manna and Waldinger 81] and it can be applied generally to unify two expressions or two lists of expressions.

The implementation below expects both expressions and lists of expressions to be encoded using the standard list and insertion constructs, [• • •] and o. For an expression the first element of the list is the *operator* (predicate or function symbol) of the expression.

Substitutions are represented as lists of pairs; the empty (identity) substitution is represented by the empty list. When two expressions or lists of expressions are given as TABLOG lists, the program will return a most general unifier of the inputs. If the two expressions are not unifiable, or if the elements of two lists are not pairwise unifiable, the program will return the special value "nonunify".

1. unify(U) = [J.

The empty substitution unifies a list with itself

2. / ^ [] —> (unify([],/) = nonunify A unify(/, []) = nonunify).

The empty list does not unify with a nonempty list

3. varp(ar) —> [unify(#, *I)* = univar(#,Z) A unify(/,a;) = univar(x,/)].

If one of the argument is a variable univar is used

4. unify(xo/,#om) = unify(/,m).

If the two lists have the same head we have to unify the tails

5. *u* = unify*(x,y)* —» unify(#o/,t/om) = uniappend(w,/,m).

First unify the heads and then the tails; uniappend combines the two unifiers

6. unify(x,y) = nonunify.

If none of the cases above applies the two arguments cannot be unified.

7. univar(x,y) = *if* occurin(x, *y) then* nonunify *else* [[a?,y]].

8. uniappend(nonunify,/,m) = nonunify.

9. uniappend(u,/,ro) = compose(u,unify(/i,mi)) <—
li = subst(/, *u)* A *m* = subst(m,u).

10. compose(u, nonunify) = nonunify.

11. compose([],v) = *v.*

12. compose([s,y]oti,t;) = [a;,subst(y, u)]ocompose(u, *v).*

13. subst(/,nonunify) = nonunify.
14. subst(Z, []) = /.
15. subst([],u) = [].
16. subst(a;o/,u) = subst(a:,u)osubst(7,t/).
17. functorp(x) V constp(:r) —> subst(a;,ii) = *x*.
18. varp(#) —> subst(#, *[x,y]ou)* = y.
19. varp(x) —> subst(x,vou) = subst(x,w).
20. varp(x) —• [occurin(a;,y) ≡ (a: = y)].
21. -ioccurin(a;,[]).
22. constp(y) V functorp(y) —> -ioccurin(a:,y).
23. occurin(#,y) V occurin(a:,/) —> occurin(a:,yo7).

We should also assert what are the variable, constant, and function symbols. For example,

24. varp(vi) A varp(r2) A constp(a) A constp(6) A functorp(/).
25. varp(wi) A varp(t/2) A constp(c) A constp(cf) A functorp(flr).

To unify the two expressions *f(g{v*, V2), *u\)* and *f(u2,c)* the program should be supplied with the query goal

$$z = \text{unify}([/,[sf,vi,V2],Mi],[/,U2,c]),$$

TABLOG will respond with [[U2, [flS^i,^]], [t/i,c]] as the final value of *z* this answer represents a most general unifier {y2 <— ⌐jr(a?i,aj2)> yi <— c} of the two input expressions.

Note that the program above takes advantage of the known order of evaluation of the TABLOG interpreter. Except for a failing occur-check, the result "nonunify" is generated only after failure of the previous assertions. In principle it is possible to write the same program as an order independent one but it would be much more complex. Usually we can do this by using the conditional *if-then-else* which causes the computation to branch depending on the evaluation of the condition. The problem that arises here and in other large examples is that many predicates are not *complete,* i.e., we cannot always prove the predicate or its negation. In the program above, for example, the type predicates functorp, varp and constp are all incomplete. This means that we cannot use these predicates as the test of a conditional.

In assertion 9 we force applying the substitution u to l and m to get l_1 and m_1 before the unifier of l_1 and m_1 is found and the composition is attempted. This explicit version is not necessary; alternatively we could write

9′. $\mathbf{uniappend}(u, l, m) = \mathbf{compose}(u, \mathbf{unify}(\mathbf{subst}(l, u), \mathbf{subst}(m, u)))$.

Even in this form the application of the substitution u to l and m will be invoked by the call-by-need evaluation method before composition can succeed. In the program I have included the explicit version to demonstrate this point; this version is also slightly more efficient.

CHAPTER 6

COMPARISON WITH LISP AND PROLOG

6.1 Introduction

In this chapter we will use examples to compare the expressive power of TABLOG with LISP and PROLOG. The comparison in both cases is with the pure version of these languages and not with the various dialects with fancy control constructs. For the case of PROLOG in particular we will not consider here the *cut* operator and its problems.

Because TABLOG and PROLOG are both based on predicate logic the comparison between these two is longer and more detailed than the comparison with LISP.

First we will see the convenience of having functions and equality in TABLOG. Later the rich set of connectives will be contrasted with PROLOG's Horn-clause language. This comparison will be followed by a discussion of the merits and pitfalls of using PROLOG's negation-as-failure vs. TABLOG's real negation. Before comparing TABLOG to LISP, we will also discuss the differences between the unification procedure used by standard PROLOG interpreters and the one used by the TABLOG interpreter.

The comparison with LISP points out the advantages of unification-based languages over LISP rather than the unique features of TABLOG. We will see the convenience of using predicates to define multiple-output programs and to get "free" input decomposition. The final examples will show how problems that require two-pass programs in LISP can be solved by using unification in one-pass TABLOG programs.

6.2 Functions and Equality

While PROLOG programs must be relations, TABLOG programs can be either relations or functions. The availability of equality and functions 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

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

The corresponding clause in the PROLOG program would be something like

$$\begin{aligned}\mathbf{qsortp}(x \circ u, \mathit{Output}) \leftarrow\ &\mathbf{partition}(x, u, u_1, u_2) \wedge \\ &\mathbf{qsortp}(u_1, \mathit{Temp}_1) \wedge \\ &\mathbf{qsortp}(u_2, \mathit{Temp}_2) \wedge \\ &\mathbf{appendp}(\mathit{Temp}_1, x \circ \mathit{Temp}_2, \mathit{Output}).\end{aligned}$$

The additional variables Temp_1 and Temp_2 are required to store the results of sorting u_1 and u_2, respectively; *Output* will hold the sorted output.

Another example is the factorial function, written in TABLOG as:

$0! = 1.$

$x > 0 \rightarrow\ x! = x * (x-1)!.$

In PROLOG it will be something like:

$\mathbf{factp}(0, 1).$

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

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

These examples demonstrate 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.

Recently there have been attempts to add equality to PROLOG; [Kornfeld 83] proposes the addition of a predicate equal(s, *t)* to specify that the two terms *s* and *t* should be unifiable. This has the effect of unifying classes of objects denoted by *s* and *t* but does not allow replacing *s* with *t.* [Tamaki 84] introduces a reducibility predicate to specify the reducibility of one term to another. This predicate has semantics similar to that of equality in TABLOG when it is used by the equality rule. In that framework, though, there are limitations on the possible nesting of terms, and programs are restricted to Horn clause form. These approaches and others are discussed in Chapter 11.

6.3 Negation and Equivalence

One of the problems of PROLOG (actually Horn clauses) is the inability to express negation. To overcome this limitation, PROLOG systems support negation-as-failure: to prove a negated literal no£(p(<)), a PROLOG interpreter attempts to prove p(<); if this proof terminates, then *not(p(t))* will succeed if and only if the proof of p(<) fails.

As was already pointed out, TABLOG includes real negation as a regular connective; negated facts can be asserted or proved directly. When proving goals in TABLOG, the negation is treated like any other connective of logic.
The TABLOG union program, described earlier, uses both equivalence and negation:

union([],#) = *v.*

union(x ou, *v)* = if member(ar, *v)*
then union(u,v)
else (xounion(w,v)).

-»member(a;,[]).

member(a:, *y* o *u)* ≡ *(x = y)* V member(a:, *u).*

Here is a possible PROLOG implementation of the same algorithm:

unionp(*x* o *u, v)Z)* <— memberp(a;,u) A unionp(u,*v,z).*

unionp(x o u, v, *x o z)* <— unionp(u, v, *z).*

unionp([],v,r;).

memberp(x, *x o u).*

memberp(a:,you) <— 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.)

A consequence of having real, explicit, negation is that some predicates may be incomplete; for some cases we can prove neither the formula nor its negation. In PROLOG and LISP this is never the case unless the computation diverges when trying to evaluate the formula; whenever the evaluation of a formula $\mathbf{p}(t)$ terminates we know if it is *true* or *false*.

While using negation-as-failure is convenient in some cases, it is very order dependent and is not suitable for parallel execution; for this reason it is not available in CONCURRENT PROLOG ([Shapiro 83]), for example.

In some cases, however, the notion of negation-as-failure is attractive; for example when we have a database with few positive facts and we want to use the *closed world assumption*, i.e., assuming that everything which is not provable from the program assertions is false. This saves the user of the system the burden of stating many negative facts.

When dealing with a sequential order-dependent interpreter it is easy to add negation-as-failure to TABLOG in addition to real negation. We can even do it selectively only for some chosen predicates. We cannot do it within the language but it is a simple addition to the interpreter to instruct it to view failure as negation for predicates declared to be subject to negation-as-failure.

The incompleteness of the proof procedure employed by TABLOG (cf. Section 9.2) might cause problems if we try to interpret failure as negation for some formula whose truth is implied by the program but is not provable by the TABLOG interpreter. The failure to prove the formula will result in the assumption that its negation is true.

The negation-as-failure in PROLOG can result in wrong answers when applied to unground terms as demonstrated in the following example

Example 6.1.
Given the program

$$\mathbf{bachelor}(x) \leftarrow \mathit{not}(\mathbf{married}(x)) \ \wedge \ \mathbf{male}(x).$$
$$\mathbf{married}(\text{Alice}).$$
$$\mathbf{male}(\text{Bob}).$$

The goal

$$\mathbf{bachelor}(z).$$

fails, although (under the closed world assumption) the fact that Bob is a bachelor is implied by this program. The problem is that the subgoal **married^)** succeeds with *z* bound to Alice; this implies the failure of the first conjunct 7ioi(married(2)) and therefore the whole goal. If we change the order of the conjuncts in the first assertion to be

bachelor(#) <— male(#) A *not(married(x)).*

PROLOG will answer the same goal positively, with *z* = Bob.

Note that in TABLOG, because we do not assume a closed world, we must specify explicitly

-imarried(Bob).

to be able* to prove bachelor(Bob). ▟

Although TABLOG does not interpret failure as negation, there is some relation between the two, or actually between *false* and failure. For example, while a conjunction will fail if one of the conjuncts fails, a disjunction will fail only if all the disjuncts fail. So failure behaves like *false* in this case. This is particularly visible in the (sequential) TABLOG interpreter where the order of the conjuncts and the assertions is important. A failure of a condition in an assertion will cause the next assertion to be used, which is exactly the behavior of a conditional when a condition evaluates to *false.* As we have seen in the unification example (Section 5.7), this use of failure as part of the control is sometimes very convenient.

Example 6.2.
Given the following assertions specifying a rule for evaluating the **parent** predicate and a fact about the **mother** predicate:

parent(#,Ţ/)<—father(x,y)Vmother(a:,y).
mother(Ann, Bob).

We can evaluate the query

parent(2,Bob).

During this evaluation, the proof of father(ar, Bob) fails and the interpreter solves the goal by satisfying the other disjunct, demonstrating that failure is interpreted as *false* when trying to satisfy the disjunction. ▟

Note that to evaluate the false branch of a conditional and to answer queries that include explicit or implicit negation, we need real negation and the ability to reduce a formula to *false.*

6.3.1 Alpine Club puzzle

The Alpine Club puzzle was presented as an example in Chapter 5; here I will describe an attempt to solve it in PROLOG. This example shows that some problems are very hard to encode (and solve) when we restrict ourselves to the language of Horn clauses.

> Tony, Mike and John belong to the Alpine Club. Every member of the Alpine Club" is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Mike dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow.

Is there a member of the Alpine Club who is a mountain climber but not a skier?

One of the solutions in PROLOG was offered by [O'Keefe 85]:

alpinist(Tony).
alpinist (Mike),
alpinist (John).
likes(Tony, rain, yes).
likes(Tony, snow, yes).
likes(Mike, a:, yes) «— likes(Tony, a:, no).
likes(Mike, x, no) «— likes(Tony, x, yes).
likes(a:, rain, no) *— climber(a?).
nonskier(#) •— likes(#, snow, no).
climber(x) <— alpinist(x) A nonskier(x).

To solve the puzzle in this form the query

alpinist(x) A climber(x) A nonskier(a:).

should be given to the PROLOG system.

To make sure that the solution is consistent with the original statement of the puzzle, we also have to independently show the unprovability of the query

likes(#, y, yes) A likes(:r, ?/, no).

In principle we should also assert in the program

likes(#, y, yes) V likes(x, y, no).

However, since the predicate **likes** does not appear in the program or the goal with an uninstantiated third argument, this assertion can be omitted.

Other solutions in PROLOG proposed in the discussion were even less satisfactory as they did not encode the puzzle accurately.

For comparison let us look again at the TABLOG solution:

1. $\mathbf{alpinist}(\text{John}) \wedge \mathbf{alpinist}(\text{Tony}) \wedge \mathbf{alpinist}(\text{Mike})$.
2. $\mathbf{skier}(u) \vee \mathbf{climber}(u) \leftarrow \mathbf{alpinist}(u)$.
3. $\neg\mathbf{climber}(u) \leftarrow \mathbf{likes}(u, \text{rain})$.
4. $\neg\mathbf{skier}(u) \leftarrow \neg\mathbf{likes}(u, \text{snow})$.
5. $\mathbf{likes}(\text{Tony}, \text{rain}) \wedge \mathbf{likes}(\text{Tony}, \text{snow})$.
6. $\mathbf{likes}(\text{Mike}, x) \equiv \neg\mathbf{likes}(\text{Tony}, x)$.

The puzzle is solved by proving the goal

$$\mathbf{alpinist}(z) \wedge \mathbf{climber}(z) \wedge \neg\mathbf{skier}(z).$$

The solution produced by the interpreter is $z = \text{Mike}$. The computation that leads to the solution is discussed in Section 5.5.

6.4 Unification

The unification procedure customarily 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 (and might take infinite amount of time to print). For example, look at the following program specifying the **parent** relation:

$$\mathbf{parent}(\mathbf{father}(x), x).$$

If this program is called with the goal

$$\mathbf{parent}(z, z)$$

a PROLOG interpreter will succeed but with the binding

$$\{z \leftarrow \mathbf{father}(z)\}$$

i.e.,

$$\{z \leftarrow \mathbf{father}(\mathbf{father}(\mathbf{father}(\cdots)\cdots))\}$$

which is cyclic and cannot be printed unless a special notation for such cases is introduced. This answer is also wrong because logically the program does not imply the truth of the goal. The fact that everyone's father is his or her parent does not imply that someone is his or her own parent. This example is essentially the skolemized version of proving the nontheorem

$$(\forall x \exists y)\mathbf{parent}(y, x) \rightarrow (\exists z)\mathbf{parent}(z, z).$$

The unification used by the TABLOG interpreter does include an occur-check, so that only theorems can indeed be proved. This choice is orthogonal to the other design decisions in the implementation of both TABLOG and PROLOG, so if future implementors think that the cost of this test is too high, they will be able to use unification without the occur-check and pay by losing soundness for some cases. TABLOG allows using nested function calls, and hence programs tend to have fewer repetition of variables than the corresponding PROLOG programs. Since the occur-check is necessary only if there is at least one variable that occurs more than once in one of the unified expressions (assuming renaming of variables to preserve their locality) this observation can lead to a more efficient unification with restricted application of the occur-check.

In the QUTE language [Sato and Sakurai 85], the omission of the occur-check is essential to the way recursive definitions are introduced. Since QUTE is not based on resolution theorem proving, this does not compromise its soundness.

6.5 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.

The main advantage of TABLOG over LISP is in the power of unification as a binding mechanism for inputs and outputs. This allows decomposition of input structures on the fly and an easy way to support multiple outputs. The first example will demonstrate these two points.

6.5.1 Partition for quicksort

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 clearer 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. Even if we use a dialect of LISP that supports multi-valued functions we still get a program which is clearly more complex than the TABLOG program above.

$\mathbf{filter}(u, x) \Leftarrow$
  if $\mathbf{null}(u)$ then $\mathbf{values}(\mathbf{nil}, \mathbf{nil})$
    else $(\lambda v_1 v_2$.if $x \leq \mathbf{car}(u)$
      then $\mathbf{values}(\mathbf{car}(u) \circ v_1, v_2)$
      else $\mathbf{values}(v_1, \mathbf{car}(u) \circ v_2))$
    $(\mathbf{filter}(\mathbf{cdr}(u), x))$.

The **values** function is a special mechanism for returning multiple values from a LISP function. The λ construct must be used to retrieve multiple values; alternatively we can define an auxiliary function to handle it.

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**.

Whereas the effect of getting the partition to generate the two lists simultaneously can be achieved by modern LISP dialects, the power of unification to compute with unevaluated variables is harder to simulate in LISP, as will be demonstrated next.

6.5.2 Computing with future values

In this section I will present two examples that show that the power of unification and logical variables goes beyond the easy handling of input destructuring and producing multiple outputs. Both of the examples show how we can build the answer of a computation using unbound logic variables that will get concrete values in the future of the computation. In these examples I use a functional style to emphasize that the feature demonstrated here is the result of having logical variables rather than of the relational style of PROLOG-like languages.

Example 6.3. *Tree transformation*
The tree transformation problem is the following:

> Transform a given tree s to get the output tree t with the same structure (shape) as s. All the tips (leaves) of t however should have the same value, namely $\mathbf{min}(t_1, t_2, \ldots, t_n)$ where $t_1, \ldots, t_n$ are the tips of the original tree s.

The solution given below can be generalized to replace the tips with any associative function of all the tips instead of the minimum value. We will assume that trees are represented using the o operator and that all the leaves are numbers.

A possible solution in LISP is:

$$\mathbf{transform}(s) \Leftarrow \mathbf{sameshape}(s, \mathbf{minleaf}(s))$$

$$\mathbf{minleaf}(s) \Leftarrow$$
$$\quad \text{if } \mathbf{atom}(s) \text{ then } s$$
$$\quad \text{else } \mathbf{min}(\mathbf{minleaf}(\mathbf{car}(s)), \mathbf{minleaf}(\mathbf{cdr}(s)))$$

$$\mathbf{sameshape}(s, m) \Leftarrow$$
$$\quad \text{if } \mathbf{atom}(s) \text{ then } m$$
$$\quad \text{else } \mathbf{sameshape}(\mathbf{car}(s), m) \,\mathbf{o}\, \mathbf{sameshape}(\mathbf{cdr}(s), m).$$

This program has to traverse the input tree s twice, once to find the minimum value that can be found in a leaf and then once again to copy the shape of the tree.

The following TABLOG program solves this problem using an one-pass algorithm. **min** is a built-in (basic) function while **transform** and **trans3** are the functions being defined by the program. The built-in predicate **atomp** is assumed to be true for all the tips in the original tree.

1. $\mathbf{transform}(t) = \mathbf{trans3}(t, m, m)$.
2. $\mathbf{trans3}(l \circ r, x, \mathbf{min}(m_1, m_2)) = \mathbf{trans3}(l, x, m_1) \circ \mathbf{trans3}(r, x, m_2)$.
3. $\mathbf{trans3}(n, m, n) = m \leftarrow \mathbf{atomp}(n)$.

A sample call to this program will be

$$z = \mathbf{transform}((3 \circ 2) \circ ((1 \circ 0) \circ (1 \circ 5))).$$

The call to $\mathbf{transform}(t)$ is translated into a call to **trans3** with the last two arguments bound to the same variable. For the sample call above this will become

$$z = \mathbf{trans3}((3 \circ 2) \circ ((1 \circ 0) \circ (1 \circ 5)), m, m).$$

This coupling of arguments makes the variable x in the recursive calls (on **trans3**) get bound to the value of the third argument of these calls, which gets the minimum value in the tree. One of the recursive calls in the on-going example will then have to evaluate

$$\mathbf{trans3}((1 \circ 0) \circ (1 \circ 5), \mathbf{min}(2, m_7), m_7)$$

in which the second argument $\mathbf{min}(m_7, 2)$ was inherited from the outside call. In this expression 2 is the minimum of the left subtree of the input tree as was evaluated after the recursive calls for this part were completed. The other argument m_7 represent the minimum value of the right subtree. The minimum value is computed bottom-up: execution of assertion 3 on each tip binds the third argument of the corresponding recursive call to the tip's numerical value and this value propagates up as a candidate for the minimum value. For example the value 2 was propagated up as the result of evaluating

$$\mathbf{transform3}(2, \mathbf{min}(m_{12}, m_7, 3), M_{12}).$$

This evaluation returned (the unbound) $\mathbf{min}(2, m_7)$ after binding m_{12} to 2 and eliminating 3 as a candidate for the minimum. Each execution of assertion 2 (on a non-tip tree) assigns to the third argument the minimum of the two values returned

by the recursive calls for computing the function over the left and right subtrees. When the recursion returns all the way to the first call, the minimum value of the whole tree is returned as the value of the third argument. Because the "filling element" *x* is bound to the same variable, all the tips of the result tree get bound to this minimal value.

Example *6A. Address translation*

This example from [Reddy 85] demonstrates once again the power of unification and is less artificial than the previous one. The input to the program is a list of instructions of two kinds: def(a) and use(a), where a is a constant. The program outputs a stream of instruction with def(a) translated into assign(a,n) where n is an integer address; use(a) is translated into get(n) where *n* is the address assigned to *a.* The get(n) should use the correct n value even if the address for the atom *a* in use(a) is assigned by an input def(a) that comes later in the stream. The power of the logical variable, which can be used in building the output stream before it actually gets bound, lets this program solve the problem in one pass.

In the following program *inlist, table, address, u, v, x,* and *n* are all variables.

1. translate(ṁfei) = *ma.p(inlist, table,* 1).
2. map([],ti,n) = [].
3. map(def*(x) o inlist, table, n)* = assign(:r, *n)* o **map(** infoi, *table, n* + 1)
   «— member(assign(#, n),tfa&Ze).
4. *map(use(x)o inlist, table, n) = get(address)oma.p(inlist, table, n)*
   •— member(assign(x, *address), table).*

A call to this program will be a goal like

$$z = \text{translate}([\text{use}(a), \text{use}(6), \text{def}(c), \text{def}(6), \text{use}(6), \text{def}(a), \text{use}(c)]).$$

The output of the program is built by repeated calls to the map function until the input stream is exhausted. At each step a new term is added to the output list using the address of the argument of the term in the input list. The table represented by the list *table* holds the assignment of addresses to atoms. The beauty of this solution is that the relative order of the corresponding use and get for the same variable is not important.

The answer for the sample goal given above will be:

$$z = [\text{get}(3), \text{get}(2), \text{assign}(c, 1), \text{assign}(6,2), \text{get}(2), \text{assign}(a, 3), \text{get}(l)].$$

6.6 Summary

This chapter demonstrates the advantages of TABLOG's expressive power over PROLOG and LISP.

The availability of a rich set of connectives and the functional notation makes TABLOG more problem oriented than PROLOG, especially when dealing with problems related to logical reasoning. Real unification and negation make the TABLOG interpreter sound, which is not always the case for PROLOG.

When compared to LISP, TABLOG benefits from the power of unification to handle unbound variables and to put values in their destination before they are actually computed. The unification and the relational notation also make it easier to write multiple-output programs and to decompose data structures on input.

One point in favor of both LISP and PROLOG is the fact that all predicates are complete, i.e., they always evaluate to *true* or *false* (or their appropriate representation) while in TABLOG the evaluation of a predicate may fail. Sometimes this is an advantage both in writing programs and in formally defining their semantics. In PROLOG, however, this is achieved using negation-as-failure, which has its own problems, as illustrated earlier.

Chapter 7

Procedural Interpretation

7.1 Introduction

The deductive-tableau proof system provides deduction rules but does not specify in which order to apply them. To use this proof system as a programming language, we must devise a *proof procedure* that employs the rules in a predictable and efficient manner. This proof procedure, which is used to prove the goal from the program assertions, defines the procedural interpretation of TABLOG programs. In this chapter I will describe the proof procedure and will detail the exact order used to evaluate predicates and functions. The proof procedure uses the inference rules of the deductive tableau that were described in Chapter 3.

Initially the *current goal* is the user query goal with all its variables appearing in the output column. The proof procedure reduces the goal by applying inference rules between the current goal and one of the assertions, creating a new current goal. This is done repeatedly until the goal *true* is derived or until no further deduction steps are possible. During this process, the output variables are bound to values of the computed functions or to values satisfying the computed predicates. The variables are bound when subexpressions of the current goal are unified with subexpressions of the assertions. The output of the program is the final binding of the variables of the original goal, which can be found in the output column of the success goal *true*.

The process I have just described is 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 (as kept in the output

to the identity matrix; in TABLOG we are done when we have reduced the original goal to *true*.

7.1.1 Function classes

The function symbols of TABLOG are grouped according to their intended use:

Constructor function symbols serve to build data structures in the language; for example, o is a predefined constructor.

Basic (or *built-in*) functions have attached procedures hard-wired into the implementation to define their semantics; basic arithmetic functions like + and **min** are predefined built-in functions.

Defined functions are those that are defined by the assertions of a TABLOG program.

The constructor and basic functions are called *primitive* functions, and the basic and defined functions are called *reducible* functions. The difference between the two kinds of reducible functions is the way they are reduced: basic functions are reduced by the built-in simplifier while defined functions are reduced by the equality rule.

Although there are no constructor predicates, we do distinguish between *basic* (primitive) predicates and *defined* predicates. The *primitive operators* include the primitive functions, primitive predicates, the logical connectives, and the *if-then-else* construct (in both usages).

A *primitive expression* is an expression that does not contain defined (i.e., non-primitive) operators; a *concrete expression* is a variable-free (ground) primitive expression. For example, the term $[(2 + x + 5)]$ (i.e., $(2 + x + 5) \circ [\,]$) is a primitive (but not ground and therefore not concrete) expression (with constructor function o, and primitive built-in function +), and will be automatically simplified to $[(x + 7)]$.

7.1.2 Variables

As in PROLOG, variables are local to the assertion or goal in which they appear; there are no global variables in TABLOG. Although we may use the same variable name in different entries, renaming of variables is done automatically by the interpreter to prevent collision of names. This renaming is done before every derivation step.

As was already mentioned, the variables of the original goal are the *output variables*, whose bindings are recorded (in the output column) throughout the proof (computation) and which become the outputs of the program upon termination. Once again, the same variable names can be used for a different purpose in other assertions; renaming will prevent any problems. In principle only a designated subset of the variables of the original goal should be output variable, but for simplicity all of them are assumed to be in this subset.

7.2 Definitions and reductions

Every assertion *A* in a TABLOG program is a defining formula (see Section 3.2); we can view each term in J9/[*4] as the head of a procedure defining a function and each formula in D_r[,4] as the head of a procedure defining a relation. Such procedures are invoked via the equality rule for functions and via the equivalence rule or nonclausal resolution rule for predicates, according to the form of the defining assertion. Every function and predicate can be defined more than once; the alternative definitions can be regarded as procedures to be tried in sequence.

Equality is used to define terms; equality definitions are used according to the equality rule:

assertions	goals	outputs
A[s = t]		
	G[ŝ]	***9***
	not A0[false] ***and*** ***G9(tO)***	***90***

where *9* is a unifier of *s* and ŝ.

The TABLOG version of the rule is used in a directional way, always replacing an instance of the left-hand side *(s9)* with the corresponding instantiation of the right-hand side *(tO)*. The assertion after the replacement of the equality by *false* can be regarded as the body of the procedure defining the term and its negation is added as a conjunct in the result of the equality rule. There is a great similarity between this rule and narrowing as used by [Goguen and Meseguer 85] or by [Reddy 85b].

The definition of an atomic formula is used by applying the nonclausal resolution rule according to the polarity of the occurrence. In this case, the body of the procedure is the formula after replacing the occurrence of the formula with *false* or with *true*, according to its polarity. When the formula is defined using the equivalence connective, the equivalence rule is used to replace the formula by another one, and the body of the procedure is determined in a way similar to the equality rule.

Whenever we reduce an expression, its alternative definitions are used sequentially until one succeeds. We still, however, have to specify the procedure used

to select which expression to reduce. This order, which will be described in Section 7.4, can be utilized by the programmer, as was already demonstrated in some of the examples.

Before we specify this order, let us observe how reductions are applied to equality and standard connectives. The connectives are not reduced directly; the atomic formulas that appear as their arguments are reduced using the inference rules in the order to be specified; following each such reduction the resulting goal is simplified and at this point the connectives may get reduced according to standard propositional simplifications.

7.3 Order of evaluation

The current TABLOG interpreter follows the standard PROLOG interpreter in the way it chooses the next assertion to use: the assertion is chosen according to its position in the program. The execution is also based on backward chaining, in which we always try to reduce the current goal rather than create more facts from the assertions in the program.

Pure logic is of course totally nonprocedural and the order of writing conjuncts or disjuncts is not important. Ideally, logic programming is considered to be nonprocedural and therefore one might expect the order not to be important. However, since most tasks solved by computer programs are at least partially sequential and also most current day computers are sequential (at least at the machine instruction level and above), it makes sense to let the programmer have some control over the order of events.

7.4 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.

In contrast to the declarative (logical) semantics of the tableau, the procedural interpretation of the tableau as a program takes the order of entries into account; changing the order of two assertions or changing the order of the conjuncts or disjuncts in an assertion or a goal may lead to different computations and results.

The predefined order of evaluation (reduction) of the tableau can be used by the programmer when specifying an algorithm in TABLOG.

Selecting an expression to reduce: 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-outermost) basic expression is chosen and reduced, if possible.

Some functions and predicates (e.g., o) are predefined to be primitive; a basic expression in which such a symbol is the main operator is never selected to be reduced, although its subexpressions may be selected for reduction.

Reducing an expression: The reduction is done by applying an appropriate inference rule: the equality rule for a term, and nonclausal resolution or the equivalence rule for an atomic formula. These inference rules can be invoked using procedures as described before.

If the reduction fails, the choice of the basic expression is suspended and a subexpression of it is chosen instead. If no such subexpression exists, a form of backtracking takes place, as will be described later.

If the atomic formula is an equality and its two sides do not contain any *defined* functions, the equality is reduced by unifying the two sides and replacing the equality by *true*. If this is not the case or if the unification fails, the choice is suspended and the two sides are searched for the next basic expression. If the two sides of the equality are syntactically equal the equality will be reduced to *true* even if the expressions do contain defined functions.

If the operator (function or predicate) of the chosen expression is primitive it gets special treatment. Operators with built-in semantics (in the form of attached procedures) are evaluated when they have appropriate arguments; otherwise they are treated like failed reductions, i.e., the choice is suspended. Since constructors are not reducible, the choice of a term with a constructor function as the main operator is suspended immediately and subexpressions are reduced.

Formulas generally occur as the outermost expressions; therefore resolution and equivalence rules are in most cases tried first. Only if they cannot be applied do we reduce the terms inside the formulas; this is very similar to the way narrowing is applied in other approaches. Note however that this is not always the case; for example, we can have formulas inside the terms (as the condition of an *if-then-else* expression) and we also have the notion of suspension.

The order of evaluation described here can be called *call by need* and it is employed in a *lazy-evaluation* manner where arguments are not computed unless their values are needed. Given the left-to-right order of evaluation between (for example) conjuncts in the goal, we can force the evaluation of an argument by using an auxiliary variable (this is similar to the way [Haridi 81] removes nested function calls).

Before we demonstrate this with examples, it is important to emphasize again that matching of the selected expression against program assertions is done in the order of appearance. This dependence on order makes it possible to guide the control of execution of the program and achieve a more efficient program.

All of the order dependence of programs is part of the sequential model for TABLOG execution. A parallel model does not necessarily require programs to be order-dependent.

7.4.1 Quicksort example

We will now try to clarify this via an example:

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

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

Since the right-hand side of the equality contains the defined function symbol **qsort**, the unification of the two sides is delayed and the basic expression chosen for reduction will be the term $\mathbf{qsort}([2,1,4,3])$. 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, which is then negated; 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

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

This formula is reduced by the simplifications

$$(\mathit{false} \leftarrow P) \;\Rightarrow\; \mathit{not}\ P$$

and

$$\mathit{not}(\mathit{not}\ P) \;\Rightarrow\; P$$

to obtain the new goal

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

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)

$$\begin{aligned} &\mathbf{partition}(x, y \circ u, y \circ u_3, u_4) \\ &\qquad \leftarrow\ y \leq x\ \wedge\ \mathbf{partition}(x,u,u_3,u_4). \end{aligned}$$

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 after negating the assertion, and the new goal generated is

$$\begin{aligned} &\mathit{not}(\mathit{false} \leftarrow (1 \leq 2 \wedge \mathbf{partition}(2,[4,3],u_3,u_4))) \wedge \\ &z = \mathbf{append}(\mathbf{qsort}(1 \circ u_3),\ 2 \circ \mathbf{qsort}(u_4)). \end{aligned}$$

Which is then simplified to

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

For this current goal the basic expression chosen for reduction is once again an atomic formula

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

which is unifiable with the same second assertion. This time applying the nonclausal resolution rule between the goal and the assertion results in the trivial goal *false*, because the first conjunct of the antecedent of the assertion $y \leq x$ is false for the binding implied by the unifier, since $4 \not\leq 2$. This false conjunct causes the resolution to fail, and the next assertion defining **partition** is used. Now the nonclausal resolution rule can be applied successfully to get the goal

$$\mathbf{partition}(2, [3], u_5, u_6) \wedge$$
$$z = \mathbf{append}(\mathbf{qsort}(1 \circ u_5), 2 \circ \mathbf{qsort}(4 \circ u_6)).$$

After a sequence of resolutions to compute the partition of the input list, we get the goal

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

which leads to the selection of the whole right-hand side of the equality as the expression to reduce. None of the two assertions defining **append** can be used to reduce this term; the selection is therefore suspended and the term **qsort**([1]) is chosen instead and gets reduced successfully.

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 z gets bound to the desired output

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

7.5 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. If 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.

Example 7.1. *Family relations*
Here is a program describing a family:

1. $\mathbf{parent}(x, y) \equiv \mathbf{mother}(x, y) \ \vee \ \mathbf{father}(x, y)$.
2. $\mathbf{grandparent}(x, y) \leftarrow \mathbf{parent}(x, z) \ \wedge \ \mathbf{parent}(z, y)$.
3. $\mathbf{mother}(\text{Ann}, \text{Dave}) \ \wedge \ \mathbf{mother}(\text{Fay}, \text{Bob})$.
4. $\mathbf{father}(\text{Bob}, \text{Ed}) \ \wedge \ \mathbf{father}(\text{Bob}, \text{Carl})$.

Given this program and the goal

5. $\mathbf{grandparent}(z, \text{Carl})$ $\qquad z$

with the output column displayed at the right of the goal, we get the following trace of execution

6.	**parent**(x_1, z_1) $\wedge$ **parent**$(z_1,$ Carl)	x_1
7.	(**mother**(x_2, y_1) $\vee$ **father**(x_2, y_1)) $\wedge$ **parent**$(y_1,$ Carl)	x_2
8.	**parent**(Dave, Carl)	Ann
9.	**mother**(Dave, Carl) $\vee$ **father**(Dave, Carl)	Ann
10.	**parent**(Bob, Carl)	Fay
11.	**mother**(Bob, Carl) $\vee$ **father**(Bob, Carl)	Fay
12.	*true*	Fay

Goal 8 was deduced from goal 7 by resolving it with assertion 3, binding x_2 to Ann and y_1 to Carl. After this goal reduces to goal 9 we try to prove the first disjunct

mother(Dave, Carl)

which fails.

Before PROLOG-like backtracking can take place the other disjunct must be tried; when it fails as well the interpreter backtracks to goal 7 and uses the second conjunct of assertion 3 to get a new binding and subsequently succeed in the rest of the proof.

7.6 Reversing programs

One of the features often mentioned by PROLOG fans is the ability to run programs "backwards," i.e., to change the role of some of the input and output variables. This quality is the result of the symmetry of unification which can instantiate different variables of the goal used to call the program. Unification is a more expensive operation than the standard mechanisms used for argument bindings in other programming languages. Therefore, when compiling PROLOG programs for more efficient execution (for example on DEC-10 PROLOG [Pereira and Warren 82]) *mode declarations* can be used to specify in advance the input or output role of the arguments of predicates. The compiler can take these declarations into effect to produce more efficient code by not applying unification unless necessary. When unification is not applied to arguments declared to be input variables, the capability to run programs in reverse is lost. Thus the programmer has the freedom to choose between efficiency and flexibility.

In order to have more control over the execution of PROLOG programs, PROLOG has the *cut* operator to limit backtracking and to enable the system to simulate conditionals (*if-then-else* in TABLOG or **cond** in LISP). The addition of the *cut* once

again destroys the ability to reverse the role of variables since the reverse execution usually resorts to backtracking which the *cut* blocks.

One of the disadvantages of reverse execution is its heavy dependence on the order of the assertions in the program. In the examples below we will see how reverse execution fails in PROLOG because of the order dependency and the attempt to evaluate unbound variables.

Since the recommended programming style in TABLOG is order-independent, reverse execution is somewhat against the desire to keep the procedural interpretation as close as possible to the logical one. TABLOG's ability to partially support reverse execution can be viewed as a curiosity rather than an important issue.

As in PROLOG, the use of unification as the binding mechanism in TABLOG makes reverse execution available as long as predicates are used to define programs. The use of functions to code programs and the directional use of equality causes this mode of execution to fail in some cases.

I believe that it is reasonable to expect the possibility of reverse execution when dealing with relations, especially if we use a logic-programming language for database queries. When using the functional form of programs, one must be aware of the properties of functions in mathematics and logic and recognize the possible loss of ability to run programs backwards. The availability of both functional and relational programming styles in TABLOG lets the user choose.

The directionality of the use of "=" in the assertions does prevent the system from succeeding in running programs backwards in some cases. Nevertheless, many programs will still run backwards even when using equality. Any program that can run backwards in (pure) PROLOG and is coded in the same form in TABLOG will be executed successfully by the TABLOG interpreter as well.

If we use the following definition of append

append([ ], *u)* = *u*
append(a:oti, *v)* = a?oappend(u, *v)*

and we pose the query

[1,2]= append(l o y, *z)*

the TABLOG interpreter will come up with a correct answer, binding *y* and *z* to [ ] and [2] respectively. If this solution is rejected the other result, *y* <— [2], *z* <— [ ], will be reported.

Note that although we use $=$ in a directional way when defining functions, equality has its regular meaning in logic when used in a goal. Thus the system will return the same answer if the specified goal to prove is

$$\mathbf{append}(1 \circ y,\, z) = [1,\, 2].$$

By reversing the order of the assertions in the **append** program we get a version which is more efficient for running forward on a sequential order-dependent TABLOG interpreter. This is the result of first trying the case of the nonempty first argument which in most cases will actually match the current goal.

$$\mathbf{append}(x \circ u,\, v) = x \circ \mathbf{append}(u,\, v)$$
$$\mathbf{append}([\,],u) = u$$

We can still run this program backwards; for example with the goal

$$[1,\, 2] = \mathbf{append}(1 \circ [\,],\, z)$$

and get the correct result [2] as the binding of z.

The following example shows that the directional interpretation of equality to define rewrite rules for the reduction of functions can cause problems when trying to execute the program in reverse. We start with the goal

$$[1,\, 2] = \mathbf{append}((1 \circ y), z).$$

Under a standard PROLOG interpreter, the PROLOG equivalent of this program will run and solve the corresponding PROLOG version of the goal, finding the two possible values for y and z and stop. Under the sequential order-dependent TABLOG interpreter, the computation will diverge, introducing more variables into the list which represents the desired output:

$$[1,\, 2] = 1 \circ \mathbf{append}(y, z)$$
$$[1,\, 2] = 1 \circ (x \circ \mathbf{append}(y_0, z))$$
$$[1,\, 2] = 1 \circ (x \circ (x_0 \circ \mathbf{append}(y_1, z)))$$
$$[1,\, 2] = (1 \circ (x \circ (x_0 \circ (x_1 \circ \mathbf{append}(y_2, z)))))$$
$$\ldots$$

[Reddy 85b] suggests a technique called *lazy narrowing* that helps avoiding the behavior demonstrated in this example. If we adapt his method to our framework, for the example above it will imply that after the generation of the goal

$$[1,\, 2] = 1 \circ (x \circ \mathbf{append}(y_0, z))$$

resolving with the first assertion will fail. This failure occurs because under this approach we recognize that the equality in the result of the resolution

$$[1, 2] = 1 \circ (x \circ (x_0 \circ \mathbf{append}(y_1, z)))$$

must be *false* because we can never solve the implied equality

$$[\,] = (x_0 \circ \mathbf{append}(y_1, z)).$$

This is the case because both the constant 2 and the operator $\circ$ are constructors. At this stage a system with lazy narrowing (or its equivalent in the tableau system) will try using the second assertion and will be able to terminate. The difference between this and the way TABLOG works is that TABLOG will keep trying to reduce the nested **append** terms.

Note that if we know in advance that we might want to execute the program backwards we could define it using the predicate **appending** and get a program that will run exactly like the corresponding PROLOG one:

$$\mathbf{appending}(x \circ u, v, x \circ w) \leftarrow \mathbf{appending}(u, v, w).$$
$$\mathbf{appending}([\,], v, v).$$

Trying this program with the goal

$$\mathbf{appending}(y, z, [1, 2, 3])$$

will generate (after forced backtracking) all the pairs $\langle y, z \rangle$ satisfying the relation defined by this program.

Let us look again at the sorting program **qsort** that uses the **append** program:

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. $\mathbf{partition}(x, y \circ u, u_1, y \circ u_2)$
   $\leftarrow\ y > x\ \wedge\ \mathbf{partition}(x, u, u_1, u_2).$

For example, if we give the goal

[1, 2, 3] = **qsort(*)**

to the (sequential, order-dependent) TABLOG interpreter, it will succeed in running the program backwards to generate the solutions [3, 2, 1], and [3, 1, 2]. Although the program is order independent for forward execution, changing the roles of input and output makes it very order dependent. Since we do not promise backwards execution for functions in any case, the subset of solutions that do succeed can be regarded as an incidental feature. Note that the same program in PROLOG (using only predicates but preserving the order), will result in an error. The point is that while the TABLOG interpreter delays the evaluation of the arithmetic relation $\leq$ when the arguments are not concrete numbers, PROLOG tries to evaluate an unbound variable, which results in an error.

If we change the definition of **qsort, append,** and **partition,** so that the case for a nonempty input precedes the one for empty input, we get a program which is more efficient in most cases when executed forward, using sequential order-dependent evaluation. The same program however, will diverge in both TABLOG and PROLOG when executed "backwards," as new unbound variables will be generated with each recursive call and will never get instantiated.

CHAPTER 8

IMPLEMENTATION

8.1 Introduction

A prototype interpreter for TABLOG is implemented in MACLISP. The interactive implementation can serve as a program construction facility, debugger, and interpreter.

Because the interpreter is built over a versatile theorem prover, the overhead is high and the execution of programs is slow, but efficiency has a low priority in the current stage of research. The performance will be improved considerably by the introduction of faster unification, faster simplification and fast rewriting of terms.

All the examples mentioned in this dissertation were executed on the interpreter. Appendix B is a short guide for using the implemented interpreter described here.

The interpreter has been implemented in MACLISP under the WAITS operating system on the SAIL computer system. The nonclausal theorem prover that serves as the foundation for the implementation has been ported to run on other MACLISP systems as well as on Symbolics 3600 Lisp Machines and FRANZLISP under UNIX.

8.2 The language

Program assertions are formulas in first-order logic. Although standard logical connectives, basic arithmetic operations and predicates, and the list constructs are predefined, the parser depends on the user's definition to recognize the syntactic categories of the symbols used. All nonstandard symbols must therefore be declared before being used; undeclared symbols are taken to be constant or function symbols according to the context. Some standard symbols have a few alternative notations.

The parser is implemented as a shift-reduce parser. It uses the known precedences of the built-in operators and the user-specified relative precedences of the new constructs introduced in the program.

8.3 The tableau

The tableau is kept in a MACLISP array and all the derived subgoals are kept together with their corresponding output and information about about how each subgoal was derived.

Initially the user enters the assertions of the program and then one goal to be executed; later more goals can be entered and executed one at a time.

8.4 Mode of execution

Before executing a program the user can specify that it should be run in a *single-step* mode. In this mode the execution will pause after the first subgoal is derived; at this stage the user can specify the number of steps to be executed next.

A program can be executed in *verbose* mode in which all derived subgoals are displayed when generated. In the standard mode only the final goal (hopefully *true*) and the associated output are displayed.

8.5 Indexing

Indexing is the term used in the PROLOG community to describe the way predicates are connected to their definitions. The DEC-20 PROLOG compiler ([Pereira and Warren 82]) uses a hashing function based on the predicate name and the first argument to store and retrieve pointers to definitions. Some LISP implementations use the **expr** or **subr** properties to point to the definition of a function.

The implementation of TABLOG uses *hooks* as the indexing mechanism.

Each assertion has hooks associated with it to denote all the functions and predicates defined by this assertion. There are four types of hooks associated with each assertion, corresponding to the different inference rules that can be applied between an assertion and a goal:

Term hooks are the terms that are defined by this assertion.

Equivalence hooks are the (atomic) formulas that are defined in this assertion on the left-hand side of an equivalence;

Positive hooks are the atomic formulas defined with a positive polarity;

Negative hooks are the atomic formulas defined with negative polarity.

An atomic formula $\mathbf{p}(t_1, \ldots, t_2)$ is defined by the assertion $\mathcal{A}$ (and will appear in the hooks) if $\mathbf{p}(t_1, \ldots, t_n) \in D_r[\mathcal{A}]$ (see Section 4.1) and thus the assertion $\mathcal{A}$ is a *defining formula* for $\mathbf{p}$. The atomic formula will be placed in the appropriate hook depending on the polarity of its occurrence in $\mathcal{A}$; it will be included in the equivalence hooks if and only if it is defined using equivalence.

Similarly a term g(*i,... ,<m) is defined by the assertion *A* and will be placed in the term hooks if g(<i,..., t_m) € 2?/[«4].

When the assertion is used to reduce a term or a formula the expression being reduced will be unified against the appropriate hooks to test the applicability of the assertion to the reduction.

Associated with each (nonprimitive) function or predicate symbol we have a list of the assertions in whose hooks this symbol appears. The order of assertions in this list is the order used when trying to reduce an expression involving this symbol. This mechanism is implemented using LISP's property lists. This definition list and the hooks are also used to implement backtracking.

8.6 Simplification

Whenever a new goal is generated it is simplified to eliminate occurrences of *true* and *false.* The simplification is built into the theorem prover and is not considered an extra derivation step. (The general deductive-tableau framework assumes that such simplification will be done using transformation rules).

Occurrences of arithmetic expressions with all the arguments being integers are simplified by calling the appropriate function of the underlying MACLISP system. This means that $5 \geq 3$ will be simplified to *true* while $x * (4 - 4)$ will be transformed to $x * 0$ (and not to 0). For predicates a translation from the LISP representation of truth values, t and **nil,** to the standard truth values *true,* and *false,* takes place.

The simplifier used is almost the same as the one used for an interactive program synthesis system that was implemented by the author earlier. It is one of the bottlenecks in the interpreter that can be improved to give a faster TABLOG system.

The simplifier has built-in knowledge about the properties of the standard propositional connectives and *and-or* transformation. When simplifying formulas it eliminates all occurrences of *true* and *false* (unless the formula itself reduces to *true* or *false).* The expressions simplified are not converted into normal form, but retain their original structure as much as possible.

CHAPTER 9

THEORETICAL ISSUES

9.1 Introduction

This chapter is an attempt to study the semantics of TABLOG and the relation between its declarative and procedural interpretations. Most of the chapter is devoted to explaining why certain desirable properties of pure LISP and PROLOG, which enable us to reason about them, do not hold for TABLOG. We should remember however that in many cases theoretical properties are secondary to expressive power in a programming language.

In particular, we will discuss the relation between completeness, fixedpoints, consistency, and reasoning about programs, and will study the problems that arise when dealing with TABLOG.

I also describe what can be done to improve the situation, from a theoretical point of view, by either extending the proof procedure or by restricting the language.

9.2 Completeness

One of the most basic and most important relations between declarative and procedural semantics is *completeness.* The completeness property means that if the logical sentence associated with a program is valid, then the proof procedure will be able to prove the goal using the given assertions (and to compute the values satisfying the goal). Completeness is a property that gives us confidence in a proof procedure; this is important, for example, if we want to find all the solutions to a query.

When studying TABLOG, we unfortunately find that the proof procedure used as an interpreter is not complete. In this section we study the reasons for the lack of completeness and what can be done to improve this and at what cost. Once

and the restriction on the deduction engine underlying TABLOG's program execution were made consciously to achieve predictability and the possibility of more efficient implementations.

An important feature of the language of Horn clauses is the completeness of the SLD resolution proof system for formulas in this language: all the formulas that are logically implied by a set of Horn clauses are provable using this proof system. The completeness result holds if the proof procedure selects the appropriate clause to resolve at each step.

This property breaks down for the standard way PROLOG interpreters work; as these interpreters traverse the proof tree in a depth-first mode, the computation might follow an infinite branch without ever starting to explore the branch leading to a proof. Using a complete search strategy like breadth-first would cure this problem; unfortunately, breadth-first search is very space inefficient. This divergence is one form of incompleteness; another form is failing to continue the deduction when a proof does exist. PROLOG interpreters do not exhibit this form of incompleteness. When such an interpreter stops with failure, it will never be the case that the goal it tries to prove is implied by the program assertions.

Completeness is crucial for PROLOG because its absence can lead to unsoundness of a proof system when it is extended to include negation-by-failure. For instance, when an incomplete proof procedure so augmented tries to prove -vP, it is possible that there is a proof of *P* but that the proof system cannot find it; negation-by-failure implies that the interpreter will succeed in the proof of -vP.

The nonclausal resolution rule is complete for the language of (skolemized) first-order logic ([Murray 82]). The completeness holds if the appropriate assertions or goals are selected and if the right set of subformulas is unified.

The interpreter for TABLOG does not have this completeness: when making the proof system more directed and efficient, we do not use all the power of the general framework and lose completeness. The loss of completeness is visible in two forms: divergence and failure. The interpreter might diverge although a proof does exist using a different order of applying the inference rules to the assertions; the interpreter might also stop with failure even though the goal is actually implied by the program assertions. In the rest of this section I will describe some of the reasons for this latter sort of incompleteness.

The completeness proof of [Murray 82] is written for a system that always matches one atomic subformula against another one of the opposite polarity. Thus, imposing these restrictions on TABLOG is not the source of its incompleteness.

Each of the subsections of this section is devoted to one of the sources of incompleteness of the TABLOG interpreter. The first problem arises from the directional use of equality and equivalence in definitions.

9.2.1 Directionality of definitions

The directional use of equality and equivalence makes the procedural semantics more intuitive and prevents some infinite loops in the execution of programs; this same restriction hurts the completeness of the theorem prover when compared with the declarative semantics.

Example 9.1.
Given the program

$$\textbf{nervous}(x) \equiv \textbf{tired}(x).$$
$$\textbf{nervous}(\text{Ben}).$$

We cannot prove the goal

$$\textbf{tired}(\text{Ben}).$$

The problem is that although the program assertions imply the goal, none of them is a defining assertion for **tired**.

9.2.2 Lack of factoring

The clausal resolution principle that was described in Chapter 2 is the only inference rule for a complete proof system. The unification of more than one literal from each clause is called *factoring* and is sometimes used as a separate deduction rule with the resolution rule restricted to unifying one literal from each clause.

The nonclausal resolution rule as introduced in Chapter 3 and used in the TABLOG interpreter is actually a generalization of the restricted form of resolution to nonclausal formulas. To have completeness, the nonclausal resolution rule has to be a generalization of the more powerful clausal resolution rule and this is the form it actually takes in [Manna and Waldinger 80]. Rather than unifying a pair of subformulas, sets of subformulas are unified. I refer to the more general rule as resolution with factoring.

The following example demonstrates the need for some sort of factoring.

Example 9.2.
Given the tableau

Assertions	Goals	Output
1. p(«,y)Vp(y,aj)		
2.	p(a,*)Ap(s,a)	*z*

If we use resolution without factoring to resolve the goal with the assertion, we can choose among four selections of the literals to match. All these matches will, however, result in the same goal

3.	***false***	*z*

which does not lead anywhere.

If on the other hand, we apply resolution with factoring, we can get the success goal

4.	***true***	*a*

in one step, by unifying all four literals simultaneously. If we factor only the assertion, we can get this same goal in two steps by resolving on one conjunct at a time and getting the intermediate goal

5.	p(a,a)	***a***

This example, while showing that at times we might need to have the general form of resolution, is not a typical program. I do not expect the lack of factoring to cause problems in any real TABLOG program. Therefore, the TABLOG interpreter does not include this form of the rule. Two important points against the inclusion of factoring are the cost of the extended search space and the loss of predictability in some cases.

A possible approach to adding factoring to the proof procedure without a massive expansion of the search space is to take the other extreme. Instead of always choosing exactly one literal from each entry we will choose a maximal set of unifiable atomic formulas whenever applying nonclausal resolution. This approach is more efficient than just adding the most general version of factoring because the

candidates for such unifiable sets of formulas can be detected and marked as part of a preprocessing of the program. Another advantage is that when using this extension, we still make each deduction step slightly slower, but we do not increase the branching factor of nodes in the proof tree, i.e., the number of possible results of applying deduction rules at each point of the proof. Under this approach, for the above example we will have only one possible deduction, the one that leads to the desired goal in one step.

Unfortunately, just like the restricted version of resolution that always matches exactly one atomic formula in each resolution step, the version just described above also leads to incompleteness, as the following example shows:

Example 9.3.

1.	$\mathbf{p}(x,y) \vee \mathbf{p}(x,z) \vee \mathbf{q}(y,z) \vee \mathbf{q}(y,x)$	
2.		$\mathbf{p}(a,b) \vee \mathbf{p}(a,c) \vee \mathbf{q}(b,c) \vee \mathbf{q}(b,a)$

If we always apply factoring there are four ways to resolve the two entries, depending on which disjunct of the goal is resolved upon. Each such resolution will unify two disjuncts of the assertion against one from the goal. The goals that we can derive are

3.	$\neg\mathbf{q}(b,b) \wedge \neg\mathbf{q}(b,a)$
4.	$\neg\mathbf{q}(c,c) \wedge \neg\mathbf{q}(c,a)$
5.	$\neg\mathbf{p}(c,b) \wedge \neg\mathbf{p}(c,c)$
6.	$\neg\mathbf{p}(a,b) \wedge \neg\mathbf{p}(a,a)$

None of these goals is provable. ∎

Another disadvantage of including the more general form of resolution in a proof system is that it might cause the procedural semantics to be less predictable. This problem is particularly important when dealing with a sequential interpreter where the programmer might utilize the left-to-right order of the subformulas in each assertion.

9.2.3 Goal-Goal resolution

The (nonclausal) resolution rule takes four different forms in the original deductive-tableau proof system; the forms depend on the role of the entries (i.e., are they assertions or goals?) that are involved in the resolution and the polarity of the subformulas resolved upon. The two forms that involve an assertion and a goal are used by TABLOG but the other two forms that operate on a pair of assertions or a pair of goals are not.

The following example shows why we need goal-goal resolution.

Example 9.4.

	Assertions	Goals	output
1.	$\mathbf{p}(1) \vee \mathbf{q}(2)$		
2.		$\mathbf{p}(x) \vee \mathbf{q}(x)$	x

resolving 1 and 2 matching $\mathbf{p}(1)$ with $\mathbf{p}(x)$ we get

3.	$\neg\mathbf{q}(2)$	1

alternatively we can resolve 1 and 2 matching $\mathbf{q}(2)$ with $\mathbf{q}(x)$ to get

4.	$\neg\mathbf{p}(1)$	2

None of these goals can be further resolved with the single assertion of the tableau.

Applying goal-goal resolution between 2 and 3 will produce

5.	*true*	*if* $\mathbf{q}(2)$ *then* 2 *else* 1

Although goal-goal resolution lets the proof succeed, the solution that we get in this case is a conditional expression that might not always be considered an acceptable answer. The need for giving such conditional answers is related to the problem of models of disjunctive assertions: in some models the answer will be 1 and in others it will be 2; without further information about $\mathbf{q}(2)$ (or $\mathbf{p}(1)$) we cannot know which model of the assertion should be chosen.

The use in TABLOG of only the goal-assertion and assertion-goal forms was chosen to make the deductive process more efficient and predictable. It plays a major role

in the conversion from a general proof system into a logic-programming system. We should never expect to resolve two assertions if we expect our programs to be consistent. This is true because if the program is consistent we need to resolve with a goal at some point so we can start by resolving with the goal and resolve with the assertions one at a time.

In contrast, PROLOG does not suffer from the absence of resolution between two goals or two assertions. Because there is exactly one positive subformula in each assertion and there are no positive subformulas in the goals, such resolutions are not even possible. Each SLD resolution step between a goal and a Horn clause keeps this property invariant.

When all the formulas are in clausal form, the nonclausal resolution system reduces to standard resolution. When they are all Horn clauses, the TABLOG proof procedure, based on nonclausal resolution, and the PROLOG proof procedure, based on SLD resolution, have exactly the same behavior. This implies that all the completeness results that hold for PROLOG hold automatically for TABLOG when programs are restricted to the language of Horn clauses (without equality).

In addition to the above problem with answers that are not fully specified, the introduction of goal-goal resolution also affects the efficiency and predictability of the interpreter, since the number of possible rules to apply at each stage of the proof (computation) can get much larger. This explosion can be somewhat controlled by using *connection graphs* ([Stickel 82], [Kowalski 75 & 79]) to link predicates and terms to possible matching expressions.

9.3 Proving program properties

Programs in logic-programming and functional-programming languages are generally easier to reason about. The simple declarative semantics helps to perform such reasoning in a straightforward way. In TABLOG it seems that the assertions of a program can be used directly to reason about the function or relation computed by such a program; unfortunately, not all the assertions in the program are used to compute a function or a relation and the assertions by themselves do not fully describe the computation.

Example 9.5.
Let us look at the following program

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

A sequential TABLOG interpreter will compute the same **append** function given this program or the standard (correct) version of the program. It is clear, however, that from the assertions of the program given here we can prove the commutativity of **append**. Of course, this is not true for the standard version of the program. The problem is that the interpreter does not use the last assertion in the program unless the first solution is rejected and backtracking is invoked to search an alternative solution. On the other hand, the proof system we use for reasoning will generally disregard the order of the assertions in the program and use only assertions that are relevant to the proof.

Viewed as a statement in logic, the program given here is actually inconsistent because we can, for example, deduce from it

$$[1,2] = [2,1]$$

which should be false under the intended interpretation for lists and also in our proof system, because $\circ$ is a primitive constructor. Therefore, it is not surprising that many facts can be proved about this program.

If we really want to reason about programs we have to take into account the order of evaluation. In the case of sequential TABLOG interpreter and the program above, the (second-order) sentence we should use for the reasoning is actually

$$\begin{aligned}(\forall u\, v\, v_1\, x_2\, u_2\, v_2\, u_3\, v_3)[&\mathbf{append}(u,v) = \\ &\text{if } ((\exists\theta_1)(u = [\,] \wedge v\theta_1 = v_1)) \text{ then } v_1 \\ &\text{else if } ((\exists\theta_2)(u\theta_2 = x_2 \circ u_2 \wedge v\theta_2 = v_2)) \text{ then } x_2 \circ \mathbf{append}(u_2, v_2) \\ &\text{else if } ((\exists\theta_3)(u\theta_3 = u_3 \wedge v\theta_3 = v_3)) \text{ then } \mathbf{append}(v_3, u_3) \\ &\text{else } \bot].\end{aligned}$$

This can be simplified to

$$\begin{aligned}(\forall u\, v\, x_2\, u_2)[&\mathbf{append}(u,v) = \\ &\text{if } (u = [\,]) \text{ then } v \\ &\text{else if } ((\exists\theta_2)(u\theta_2 = x_2 \circ u_2)) \text{ then } x_2 \circ \mathbf{append}(u_2, v) \\ &\text{else } \mathbf{append}(v, u)].\end{aligned}$$

If we know that the inputs for this program are always lists, which always take the form $[\,]$ or $x \circ u$, we can actually prove that only the first two cases can occur. Therefore, for such inputs we can reduce the sentence to

$$\begin{aligned}&(\forall u\, v\, x_2\, u_2) \\ &\quad [\text{if } (u = [\,]) \text{ then } \mathbf{append}(u,v) = v \\ &\quad \text{else } (\exists\theta_2)(u\theta_2 = x_2 \circ u_2) \wedge \mathbf{append}(u,v) = x_2 \circ \mathbf{append}(u_2, v)]\end{aligned}$$

If we write the program above in an order-independent PROLOG, we will get a program which is actually consistent because it specifies a relation and not a function. Special care should be taken however, when proving properties of such a program (without taking the order of evaluation into effect).

In general, when we have only one-way implications in a program as in the PROLOG version of the **append** function, we do not have much information about the properties of the relation computed. The program gives us sufficient conditions for the relation to hold but not necessary conditions.

The fixed-point characterizations give a better ability to reason about programs.

9.3.1 Fixed points

The declarative semantics of a functional language such as (pure) LISP has been extensively studied using the theory of fixed-points of programs. It has been shown (cf. [Manna 74] Chapter 5) that some computation rules for pure LISP programs will actually compute the least fixed-point of the functional associated with the program. The results of these studies make it possible to get a better understanding of LISP programs and to prove theorems about their behavior. In particular the relation between the evaluated function and the fixed-point enables us to use the program as an equation in the logic and to prove properties of the computed function from it; these ideas are described in [McCarthy and Talcott 80].

The semantics of a language like PROLOG which is based on first-order predicate logic, can be studied using model theory to describe the models that satisfy the relations defined by programs. The work described in [van Emden and Kowalski 76] and [Apt and van Emden 82], and summarized in [Lloyd 84], has shown that we can view models as fixed-points of the transformations defined by Horn clause programs. The interesting result is that the least fixed-point of the transformation associated with a Horn-clause program actually coincides with the least Herbrand model, which also coincides with the set of all ground atomic formulas logically implied by the program clauses. Those are essentially two forms of declarative semantics that agree on the models that they specify. These results are used by [Sterling and Bundy 82] in verifying properties of PROLOG programs.

These nice properties do not carry over to TABLOG. The functional part of the language is more complex than LISP's and, in particular, there is no simple functional defined by a program since a function can occur on the left-hand side of more than one equation. Evaluation is also not by simple substitution like in LISP, and conditionals cannot always be evaluated to choose one of the branches because

some predicates are not complete. Therefore, we cannot study the semantics of TABLOG as the fixed-point of such a functional.

The logical component of TABLOG is also too powerful to be easily studied. The first problem in trying to study the fixed-point of models is that TABLOG models do not, in general, have the *model intersection property:* a formula might have two models while the intersection of these models will not be a model of the formula.

Example 9.6.
Both

$$\{P(a)\}$$

and

$$\{P(b)\}$$

are Herbrand models of the formula

$$P(a) \vee P(b).$$

Their intersection, the empty set, is not a model of the formula.

In contrast, the models of Horn clauses do have the model intersection property and this is used in the process of proving the existence of the least Herbrand model.

9.4 Consistency of a program

Another problem with TABLOG that we do not find in PROLOG is that it is easy to specify an inconsistent program and then essentially any output will satisfy the program. Although it is possible that the interpreter will find only reasonable solutions, we cannot trust the solutions of an inconsistent program. Of course we have this same problem with program verification or program synthesis, where the input specification might be inconsistent and therefore trivially imply the program correctness.

Even without the introduction of real negation we still have to worry about consistency because we want function symbols to denote objects that have the properties of mathematical functions. In particular a function should be a mapping that maps every object to exactly one value. Because in TABLOG functions can appear on the left-hand side of more than one equational assertion, it is possible that the same term will be evaluated to more than one value, especially when backtracking occurs.

The TABLOG interpreter solves this problem by not allowing a term to be replaced by two different values, unless the conditions for the first replacement fail. Whenever the equality rule is used to reduce a term it is marked. If backtracking comes back

to the same point in the proof due to failure of another part of the goal, the term will not be reduced again, and the backtracking will propagate further up the proof tree. An exception to this is when backtracking comes back to the same point but an alternative reduction will bind some of the arguments of the function to a different value than the previous reduction; in such a case it is correct to use the alternative reduction.

9.5 Restricted subsets of Tablog

From the discussion so far it is clear that we have to pay for the expressive power of TABLOG by sacrificing the nice semantics possessed by LISP and PROLOG. If we restrict the class of programs allowed, we can hope to have more positive results for TABLOG's semantics.

9.5.1 The Prolog subset

If we restrict our programs to be in Horn clause form, the nonclausal proof procedure has the same behavior as SLD resolution. This implies that we can then use the results about fixed-points and completeness and use the program assertions to reason about the program as was done in [Sterling and Bundy 82] for example.

We have to remember, however, that even in this case we still have the problem that all the nice results about the theory of SLD resolution are based on the fact that the *selection function* (the S in the name of the system) makes the right choice. In practice PROLOG interpreters (like TABLOG's) always reduce a subgoal according to the order of the clauses in the program.

In contrast, the results for LISP are valid for real LISP interpreters because the order of evaluation (call-by-value in most cases) is taken into account in the development of the theoretical results.

9.5.2 Adding functions and equality

We now look at the subset of the language with equalities and functions added but with the formulas still restricted to be Horn clauses. This subset is essentially the same as the language of EQLOG. It seems that now we will have completeness for all the predicates except equality. However, the proof procedure is not a complete equality reasoning system because of the directional use of equality.

Example 9.7.
Given **the program**

$$f(*) \Leftarrow g(x - 1).$$
$$P(f(5)).$$

The TABLOG interpreter will not be able to prove the goal

$$P(g(4)).$$

The problem is that the interpreter cannot reduce *g(4)* to /(5) as we **do** not have any defining assertion for *g*. On the other hand, /(5) will not be **reduced** to *g(A)* because the interpreter only tries to reduce terms that appear in the goal and never terms that appear only in an assertion.

Adding the matching rule of [Manna and Waldinger 86] to the interpreter will solve this sort of problems. We should still investigate, however, the effect of adding such rules on the predictability of TABLOG because they introduce many possible matches.

9.6 Functions in Tablog

As mentioned earlier (in the section about fixed-points), the definition of functions in TABLOG can be complex since we can have the same function symbol on the left-hand side of more than one equational definition.

It is reasonable to expect that programs that are supposed to compute functions actually define functions.

Rather than enforcing the consistency of functional definitions I will now prescribe a condition that guarantees it. Unlike LISP functions, which diverge for the values on which they are not defined, a TABLOG function might also fail.

Definition 9.8. *Nonoverlapping*
For a function symbol /, we say that its definition is nonoverlapping if for any two assertions, *4i, and *Ai,* such that

$$/(*!,\ldots,*\text{»}) = * \ 6 \ Df[Ai]$$

and

$$f(t'_1,\ldots,t'_n) = t' \in D_f[\mathcal{A}_2],$$

whenever there exists a most general unifier θ of the two left-hand sides, i.e.,

$$f(t_1, \ldots, t_n)\theta = f(t'_1, \ldots, t'_n)\theta$$

then θ also unifies the right-hand sides, i.e.,

$$t\theta = t'\theta. \blacksquare$$

If all the functions in a program P have nonoverlapping definitions then all these functions have a unique value for each input tuple whenever they exist.

The nonoverlapping condition is sometimes too strong as it does not allow to distinguish between cases with the same form of the term but different values.

Example 9.9.
The definition of **gcd** in the program below is not nonoverlapping as the left-hand side of the equality of all the assertions are unifiable. The program still defines the **gcd** function uniquely for every pair of nonnegative integers because the conditions on the values of the arguments ensure that only one assertion is applicable for any specific pair.

$$\mathbf{gcd}(x, y) = \mathbf{gcd}(x - y, y) \leftarrow x > y.$$
$$\mathbf{gcd}(x, y) = \mathbf{gcd}(x, y - x) \leftarrow x < y.$$
$$\mathbf{gcd}(x, x) = x. \blacksquare$$

Since the nonoverlapping condition is too strong we do not to impose it as a syntactic constraint on all TABLOG programs. The weaker condition of *semantic nonoverlapping* is more appropriate but harder to check as it is generally undecidable if two conditions are equivalent.

Definition 9.10. *Semantic nonoverlapping*
The definition of a function in a program P is semantically nonoverlapping if for any two assertions, $\mathcal{A}_1$, and $\mathcal{A}_2$ in P, such that

$$f(t_1, \ldots, t_n) = t \in D_f[\mathcal{A}_1]$$

and

$$f(t'_1, \ldots, t'_n) = t' \in D_f[\mathcal{A}_2],$$

whenever there exists a most general unifier θ of the two left-hand sides, i.e.,

$$f(t_1, \ldots, t_n)\theta = f(t'_1, \ldots, t'_n)\theta,$$

and also there exists a substitution ρ such that

$$\mathcal{A}_1\theta\rho[false] \wedge \mathcal{A}_2\theta\rho[false]$$

is provable from assertions of the program P, then using these two assertions to rewrite θ also unifies the right-hand sides, i.e.,

$$t\theta = t'\theta. \blacksquare$$

This condition states that if $\mathcal{A}_1$ and $\mathcal{A}_2$ are both applicable definitions for a term $f(s_1, \ldots, s_n)$, then they should reduce it to the same term.

There are cases for which even this condition is too strong and we actually want the two definitions to replace the term by two equivalent terms; i.e., two terms that will be reduced to the same concrete term by the program. For most cases however the semantic nonoverlapping is the appropriate restriction on programs.

The nonoverlapping conditions are sufficient to ensure that wherever the function is defined the computation will find at most one value.

To prove the totality of a function we have to prove termination for all domain points as well as completeness of the predicates used in the definitions of the functions. This requires a formal definition of the behavior of the interpreter and a characterization of the cases for which it is complete.

9.7 The interpreter

The inclusion of real negation in TABLOG and the distinction between failure and negation makes the logic for reasoning about TABLOG programs a four-valued one. In addition to the standard truth values *true* (T) and *false* (F) we also have the extended values *fail* and *diverge* which will be represented by $\perp$ and ω, respectively.

The following truth tables for the boolean connectives with the extended logical values formalize the way the TABLOG interpreter treats these connectives. Each row corresponds to the result of evaluating the first argument of a connective and the columns correspond to values of second argument (when evaluated). For example, the first row in the table for disjunction shows that the evaluation is sequential and if the first disjunct evaluates to *true*, the second one is not evaluated at all and the result will always be *true*. For all the connectives, if the evaluation of the first argument diverges, the evaluation of the whole formula will diverge.

V	T	F	ω	±
T	T	T	T	T
F	T	F	u	±
uJ	uJ	(jj	u>	ω
⊥	T	J.	ω	±

A	T	F	uJ	±
T	T	F	<jj	±
F	F	F	F	F
ω	u	u	uJ	uJ
±	±	±	J	±

–*	T	F	u	J.
T	T	F	u>	⊥
F	T	T	T	T
uJ	u	ω	u>	ω
J.	T	±	ω	J.

=	T	F	uJ	±
T	T	F	uJ	⊥
F	F	T	uJ	X
us	ω	uJ	uJ	uJ
±	±	±	uJ	±

Chapter 10

Concurrent and Parallel Tablog

10.1 Introduction and motivation

The rapid advances in semiconductor technology in recent years have made parallel computers available; this same technology is rapidly approaching the theoretical limits of density and speed, making it clear that in order to keep up with the growing demand for computing power we must resort to the parallelism available in multi-processor computers.

Research on parallel execution of logic programs, especially PROLOG, has become very popular. The growing number of works on the subject does not indicate that the problem is solved; we still do not fully understand how to exploit parallelism in future machines and in languages like PROLOG or TABLOG.

In light of the wealth of recent publications on parallel and concurrent logic programming, I will not try to create a new approach to the subject but rather will try to identify the points that are special to TABLOG and to other languages that combine functional and relational programming. While concentrating on the problems that are specific to TABLOG, I will try to build on the works of others for problems that arise in the context of parallel execution of PROLOG.

This chapter does not propose ultimate solutions for the problems it describes, but rather suggests possible directions for developing solutions. The chapter presents a model for the parallel execution of TABLOG programs that is also applicable to other attempts to unify relational and functional programming. Some parts of this chapter are suggestions for future research rather than completely worked out ideas.

While the parallel version of TABLOG described in the rest of this chapter does not exist everything that was described in the previous chapters was implemented.

The next section presents an overview of the two basic approaches to the ex-

10.2 Logic programming and parallel computation

10.2.1 Implicit and explicit parallelism

We can distinguish between two types of languages for parallel computation: languages that deal with parallelism explicitly and those that treat it implicitly. The latter are expected just to give more efficiency when many processors are available while the former also enable the programmer to explicitly control the available processors or at least the logical processes. Most of the Algol-like languages for parallel programming are of the explicit type, which makes it easier to write programs to execute parallel algorithms. The languages of the implicit type, on the other hand, support better separation of logic from control and make it easier to write programs when we do not know (or do not care) how to parallelize them. One of the disadvantages of this type of programs is that in many cases parallelizing sequential programs results in algorithms that are far from being the optimal parallel algorithms for the problem.

[Conery 83] proposes the And/Or model for the parallel execution of logic programs. This model is based on the desire to gain efficiency when many processors are available without making the parallelism visible to the programmer. It makes PROLOG an implicit parallelism language. In the rest of this chapter I will refer to this language as PARALLEL PROLOG.

The attractiveness of this approach (and implicit parallelism in general) is the separation of logic and control which is the main idea of logic programming. The programming process is very high level with no reference to the number of processes and their communication, and there is no change in the semantics of a program in the transformation from (sequential) PROLOG to PARALLEL PROLOG.

Following this model, I will describe a procedural interpretation for the parallel execution of TABLOG programs that will deviate minimally from the declarative semantics of a deductive tableau.

[Clark and Gregory 83] and [Shapiro 83a] describe PARLOG and CONCURRENT PROLOG, respectively; both are extensions of PROLOG to explicit parallelism languages.

While based on PROLOG, these languages do not try to adhere to the original operational semantics of sequential PROLOG, but rather strive to be useful for real concurrent programming applications. Among the advantages of this approach are the extra control the programmer is given over the evaluation process, which results in the power to write more efficient programs, and the applicability of the language to more tasks, including process control programs.

10.2.2 Nondeterminism

Another way to view the differences between various approaches to parallel execution of logic programs is the way they treat the nondeterminism inherent in logic programs. In a PROLOG program, a predicate can appear in the head of more than one clause; the order in which these clauses are tried when solving a goal is in principle undeterminate; ([Kowalski 79] calls this *nondeterminismi). DonH-care* nondeterminism assumes that it does not matter which of the applicable clauses will be chosen; we expect all to result in acceptable solutions or even the same solution. *Don*^{y}*t-know* nondeterminism, on the other hand, anticipates the possibility of getting a solution that will have to be rejected later because of further conditions. Standard (sequential) PROLOG interpreters support *don't-know* nondeterminism by going through the alternatives, using backtracking, until a desired binding is achieved satisfying all the goal conjuncts; in a parallel model that supports this sort of nondeterminism, we expect to have some mechanism for getting alternative solutions if the current solution fails. Conery's PARALLEL PROLOG supports *don't-know* nondeterminism; it is implemented using `fail` and `redo` messages to communicate the rejection of a solution and the request for an alternative one, respectively.

PARLOG and CONCURRENT PROLOG both support *don'i-care* nondeterminism, which frees them from looking for alternative solutions once some solution is generated; control annotations enable the programmer to specify the desired solution.

10.2.3 Possibilities for parallelism

[Conery and Kibler 81] lists the following types of parallelism that can be applied to the execution of Horn-clause logic programs:

Or-parallelism: When many assertions (clauses) match a predicate, they can all be tried in parallel. Every process can compute a solution using another assertion

And-parallelism: Several conjuncts of the current goal can be reduced simultaneously. Each of these reductions is done by a different process.

Stream-parallelism: Sometimes a predicate can be evaluated even when its arguments are only partially evaluated. One process can compute using the available data while another can produce more data by binding shared variables.

Search-parallelism: If the database of assertions is very big, just finding the applicable assertions will be very expensive; parallelism can therefore be used for this task.

For most problems when we want to employ parallelism the problem of partitioning the task and then communicating between the parallel tasks and synchronizing them becomes harder as we try to achieve more degrees of parallelism.

When employing or-parallelism, the order of generation of the different solutions by the parallel activities may affect the ultimate solution. In particular, the way we treat computations seeking alternative solutions after the first solution is produced is closely related to the type of nondeterminism supported. The application of and-parallelism is somewhat difficult because the different solutions found in parallel must be compatible for all the variables shared among the conjuncts. The other two forms of parallelism mentioned above are less general and depend on the program and query.

The different research efforts on parallel logic programming take different approaches to choosing the degree of parallelism to employ.

For example, the approach of [Haridi and Ciepielewski 83], describing the Or-Parallel Token Machine, is to exploit only or-parallelism and execute conjuncts serially. The PARALLEL PROLOG model of Conery and Kibler utilizes both and- and or-parallelism, but the and-parallelism is restricted to "safe parallelism" which is enabled only if there is no conflict in trying to bind shared variables. An algorithm to find the scheduling of conjuncts for this version of and-parallelism is described in [Conery and Kibler 83]. The algorithm is based on a data-flow analysis of the program and the query, and it guarantees that the and-parallelism will produce the same solutions as will be produced by sequential interpretation.

10.2.4 Parlog and Concurrent Prolog

PARLOG and CONCURRENT PROLOG are extensions (actually modifications) of PROLOG to support parallel programming. Both PARLOG and CONCURRENT PROLOG want to support or-parallelism and and-parallelism simultaneously; to achieve this, both approaches extend the language of PROLOG with control constructs, which explicitly specify synchronization of possibly parallel processes.

In both dialects, a program is a set of Horn clauses with the body of each clause starting with a (possibly empty) sequence of guards.

In PARLOG a clause is of the form

$$P \leftarrow G_1 \wedge \cdots \wedge G_k | S_1 // \cdots // S_m$$

where $k \geq 0$ and $m \geq 0$, P is called the clause head, G_i's are the guards, and each S_i is a *sequential component* which is a conjunction.

In CONCURRENT PROLOG a clause is of the form

$$P \leftarrow G_1 \wedge \cdots \wedge G_k | B_1 \wedge \cdots \wedge B_m$$

where $k \geq 0$ and $m \geq 0$, P is called the clause head, G_i's are the guards, and each B_i is a goal.

The guards limit the nondeterminism and help control the or-parallelism. This is achieved by deferring the commitment to a particular assertion until its guards are successfully evaluated; if the guards are used appropriately, the commitment will ensure that the chosen assertion will lead to an acceptable solution if one exists. Once an assertion is commited to, the CONCURRENT PROLOG semantics implies *don't-care nondeterminism*—if more than one solution is possible we do not care which one will be returned as the answer for a query.

The other component of the control annotations specifies the flow of data among the various occurrences of shared variables in a conjunctive goal and thus helps solving binding conflicts that arise in and-parallelism. The shared variables serve as communication channels among the parallel processes.

While in PARLOG exactly one sequential component in each clause body is marked as the *producer* of the binding for each variable, in CONCURRENT PROLOG the notion of *read-only variables* is introduced. A variable occurrence annotated as read-only must be a *consumer* of bindings while all other occurrences can be either producers or consumers.

In both languages the shared variables serve as communication channels among the parallel processes. Note that although there is no sequential construct in CONCURRENT PROLOG, sequential conjunctions can be defined using the parallel conjunction and read-only variables.

10.3 Parallelism for Tablog

When extending TABLOG for parallel execution, we should consider both approaches to parallelism mentioned in the previous section. In fact, we can follow either one and get two different languages. One extension, which will be called CONCURRENT TABLOG, follows the approach of CONCURRENT PROLOG ([Shapiro 83a]) and requires the addition of control constructs: guards and read-only variables. This extended language does not preserve the procedural semantics of TABLOG in the same way that CONCURRENT PROLOG is not PROLOG anymore.

Following the implicit parallelism approach, we will study a model of parallel execution of TABLOG programs that can either agree with the set of solutions

generated by a sequential TABLOG interpretation, or (depending on the exact assumptions in the model) will give an order-independent procedural semantics for the logic program. The extended language following this model will be called PARALLEL TABLOG,

I do not suggest a definition for the terms *concurrent* and *parallel* and the difference between them; the names were chosen only to indicate the relation to the existing variants of parallel logic programming.

First we have to recognize the possibilities for parallelism that are available in TABLOG as a result of its syntax being richer than PROLOG'S. In particular, the existence in TABLOG of reducible functional terms and all the standard connectives, in addition to PROLOG'S conjunction, generates new forms of possible parallelism:

Argument-parallelism: The various arguments of a function can be evaluated in parallel. The argument-parallelism is similar in a sense to and-parallelism because in general all the arguments must be evaluated and the various parallel reductions may share variables that must be bound to the same value.

Nesting-parallelism: When reducing a functional term, we can choose to use definitions for nested function symbols before or after using definitions for the outer function symbol. Nesting-parallelism can be used to try these alternatives in parallel. If the call-by-name (outer-most function reduction) succeeds it can provide a short-cut in the computation, and the evaluation of the nested argument terms can be aborted. In many cases, however, no assertion will match the outer term in its original form so call-by-name has to be suspended and wait for the arguments to be evaluated.

This nesting-parallelism is similar to or-parallelism, since usually it suffices that one of the subprocesses succeeds and enables further reductions later. Usually if more than one possibility can be applied successfully different choices will lead to different solutions. If we want to make sure that the functions defined by the program have the properties of mathematical functions (exactly one value for each domain element) we can extend the restrictions on the form of the equations defining functions suggested by [Hoffman and O'Donnell 82]. If such restrictions are enforced, the nesting-parallelism will search for appropriate reductions in parallel but only one reduction will be successfully applicable. In such a case it will, in effect, be a special case of search parallelism, as there will be no need for coordination.

Connective-parallelism: This is the analog of argument-parallelism when applied to formulas rather than to terms. We treat this sort of parallelism separately since in TABLOG, like in standard logic, connectives have special predefined meaning.

Depending on the particular connective, this form of parallelism can behave like and-parallelism, or-parallelism, or argument-parallelism. While connective-parallelism that comes from explicit disjunction or implication in a goal can be treated as or-parallelism, most of the other connectives lead to and-like parallelism.

In order to make PARALLEL TABLOG realizable, we will have to limit the amount of parallelism we actually employ, rather than utilize all the possible forms of parallelism mentioned here and in Section 2.

10,4 Proposed syntax of Concurrent Tablog

While the syntax of PARALLEL TABLOG is the same as TABLOG's, in CONCURRENT TABLOG we will have a few extensions in the spirit of CONCURRENT PROLOG. Although we want CONCURRENT TABLOG to support *don't-know* nondeterminism, we still want to be able to control it. Guards are introduced into the language to limit the nondeterminism and allow the language to be useful without requiring backtracking. Guards make selecting the appropriate assertion to be used for the reduction of a goal depend not only on successful unification but also on explicit conditions. *Read-only variables* are added to control the synchronization of several processes and the flow of data among them.

Each CONCURRENT TABLOG assertion starts with a (possibly empty) guard followed by a defining formula (see Chapter 4). When the current basic expression of the current goal is defined by the defining formula in the assertion, after matching and binding the variables only the guard is evaluated first; the rest of the assertion will be used to actually reduce the basic expression only if the evaluation of the guard succeeds. Unlike CONCURRENT PROLOG, the guard itself can be any quantifier-free first-order logic formula.

Logically a guard in an assertion is the antecedent of an implication. The syntactic construct used to introduce guards is the double arrow, =4>, and it can also be written in reverse direction.

In CONCURRENT TABLOG we also interpret the conditional as defining a guarded command. When using *if"then-else* the condition part is taken to be a guard of the then-part and its negation is considered the guard for the *else-part;* the condition

has to be successfully evaluated to *true* or *false* and only then the corresponding branch can be used to reduce a goal.

While the guards help control the or-parallelism, the read-only variables introduced next are another synchronization mechanism that helps control and-parallelism and argument-parallelism.

Read-only variables were introduced in [Shapiro 83a] as a more elegant version of the synchronization annotation suggested in [Clark and Gregory 81].

A read-only variable is a variable annotated with '?'. It is the occurrence of the variable which is read-only. This annotation changes the way a variable is treated by the unification mechanism. Intuitively it means that a read-only variable cannot be instantiated by the solution of the predicate or term in which it appears but only by some other occurrence of the same variable.

A goal which has an uninstantiated read-only variable must wait for other processes to instantiate it. This way we can impose synchronization on parts of the program.

10.5 Model of computation

Functional programming and relational programming can both be regarded as reduction systems: either terms or formulas are reduced at each step. TABLOG, as a combination of these two families of languages can also be viewed as a reduction system.

The model of computation for the parallel execution of TABLOG programs is in the spirit of existing proposals for data-flow architectures that have been studied by various researchers. In this model we run a program by solving a goal. The goal is represented as a graph and computations proceed by transforming the graph. In principle, internal nodes in the graph are the various operators of TABLOG: connectives, predicates, and functions; the leaves are variables and constants. Each transformation is a replacement of some subgraph by a new one. These reductions are performed by procedures which are either predefined (for connectives and other built-in operators) or are derived from the program assertions. The order in which the reductions can take place is affected by the form of parallelism that we want to employ. The procedures for reducing the built-in connectives contain not only the logical transformations but also information to support the desired parallelism and order dependence when activating the nodes corresponding to subformulas.

To make sure that bindings are consistent, each variable that is shared in the program text or in the goal is also shared in the goal graph; binding an occurrence of such a variable by one of the subgoals will affect the other occurrences as well.

Sometimes (for example for or-parallelism) a private copy of the subgraph is manipulated by a procedure and only later the nodes in the global graph are replaced and the shared variables get bound.

In principle a formula can be naturally represented by a tree, however, since variable nodes are shared the goal graph is not a tree. When viewing the arcs of the goal graph as directed from every node to its children (arguments) in the formula the graph is a directed acyclic graph (DAG) with a distinguished root.

While the global graph representing the goal is conceptually shared, it can be implemented in a distributed way. In particular, we can envision a pool of active nodes which have to access only the structure below them. This model is similar in principle to the ALICE machine ([Darlington and Reeve 81]), the Or-Parallel Token Machine ([Ciepielewski and Haxidi 84]), and the REDIFLOW architecture ([Keller and Lindstrom 79]).

A node can be implemented as a *packet* (or a *token),* which is a logical unit consisting of various fields representing a term or a formula and status information. When a processor operates on such a packet we have a *process.* Data communication between processes is achieved via the shared variables. We can also envision some sort of control communication where a parent node can activate one or more of its children and a child node can signal its parent upon completion or failure of a reduction.

The fact that TABLOG supports both functions and predicates makes the model of execution somewhat more complex than the one described in other works (e.g. [Ciepielewski and Haridi 84], [Darlington and Reeve 81], [Conery and Kibler 81 & 83]); we have to be able to reduce both terms and formulas.

Different evaluation strategies (call-by-name, call-by-need, lazy evaluation) and forms of parallelism will be implemented by specifying different order of reducing packets representing nodes in the graph. The description of the control schemes is too elaborate and is left out of this dissertation as these schemes need to be evaluated further after being implemented or at least simulated on various examples. This is an area for further research and implementation.

The next section describes how the nonclausal resolution rule can be modified to better support the parallel reduction of nodes in the graph by minimizing the interdependence between such reductions.

10.6 Modified inference rules

While the nonclausal inference rules of the deductive-tableau proof system have the advantage of being applicable to formulas in any form, they have a major disadvantage—a global nature. Applications of the equality rule, the equivalence rule, and the nonclausal resolution rule result in new (bigger) formulas (at least before simplification) and use the whole formulas of the entries involved. This does not seem to be too promising for the parallel application of these rules. The problem here is that an application of such a rule creates dependency among the various components of the formula, which requires communication—the major problem in utilizing parallel computation. One inference rule that is helpful in being able to distribute the application of these rules is the splitting rule, which results in smaller formulas that can be treated separately.

[Traugott 85] describes *nested resolution*, a new version of resolution, and supplies a proof of its soundness and completeness. The idea is to replace occurrences of the subformula resolved upon in one of the entries by a modified copy of the other entry. Such replacements can be done locally with no reference or effect on the rest of the goal except for the shared variables.

In order to be able to apply the rule in a distributed manner we use a relaxed version of the rule, which allows replacing only some occurrences of a subformula in the assertion and goal (in Traugott's original version all occurrences are replaced). The soundness of this rule can be proved in a way analogous to Traugott's proof using the *polarity replacement proposition* of [Manna and Waldinger 86] that can be applied to zero or more replacements.

The first version is

assertions	goals	outputs
$\mathcal{A}\langle\mathcal{P}\rangle$		
	$\mathcal{G}\langle\hat{\mathcal{P}}^{-}\rangle$	g
	$\mathcal{G}\theta\langle\neg\mathcal{A}\theta\langle false\rangle\rangle$	$g\theta$

where the angle brackets notation, $\mathcal{G}\langle\hat{\mathcal{P}}^{-}\rangle$, indicates replacing some occurrences of $\hat{\mathcal{P}}$ that have negative polarity in $\mathcal{G}$; similarly $\mathcal{A}\langle\mathcal{P}\rangle$ denotes replacing some (possibly zero) occurrences of $\mathcal{P}$ in $\mathcal{A}$ without caring about their polarity. We replace the occurrences in the goal with the negation of a modified version of the assertion. While the polarity of occurrences in the original nonclausal resolution rule might

affect only the completeness but not the soundness of the rule, in this rule the soundness also depends on polarity of the replaced subformula.

The other version is for the case of resolving on a subformula that occurs in the goal with positive polarity in the tableau.

assertions	goals	outputs
$\mathcal{A}\langle\mathcal{P}\rangle$		
	$\mathcal{G}\langle\hat{\mathcal{P}}^{+}\rangle$	g
	$\mathcal{G}\theta\langle\mathcal{A}\theta\langle true\rangle\rangle$	$g\theta$

These rules can be executed locally. To reduce an occurrence of an atomic formula in the goal we need only to replace that occurrence by the appropriate formulas according to the polarity of the occurrence. Of course the unifying substitution θ still has to be applied to the rest of the goal to assure the correctness of this rule. This substitution is transmitted to the other parts of the goal and must be checked for compatibility in the same way that it is done for PROLOG programs.

While the polarity of $\hat{\mathcal{P}}$ in the goal determines the exact form of the rule that we can soundly use, we will still obey the polarity strategy and replace $\mathcal{P}\theta$ in the assertion by *false* only if $\mathcal{P}$ has an occurrence of positive polarity and by *true* only if there is a negative occurrence. This means that for the nested resolution rule, as for the original nonclausal resolution rule, we need to have occurrence of the matching subformulas of opposite polarities to be able to apply the rule under the restriction of the polarity strategy.

10.6.1 The equality rule

The locality of the nested resolution rule is very useful for concurrent execution. Unfortunately, we do not have an analogous version of the equality rule so the equality rule must still have a global nature. In particular, a new goal generated by this rule is generally a conjunction of the old goal with some replacements and the assertion used with some other replacements. The conjunction is formed at the top-level of the formula constituting the current goal.

When I describe the operational part of the model I will also show how to minimize the global effect of the equality rule.

10.7 Procedures

Every assertion in the program (partially) defines one or more functions or relations and is compiled into procedures for executing these definitions. When such a procedure is called, new packets are created and get thrown into the packet pool. In principle, all occurrences of a defined operator in an assertion can be compiled into one procedure that will then activate the applicable definitions as subprocedures, either in parallel or sequentially, depending on the scheduling policy used. Each of the applicable reductions implied by the various definitions will create a new subgraph; the appropriate method for combining the results of these reductions depends on the particular variant of the model used.

A procedure is a compiled instance of the negation of an assertion with occurrences of the atomic formula to be reduced by this procedure replaced with *true* or *false* according to its polarity. This compiled version of the assertion will replace an occurrence of an atomic formula in a goal according to the modified nested resolution rule that was described in the previous section.

Because the equality rule does not have a local version, and since we need some locality for parallelism, procedures for rewriting terms will be different from the ones based on nested resolution. For each applicable definition, the conjunct that has to be added to the goal will have to be completely solved before the local replacement of the term will actually take place. Since conjuncting with *true* does not have any effect, once the condition is reduced to *true*, it is correct to make the replacement.

When or-parallelism is applied, a term packet will be replaced by one of a set of new packets all of them with the return address of the reduced packet. The new packets share an *abort-packet* which is some sort of a semaphore. When one of the parallel processes succeeds (or commits) it will mark the abort packet or will replace the original term packet. When a processor starts to evaluate a packet, it first checks the status of its abort packet. A deterministic packet will pass to its children the pointer to its own abort packet and will then disappear. Depending on the type of nondeterminism policy followed, other processes will abort or wait for a request for more solutions.

A procedure is called when the operator of an active packet matches its name.

10.8 Discussion

We have studied here the basis for a family of models for the parallel execution of CONCURRENT TABLOG and PARALLEL TABLOG. Each of the alternatives has its own advantages and appropriate sets of problems that it is suitable for solving. Experimenting with implementation and different test problems on concrete architectures will give better insight into the trade-offs.

The data-flow model for CONCURRENT TABLOG and PARALLEL TABLOG is general enough to support the reduction of terms and formulas and represent their interrelation. The presentation in this chapter gives the basic idea on how to approach the implementation of the model but the details should be worked out for each choice of a particular version of the general model and an architecture used to implement it.

The main open question is how to implement such a scheme on a distributed computer system and how to isolate the bottlenecks in achieving maximal parallelism. There are different approaches to the implementation of the program database and the pool of goals awaiting reduction. They can be implemented as totally distributed, partially replicated, or centralized at one place; the choice of a particular scheme will have enormous effect on the efficiency of a real implementation. One solution will be to devise heuristics for the distribution of the program and goals that will work (relatively) well for most cases. An alternative approach is to let the programmer annotate the program with hints for the right amount of parallelism to apply. This approach was followed to some extent in [Gabriel and McCarthy 84] and [Shapiro 83a & 83b].

Since every real computer can have only a finite amount of resources (execution elements, memory, etc.) we want to prevent the proliferation of processes that are not expected to contribute to the solution of the problem being solved by the program. One example of such processes is those evaluating the guards of an assertion when the computation is already commited to another assertion for solving the same subgoal. Another example is the evaluation of arguments of a function when the term itself can be actually reduced based only on the arguments already reduced.

One approach to this problem is to assume that it is up to the implementation and to leave it as an engineering task to be solved together with the implementation of search-parallelism for example. The approach taken here however is to suggest how we can implement an abort mechanism in the model of computation described earlier.

A more general approach is the one used in the Or-Parallel Token Machine where another synchronization mechanism, the *abort frame*, is introduced. The abort frame is shared by several processes that check it when they get activated. It can be used to abort useless processes after one solution has been produced or after a specified number of solutions have been generated.

CHAPTER 11

RELATED RESEARCH

11.1 Introduction

Research on logic programming has become so popular that there are now a number of conferences dedicated exclusively to this field, and articles on the subject can be found in many computer science journals and conferences.

Recently the even more specific topic of designing and investigating languages that combine the relational and functional styles of declarative languages has been the center of many research efforts. The wealth of articles on the subject forces me to limit the discussion to only a few works. For a more extensive treatment see [DeGroot and Lindstrom 85] in which TABLOG is represented alongside other articles describing different approaches. This book is a good place to start for an intensive study of logic programming in the broader sense (as discussed in the introduction to Chapter 2).

An extensive bibliography of logic-programming research appears in [Balbin and Lecot 85]. A rich source of up-to-date information on logic programming in general and PROLOG in particular is the electronically distributed PROLOG digest, which is moderated by Chuck Restivo (RESTIVO@SU-SCORE.ARPA).

11.2 Computing with equations

In TABLOG we use equations to define ways to rewrite terms. The study of various rewrite systems and computing with equations have been active research areas.

[Hoffman and O'Donnell 82] describes a system for computing with equations. Like TABLOG this system treats equations in a directional way, but unlike TABLOG predicates are not supported and have to be encoded as functions. One-way pattern matching (rather than unification) is used when applying a rewrite rule (defined by an equation) to a term. Unlike OBJ and EQLOG, [Hoffman and O'Donnell 82] does not try to study the model theory corresponding to the language but instead takes a purely syntactic approach, studying only properties of the possible derivations implied by the equations.

The following restrictions are imposed on the equations of a program, in order to achieve clean behavior of computations:

1. No variable can occur more than once in the left-hand side of an equation.
2. If more than one left-hand side matches a term the right-hand sides must agree.
3. Two left-hand sides should never overlap on any term.

Restrictions 2 and 3 are especially important if we want to execute the equational program in parallel without any limitations of the parallelism.

While in TABLOG we can define some functions to be primitive constructors, in this framework all functions that do not appear on the left-hand side of an equation behave like TABLOG's primitive constructors; i.e., they are not reduced but their subexpressions are reduced when possible.

The choice of the expression (term) to reduce at each step can be considered "call-by-need" and starts from the outside; the arguments of a function (or predicate) will be reduced only if the function itself cannot be reduced with the current value of the arguments. This is the same as in the current implementation of TABLOG where we first try the outermost function or predicate and only if it cannot be matched against a definition will other possibilities be tried.

11.2.1 Oriented equational clauses

[Fribourg 84] introduces a language based on Horn oriented equational clauses, which is essentially the language of Horn clauses but with equality as the only predicate. The equations are oriented so they are always used from left to right, but, unlike the language of [Hoffman and O'Donnell 82], this language uses unification rather than plain pattern matching. While extending the language of Horn clauses to include functions and real equality, the language of [Fribourg 84] does not include

predicates and they are encoded using functions. This agrees with the approach of [Hoffman and O'Donnell 82] and contrasts with the PROLOG technique of encoding functions using predicates.

11.2.2 Computation using completion

Dershowitz and his colleagues extend their equational language, which is based on rewrite rules, to handle logic programming as well. While their language is still functional in syntax, it does embed relations and connectives as functions with boolean values.

In [Dershowitz 85] the Knuth-Bendix completion procedure is used as a computation engine. A program is a set of equations to be used as rewrite rules. The completion procedure generates from the program all the derived rewrite rules. To use this as a computation engine, a goal, which is a rewrite rule of the form

$$P(x, z) \Rightarrow \textbf{answer}(z),$$

is added to the program. P is the function that we want to compute, x is the input, and z is the output; **answer** is a special function symbol. A *linear* form of the completion procedure is used, in which only goals are matched against rules in the program while derived rules are not matched against each other. The use of this restricted form of the rule makes the procedure essentially equivalent to the proof procedure used by PROLOG over Horn-clause programs.

11.2.3 Obj and Eqlog

OBJ [Goguen, Meseguer, and Plaisted 82] is a language based on algebraic semantics of abstract data types; virtually all the programming in OBJ is done as part of the declaration of various data types and the operations associated with them.

EQLOG, the extension of OBJ to include logical variables and Horn clauses, supports both relational and functional programming, as well as their combination via narrowing.

EQLOG uses Horn clauses with equality as the logical basis, rewrite rules for the equational part, the usual PROLOG unification for the predicate part, and narrowing for solving equations with logical variables. The main advantages of restricting the logical part of EQLOG to Horn clauses are the completeness of the proof procedure and the existence of initial models for every program.

The language of EQLOG still enjoys the features of OBJ, including typing and type checking, generic modules and theories. As far as I know, there currently exists no implementation for EQLOG; the implemented version of OBJ, however, is

quite flexible and the programming environment supports libraries of functions and data types as well as interfaces to the EMACS editor and to the operating system.

[Reddy 85b] proposes another language which is based on narrowing. His language uses lazy narrowing, which enables infinite data structures to be handled successfully.

The languages described in this subsection do not have the full expressive power of TABLOG. They combine functional and relational programming without extending the set connective supported by the PROLOG; this approach enables them to have the completeness property.

11.3 Logic Programming based on natural deduction

[Haridi 81] presents an extensive study of a natural-deduction proof system and its applicability for the execution of logic programs. The language described there includes explicit quantifiers and is therefore richer than TABLOG. On the other hand, the rich set of inference rules available in the proof system make the deduction mechanism harder to follow and more expensive to implement. However, most of the examples that were included to demonstrate the power of keeping quantifiers either have the flavor of general theorem proving more than that of logic programming or just use enumeration of the domain to deal with the quantifiers.

Unlike the deductive-tableau proof system, the natural-deduction proof system used for the procedural interpretation of Haridi's language does not have an equality rule. This essentially precludes the possibility of directly reducing nested terms. Instead, every expression in the language is converted into a *basic form* that does not have any nested terms. The conversion is performed by introducing auxiliary variables to name terms. After this transformation takes place, all reasoning about equality is done using reflexivity and substitutivity axioms.

The appealing feature of the transformed version with no nested terms is that it makes choosing a subexpression to reduce straightforward and efficient. Since in the basic form all equalities have a variable on the left-hand side, the conversion of all programs and goals in TABLOG into such a form would allow us to eliminate the equality rule and instead always use either the nonclausal resolution rule or the equivalence rule. Note however that this conversion introduces back into the language many auxiliary variables that the functional notation is supposed to eliminate.

Under sequential interpretation, conversion into the basic form together with the left-to-right computation rule eliminates the choice between call-by-name and call-by-value; the left-to-right evaluation will always imply one or the other, depending on the exact way the basic form is constructed.

The treatment of unequality in Haridi's work is very similar to the one in TABLOG. One difference is that we do not replace an equality of the form

$$a = f(t_1, \ldots, t_n)$$

by *false* unless all the t_i's are concrete terms (a reasonable possibility is to enforce such a replacement whenever f is a constructor function). Haridi's language will assume that this equality is false whenever a is a constant.

Assertions must have an atomic formula on the left-hand side of an equivalence. As in TABLOG's defining formulas this restriction is imposed to ensure the correct procedural interpretation of equivalence.

11.4 Extensions of Prolog

Various works attempt to make PROLOG a more convenient language by adding to it a functional form of programming. Equality or reducibility predicates have been introduced into PROLOG by adding the appropriate axioms, or by extending the unification algorithm to unify terms asserted to be equal.

[Tamaki 84] introduces into PROLOG a reducibility predicate that is used to reduce terms to other terms. The reducibility predicate is directional and can be written in two ways:

$$t \triangleleft t'$$

or

$$t' \triangleright t$$

to declare possible reduction of t to t'. The meaning of this predicate is very similar to the use of equality for rewriting in TABLOG. To use this extension, all the axioms for equality except commutativity and predicate substitutivity for the $\triangleright$ predicate itself have to be added to every program.

There are two kinds of function symbols in the extended language: *f-symbols* and *d-symbols*; the former are used for describing real functions while the latter are used for data structures. This is similar to the difference between defined functions and constructor functions in TABLOG.

The f-symbols cannot be nested on the left-hand side of a clause. They can be nested on the right-hand side but all the nested occurrences are replaced by new variables when the clause is expanded.

In general, using these axioms does not force the rewriting of terms to be done to all occurrences of the term in the formula; the nice semantics is only for the case of confluent program, for which it does not matter if the replacement is done locally or not.

[Kornfeld 85] proposes an extension of the language PROLOG, called "Prolog-with-Equality" which allows the inclusion of assertions about equality. He introduces a special predicate **equal**(s, t) to specify that the two terms s and t should be unifiable. When an attempt is made to unify two terms, t and t', that do not unify syntactically, an equality theorem may be used to prove the two terms equal. If it is possible to prove **equal**(t, t'), the unification succeeds with the variable bindings introduced by the equality proof. This extension supports data abstraction with the advantages of object-oriented programming. The reduction of a term to another is not supported, however.

11.5 Other approaches

This section describes works that combine relational and functional programming in some form. The efforts described here are not direct extensions of PROLOG or based directly on computing with equations.

11.5.1 Hope and its extensions

HOPE as introduced in [Burstall, McQueen, and Sannella 80] is a functional language with pattern matching, strong typing and data abstractions. [Darlington, Field, and Pull 85] shows how to extend HOPE to have expressive power similar to that of logical variables.

The extended functional language (HOPE with unification) includes absolute set abstraction. This is the standard mathematical notation for a set of all the elements that satisfy some property. The variables used in the specification of the set behave like logical variables and allow convenient specification of many problems.

The limitation of a function returning one object (which might be a set or a tuple) is still present so some programs are still more complex than when using the relational style of TABLOG or PROLOG. By restricting the extensions to one additional construct, the absolute set abstraction, the language retains most of the simplicity characterizing functional languages. In particular, the run time behavior of functional programs is much simpler to control and understand than that of logic

programs, especially in a parallel context. Techniques such as graph reduction and data flow, which have evolved for the parallel evaluation of functional languages, take advantage of their simplicity of execution and are generally still applicable to the extended language.

The same article also proposes a technique for compiling certain types of functions defined implicitly within set expressions into explicit functions. This effectively involves synthesizing function inverses using the processes of symbolic unification and program transformation. When this can be achieved, logical variables can be eliminated altogether and replaced by function composition.

The research at Imperial College that has produced this language is also concerned with the development of the ALICE reduction machine on which HOPE and its extensions are to be executed. Most of the language is implemented within HOPE itself.

This approach is similar to the SUPER [Robinson 85] project (cf. 11.5.4) but does not suggest high-order functions. Both of these efforts pursue the language research in parallel to the development of reduction machines for parallel computation based on the data flow concept.

11.5.2 Functional Programming with logical variables

The approach described in [Lindstrom 85] starts from a purely functional programming language and extends it with logical variables. The graph reduction language FGL is extended to a language FGL+LV permitting formal parameter expressions, with variables occurring therein bound by unification. The theme of this approach is that the concept of the logical variable is orthogonal to the other features of PROLOG-like languages and can sensibly be incorporated into existing functional languages. This extension introduces a form of side-effects, since a function invocation can exert constraints on variables shared with other function invocations. To retain determinacy, even under parallel execution, the syntax of the language allows a function to occur only once on the left-hand side of a definition and places restrictions on the way unification can bind variables.

The limitations do not allow full freedom in manipulating logical variables and in particular there is no provision for answers to top-level queries to have unbound variables.

This language development has been done in conjunction with the REDIFLOW machine project and an implementation technique suitable for this and similar architectures is suggested.

11.5.3 Qute

The name QUTE was used by Masahiko Sato and his colleagues for a series of languages with the common theme of unifying logic and functional programming. Here I refer to the most recent form of QUTE, as presented in [Sato and Sakurai 85].

QUTE is basically a functional language, but it uses unification rather then pattern matching as the binding mechanism. As in the other works mentioned in this chapter, unification gives the language extra expressive power and lets the programmer write both ML-like programs and PROLOG-like programs. Although it uses unification, QUTE does not depend on resolution, and therefore the lack of the occur-check in its unification algorithm does not cause any problems. This absence of occur-check is necessary in the QUTE interpreter for defining recursive programs.

The evaluation algorithm of QUTE has the Church-Rosser property that allows and-parallelism without worrying about the order of evaluating the different conjuncts.

While the paper gives a completely formal semantics of QUTE, this semantics is defined in term of syntactical reductions and does not have any model theory. QUTE's notation for variables and their scope is awkward and a better form should be designed.

11.5.4 Super

[Robinson and Sibert 82] describes LOGLISP, a combination of PROLOG and LISP. In LOGLISP a PROLOG interpreter implemented in LISP can call the underlying LISP interpreter and also be invoked from LISP programs. This results in one interpreter for two languages with two different semantics.

The SUPER project described in [Robinson 85] is a descendant of LOGLISP but with an emphasis on getting cleaner semantics. SUPER is designed to combine lambda calculus, predicate logic, and set theory. It is intended to be interpreted on a multi-processor reduction machine. The approach of the researchers at Syracuse University working on the SUPER project is to develop a parallel reduction machine in conjunction with the language. This is similar to the attitude taken by those working at Imperial College on HOPE and ALICE and the simultaneous development of language (FGL+LV) and machine REDIFLOW at the University of Utah.

While Horn-clause logic programs have implicit quantifiers, SUPER will have only one quantification mechanism: the λ-abstraction. This scoping mechanism will be used to define the logical quantifiers and set abstraction using higher-order primitives.

Set abstraction equations are solved using two basic inference rules: decomposition and instantiation. Together, these two rules actually capture all the power of unification.

CHAPTER 12

DISCUSSION AND FUTURE WORK

12.1 Introduction

In this section I will survey some of the inherent problems of TABLOG that were not discussed earlier and suggest directions of future research and implementation related to TABLOG.

In Chapter 9 we observed that the expressive syntax of TABLOG costs us much in the ability to reason about programs and their correctness or termination. The absence of completeness also implies that we cannot always trust an interpreter to know that it has found all the answers to a query.

The next section is devoted to another problem, the absence of explicit quantifiers in TABLOG and the pitfalls relating to implicit quantification when introducing new variables.

TABLOG is still at its infancy, and to make it a useful tool, better implementations are essential. Future work on TABLOG should involve both better implementations and extensions to the language and its interpreter. Other topics for future research include the theory of TABLOG and methods for synthesizing or preprocessing non-procedural specifications into an efficient TABLOG program.

12.2 Implicit quantifiers

All the free variables of the assertions of a TABLOG program are implicitly universally quantified. This is simple when the assertion is a conjunction or disjunction, but gets more interesting when we have implication or equivalence.

For an implicative assertion, variables occuring only on the antecedent are actually *existentially* quantified inside the antecedent. Let us examine as an example the following assertion:

$$\mathbf{list}(u) \leftarrow \mathbf{empty}(u) \vee (u = x \circ v \wedge \mathbf{list}(v)).$$

The universal closure of this assertion is

$$(\forall u\, x\, v)[\mathbf{list}(u) \leftarrow \mathbf{empty}(u) \vee (u = x \circ v \wedge \mathbf{list}(v))].$$

which is equivalent to

$$(\forall u)[\mathbf{list}(u) \leftarrow (\exists x\, v)\,(\mathbf{empty}(u) \vee (u = x \circ v \wedge \mathbf{list}(v)))].$$

When we resolve with this assertion we will get a new goal with a conjunct containing x and v existentially quantified. Luckily, this is the programmer's intention when defining the list predicate using the assertion in this example.

The situation is not as simple when we use equivalence to define predicates. When we try to push the quantifiers to the right-hand side of the equivalence, the variables that occur only on the right-hand side of the equivalence are actually both universally quantified and existentially quantified. The form that takes effect depends on the polarity in which the equivalence is used. The programmer must be careful to make sure that the variables get the intended implicit quantification.

If we want to specify the **list** testing predicate using equivalence we would like to write

$$(\forall u)\,[\mathbf{list}(u) \equiv (\exists x\, v)[\mathbf{empty}(u) \vee (u = x \circ v \wedge \mathbf{list}(v))]]\,.$$

Since TABLOG does not have explicit quantifiers one is tempted to write the following program

$$\mathbf{list}(u) \equiv \mathbf{empty}(u) \vee (u = x \circ v \wedge \mathbf{list}(v)).$$

The universal closure in this case will be

$$(\forall u\, x\, v)[\mathbf{list}(u) \equiv \mathbf{empty}(u) \vee (u = x \circ v \wedge \mathbf{list}(v))],$$

which is different from the intended quantified definition above.

Trying to use this program to prove $\neg\mathbf{list}(w)$ for some given w will produce wrong results.

It is extremely important therefore to remember that the scope of the implicit quantifiers in the logical interpretation of a TABLOG program is always the whole goal or assertion.

12.3 Extensions to the language

12.3.1 Quantifiers

The current language does not support explicit quantifiers. Adding explicit quantifiers to the language will increase its expressive power. As was demonstrated in the previous section some problems lead naturally to programming with quantifiers.

Quantifiers may serve as a high-level construct for iteration on finite domains, easily generated domains, or finite structures (like sets or lists). For example we can use quantifiers over integers to achieve the partially recursive μ operator. [Bowen 82] introduces quantifiers freely into the language but the techniques used there would in effect work only for bounded quantifiers.

The problem with quantifiers is that while the extension of the syntax is simple, checking for the ramification of introducing a specific quantifier is very complex and usually depends on the semantics of the data domain. As will be demonstrated below, in many cases introduction of quantifiers will make the deduction necessary to satisfy the specification a very complicated process. Sometime it will possibly require decision procedures for the theory of the domain; such decision procedures are not available in most cases and even when they are available we probably would not want to regard this deduction as computation.

For example, it is very easy to define the subset relation using explicit quantifiers:

$$s \subset t \equiv (\forall x)[x \in s \rightarrow x \in t].$$
$$x \in u \circ s \equiv (x = u \vee x \in s).$$
$$\neg(x \in [\,]).$$

When we try to use this program (assuming quantifiers are allowed and have the right semantics) to evaluate a goal like

$$z = \text{if } [2,1] \subset [3,2,1] \text{ then } [3,2,1] \text{ else } [\,]$$

we will get an intermediate goal of the form

$$\begin{aligned} z = \ & \text{if } (\forall x)[x = 2 \vee x = 1 \rightarrow x = 3 \vee x = 2 \vee x = 1] \\ & \quad \text{then } [3,2,1] \\ & \quad \text{else } [\,]. \end{aligned}$$

The problem is that to evaluate the condition of the conditional we must be able to prove or disprove the validity of the formula in the scope of the universal quantifier.

One possible solution is to limit quantifiers to range over specific domains with antecedents selected from a predefined and well-studied set of formulas and use them as convenient notation for iteration or finite disjunctions or conjunctions.

12.3.2 Data structures

One of the reasons that programming in pure LISP or PROLOG is cumbersome at times is because the only available built-in data structure is the linear *list*. LISP has the feature that programs are also list structures, but this is not true in PROLOG. Some of the (nonlogical) built-in predicates in PROLOG are used to convert from the expression structure (terms and formulas) into list structure when we want to manipulate program clauses.

A data structure missing from PROLOG that has been added to LISP systems is the array. The ability to access in constant time any element of the array is the most important property of arrays which lists (and trees) lack. It is important to add arrays efficiently at the lowest implementation level but we should also include semantically clean programmer interface to access them. Two approaches suggesting how to add arrays to PROLOG are presented in [Cohen 84] and in [Eriksson and Rayner 84] (this last one was actually the contribution of Kahn in the PROLOG Digest). TABLOG does not improve on LISP and PROLOG in supporting more data structures an therefore can also benefit from such extensions.

Another data structure that should probably be added to TABLOG is the expression. An expression is either a term or a formula. A nice consequence of adding expressions as a basic data structure in TABLOG is that it will make TABLOG programs easier to manipulate within TABLOG; for example when writing a TABLOG interpreter in TABLOG. The problem with introducing expressions is that in the language of first-order logic we cannot nest predicates or connectives inside terms (except in the condition of a conditional term). For example,

$$\mathbf{eval}(\mathbf{p}(x,y) \wedge \mathbf{q}(x,z)) \equiv \mathbf{eval}(\mathbf{p}(x,y)) \wedge \mathbf{eval}(\mathbf{q}(x,z))$$

Is a formula that we might want to use in a program but is not a formula in first order logic.

This extension, like the previous one, will be useful only if efficient implementations will be built.

12.3.3 Types

Adding types (or sorts) to the language will make the language cleaner, in the sense that it will be easier to prove properties of programs and to detect bugs. Since the variables are local to the assertion (or goal) in which they appear, it seems that we have to declare them in every assertion. The solution is to declare the types globally for a specific program so if, for example, we declare u to be a variable of

type *list-of-atoms,* we can use it in different assertions to denote different actual variables, but they will all have the same type, namely *list-of-atoms.*

In addition to variables and constants, functions will also be typed for their arguments as well as their value. The default type for any symbol will be the ***universal* type.**

We can also have a hierarchy of types and have some automatic type inference, as in ML ([Milner 84]). For example, we can have the following subtype chain

$$\textit{natural} \subset \textit{integer} \subset \textit{number} \subset \textit{universal},$$

which will imply other relations among types, e.g.,

$$\textit{list-of-integers} \subset \textit{list-of-universals}.$$

Once the language is enriched by types, the unification algorithm should be extended to include type checking: a variable in an assertion will unify only with terms of the same type or a subtype. A variable in a goal can get instantiated to a term of the same type or a subtype. If, however, it is unified with a variable in an assertion the new goal will have a variable of the minimum of the two types. We can also unify variables in the assertion with terms in a goal of a super or unspecified type and add the conjunct to specify the type from the assertion.

For example, we can assert for integers

$$u > v \Leftarrow u \rightarrow v + 1$$

(which is not correct for the reals); this assertion can be used in solving the goal

$$x > y$$

only if x and y are of type *integer* or *natural* but not if they are of (the more general) type *number*. If we use the assertion in the latter case we should add a conjunct specifying that the new variables in the goal must be integers.

In principle, types should be optional in programs and a default *universal* type should be available for use in assertions that hold in general. Whenever no explicit type is specified or can be inferred the default universal type will be assumed.

Types can also be viewed as one place predicates. The hierarchy of types can then be expressed as implication relationship amongst those predicates. Examples can include

$$\textit{natural}(x) \rightarrow \textit{integer}(x)$$
$$[\textit{integer}(x) \vee \textit{real}(x)] \rightarrow \textit{number}(x)$$
$$\textit{sort}(x) \rightarrow \textit{universal}(x) \qquad \textbf{(for any } \textit{sort}\textbf{)}.$$

12.3.4 Controlling the backtracking

Another possible extension to TABLOG is a control sublanguage. In particular such a language will control the choice of backtracking and will fill a role similar to that of the *cut* in PROLOG, but in a cleaner way.

To give the user better control on the procedural interpretation of the program, we should supply some mechanism for preventing backtracking. There are few possible ways to do that:

1. Prevent backtracking over a conditional after the condition has been successfully proved.
2. Provide a special implication symbol (e.g. **=>**) that will be used as a guard, as in Dijkstra's guarded command.
3. Support backtracking only over deductions but not over rewriting. This actually means that we cannot backtrack over an application of the equality rule if the rest of the formula has been successfully proven.

Note that the proposals of items (1) and (3) actually imply that we evaluate the conditions of the conditional term or a conditional equation before actually manipulating the rest of the expression. This will also give us a nice synchronization mechanism for the parallel execution of TABLOG programs.

By adding a separate control sublanguage we can make the order of subformulas in a goal immaterial to the procedural interpretation unless explicitly specified to the contrary by the control language.

12.3.5 Associative operations

Earlier we have discussed the possibility of using quantifiers as iterative constructs; another useful high-level iterative construct can be found in associative operators with variable numbers of arguments. Adding such operators with parameterized arguments to TABLOG can be used not only for iteration in a sequential program but also to express parallelism. Such a usage of this construct will help remedy one of the problems with the parallel versions of logic-programming languages proposed so far, namely, that they can express only a constant factor of parallelism. For example, in CONCURRENT TABLOG we could use associative operators to define an array of processors. This approach is similar to the *insert* operator of FP ([Backus 78]), which allows applying an arbitrary (associative) operation to all elements of a list. Since TABLOG (and the proposed parallel versions) is restricted to first-order constructs, we cannot introduce higher-order operations like *insert*. The syntactic construct used to describe the insertion of associative operations like $\wedge$, $\vee$, $+$, and $*$ into an arbitrarily long list can be the standard $\wedge$, $\vee$, Σ, and Π. When dealing

with parallel or concurrent versions of TABLOG, the interpreter/compiler will be able to execute the insertion of the operation in logarithmic rather than linear time (provided enough processors are available). The arguments of the extended operators proposed here will be parameterized; this essentially means that each process invoked as a result of such a construct can actually know its very own index. We need this to enable the programmer specify the communication between the neighboring processes.

Note that the variables are shared within the assertion.

Example 12.1. *Factorial*
The CONCURRENT TABLOG or PARALLEL TABLOG program

$$\mathbf{factorial(n)} = \prod_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{i}$$

should produce (under a smart interpreter or compiler/scheduler) a tree of nodes for executing the multiplications in parallel (as much as possible). Of course, we need the associativity of the operation (multiplication in this case) for getting this parallel optimization.

12.4 Future implementations

The current experimental implementation can be improved in a few ways.

12.4.1 Efficiency

The most noticeable goal of future implementations is to build a faster interpreter or a compiler for the language. This can be achieved by using techniques that were developed for the implementation of PROLOG and ML compilers.

In the case of TABLOG, we also have the problem of the built-in simplification. The current simplifier is too general but also much too slow; recognizing the special cases of formulas that can occur in TABLOG computations and tuning the simplifier to handle only these cases, ignoring more sophisticated simplifications will be the first step in developing a leaner, faster simplifier.

12.4.2 Modularity

Currently all parts of a TABLOG program must be included in the same tableau. Future implementations should enable a program in one tableau to use the definition of functions loaded from files into the same tableau or other tableaux.

When a compiler exists, it will be desirable to let interpreted functions and predicates use pre-compiled auxiliary programs.

12.4.3 Completeness

As was discussed in Chapter 9 we can improve the completeness of the proof procedure by using connection graphs as the indexing mechanism (see [Stickel 82]) and adding goal-goal resolution. The problems associated with the goal-goal mechanism were also discussed in Chapter 9. Implementing the combination of connection graphs with goal-goal resolution only on inherited links should not cause special problems.

12.4.4 Concurrent implementations

The ideas discussed in Chapter 10 were neither tested nor implemented. These ideas should be further investigated and developed and can be fairly judged only after a collection of problems and concurrent programs to solve them is developed and such programs are tested on interpreters or compilers implementing some of these ideas. This suggests two efforts to be undertaken, the construction of a collection of test programs for parallel logic programming, and the implementation of a concurrent or parallel version of TABLOG.

12.4.5 Compiler

A compiler for TABLOG should use control information and eliminate some of the generality of the logical interpretation of TABLOG programs. On the other hand, such a compiler should be at least an order of magnitude faster than the interpreter by encoding predictable unification and deductions using low level constructs.

The compiler should be assisted by mode declarations and other appropriate control annotations. The addition of such constructs is especially important in cases where the optimization of a program cannot be deduced from the assertions of the program because they depend on the user's knowledge of the data domain and the problem one is trying to solve.

12.5 Other research directions

In addition to work in extending the language and creating better implementations, there are interesting research directions related to TABLOG that could be pursued.

Completeness: The proof procedure is not complete for the full language of first-order logic. It would be interesting to identify subset of the language for which TABLOG's proof system is complete. Also one can try to characterize an interesting set of models for a program.

In particular it will be interesting to find if there is any correspondence between the sentences that can be proved by the TABLOG interpreter and

those that can be proved by some special well studied proof-systems, e.g., intuitionistic logic.

Automatic derivation of programs: As we saw, not every formula in logic is a defining formula that can be considered an assertion in a TABLOG program. The language also lacks explicit quantifiers which are sometimes necessary for the specification of problems. More research should be done on deriving TABLOG programs from specifications in full first-order logic with no restrictions.

Proving program correctness and termination: **The methods developed for** proving properties of pure LISP programs are not directly applicable to TABLOG. Are there more powerful methods that can be used to reason about TABLOG programs and prove their correctness or termination?

12.6 Conclusions

TABLOG is a rich language that uses the familiar syntax of first-order logic. This syntax gives it benefits in expressive power over LISP and PROLOG.

The use of unification as the binding mechanism is the main advantage over LISP; unification gives us an easy way to have programs with multiple outputs and to arrive at more concise programs using the free decomposition of structured inputs.

The addition of real negation and equality are the main advantages over PROLOG, although equivalence and disjunction also add to the expressive power of the language, especially in programs that naturally involve logic (like puzzles). The equality and functional notation make programs easier to write and understand.

The ability to suspend subgoals or functional terms that need more evaluation before they can be reduced allows some programs that will cause errors on a PROLOG interpreter to run smoothly on the TABLOG interpreter.

The deductive engine that supports the rich language is very powerful, but the speed of execution can gain from new implementations that will compile programs.

The incompleteness of the proof procedure used implies that sometimes we must depend on the order of evaluation in TABLOG programs. On the other hand, knowing this order, we can guide the interpreter to use call-by-name evaluation when it is more efficient and to write tests that are expected to succeed first.

As this chapter has suggested, the TABLOG experiment is not a perfect one and more work has to be done to improve it and to understand better its promises and limitations.

CHAPTER 13

REFERENCES

[Apt and van Emden 82]

K. R. Apt and M. H. van Emden, "Contributions to the Theory of Logic Programming," Journal *of the ACM,* Vol. 29 (3), pages 841-862, July 1982. Also available as a Technical Report CS-80-13 from the University of Waterloo, Canada.

[Backus 78]

J. Backus, "Can Programming be Liberated from the von Neuman Style?" *Communications of the ACM* Vol. 21, (1978) pages 613-641.

[Balbin and Lecot 85]

I. Balbin and K. Lecot, *Logic Programming: a* Classified *Bibliography,* Wildgrass Books, Fitzroy, Victoria, Australia.

[Bowen 82]

K. A. Bowen, "Programming with full first-order logic," *Machine Intelligence 10,* J. E. Hayes, D. Michie, and Y-H Pao editors, Ellis Horwood Ltd., Chichester, 1982.

[Bronstein 83]

A. Bronstein, "Full quantification and special relations in a first-order logic theorem prover," unpublished report, Computer Science Department, Stanford University, 1983.

[Burstall, McQueen, and Sannella 80]

R. M. Burstall, D. B. McQueen, and D. T. Sannella, "HOPE: An experimental applicative language," University of Edinburgh, 1980.

[Cartwright and McCarthy 79]

R. Cartwright and J. McCarthy "Recursive Programs as Functions in a First Order Theory," in *Proceedings of the International Conference on Mathematical*

[Cartwright and McCarthy 79]
R. Cartwright and J. McCarthy "First order programming logic," in *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, pages 68–80, January 1979.

[Clark and Gregory 81]
K. L. Clark and S. Gregory, "A relational language for parallel programming" *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, October 1981.

[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.

[Clinger 82]
W. Clinger, "Nondeterministic call by need is neither lazy nor by name," in *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 226–234, Pittsburgh, Pennsylvania, August 1982.

[Clocksin and Mellish 81]
W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*, Springer-Verlag, 1981.

[Coelho, Cotta, and Pereira 80]
H. Coelho, J. C. Cotta, and L. M. Pereira, *How to Solve it with PROLOG*, 2nd. edition, Laboratório Nacional de Engenhari Civil, Lisbon 1980.

[Cohen 84]
S. Cohen, "Multi-version structures in PROLOG," in *Proceedings of the International Conference on Fifth-Generation Computer Systems*, 6–9 November 1984, Tokyo, Japan.

[Colmerauer, Kanoui, and van Caneghem 79]
A. Colmerauer, H. Kanoui, and M. van Caneghem, "Etude et réalisation d'un système PROLOG," Internal Report, Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, 1979.

[Conery 83]
J. S. Conery, *The And-Or Process Model for Parallel Interpretation of Logic Programs*, PhD Dissertation, University of California, Irvine, 1983.

[Conery and Kibler 81]
J. S. Conery and D. F. Kibler, "Parallel interpretation of logic programs," *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architectures.*

[Conery and Kibler 83]
J. S. Conery and D. F. Kibler, "AND parallelism in logic programs," *Proceedings of the International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, August 1983.

[Darlington, Field, and Pull 85]
J. Darlington, A. J. Field, and H. Pull, "The unification of functional and logic programming," in [DeGroot and Lindstrom 85].

[Darlington and Reeve 81]
J. Darlington and M. Reeve, "ALICE—A multi-processor reduction machine for the parallel evaluation of applicative languages," *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architectures.*

[DeGroot and Lindstrom 85]
D. DeGroot and G. Lindstrom (editors), *Logic Programming: Relations, Functions, and Equations*, Prentice-Hall, 1985.

[Dershowitz 84]
N. Dershowitz, "Equations as programming language," in *IEEE 1984 Proceedings of the Fourth Jerusalem Conference on Information Technology,* May 1984, Jerusalem, Israel.

[Dershowitz and Josephson 84]
N. Dershowitz and N. A. Josephson, "Logic programming by completion," in *Proceedings of the Second International Logic Programming Conference*, S-Å. Tärnlund (editor), pages 313–320, Uppsala, Sweden, July 1984.

[Dijkstra 76]
E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Dwork, Kanellakis, Mitchell]
C. Dwork, P. C. Kanellakis, and J. C. Mitchell, "On the sequential nature of unification," *Journal of Logic Programming*, Vol. 1, No. 1, 1984. Also available as Technical Report No. CS-83-26, Brown University.

[Eriksson and Rayner 84]
L.-H. Eriksson and M. Rayner, "Incorporating mutable arrays into logic programming," in *Proceedings of the Second International Logic Programming Conference*, S-Å. Tärnlund (editor), pages 101–114, Uppsala, Sweden, July 1984.

[Fribourg 84]
L. Fribourg, "Oriented equational clauses as a programming language," *Journal of Logic Programming*, Vol. 1, No. 2, 1984.

[Friedman and Wise 76]

D. P. Friedman and D. S. Wise, "CONS should not evaluate its arguments," in Automata, Languages, and Programming, Michaelson and Milner (editors), Edinburgh University Press, 1976, pages 257–284.

[Goguen and Meseguer 84]

J. Goguen and J. Meseguer, "Equality, types, modules and (why not?) generics for logic programming," *Journal of Logic Programming*, Vol. 1, No. 2, 1984. Also available as Technical Report CSLI-84-5, Center for the Study of Language and Information, Stanford University. Also in [DeGroot and Lindstrom 85].

[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,* PhD Thesis, Department of Telecommunication Systems and Computer Science, The Royal Institute of Technology, Stockholm, Sweden, 1981.

[Haridi and Ciepielewski 83]

S. Haridi and A. Ciepielewski, "An Or-Parallel Token Machine," Technical Report TRITA-CS-8303, The Royal Institute of Technology, Stockholm, Sweden, May 1983.

[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.

[Henderson and Morris 76]

P. Henderson and J. H. Morris, "A lazy evaluator," in *Proceedings of the third ACM Symposium on Principles of Programming Languages*, Atlanta, Georgia, pages 95–103, 1976.

[Hoffman and O'Donnell 82]

C. M. Hoffman and M. J. O'Donnell, "Programming with equations," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, pp. 83–112, January 1982.

[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 85]
W. Kornfeld, "Equality for PROLOG," in [DeGroot and Lindstrom 85].

[Kowalski 74]
R. Kowalski, "Predicate logic as a programming language," *Proceedings of the IFIP Congress*, pages 569–574, North-Holland (1974). Also DAI Research Report 74, University of Edinburgh.

[Kowalski 75]
R. Kowalski, "A proof procedure using connection graphs," *Journal of the ACM* Vol. 22 (4), October 1975, pages 572–595.

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

[Lindstrom 85]
G. Lindstrom, "Functional Programming and the logical variable," in *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 1985.

[Lloyd 84]
J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[Malachi 82]
Y. Malachi, "Deductive programming," unpublished report, Department of Computer Science, Stanford University.

[Malachi, Manna, and Waldinger 84]
Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG—the deductive-tableau programming language" in *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 323–330, Austin, Texas, August 1984.

[Malachi, Manna, and Waldinger 85]
Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG—a new approach to logic programming" in [DeGroot and Lindstrom 85].

[Manna 74]
Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

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

[Manna and Waldinger 85]

Z. Manna and R. Waldinger, The *Logical Basis for Computer Programming,* Volume 2: Deductive *Reasoning,* Addison-Wesley, 1985.

Volume 2: Deductive Systems, **to appear.**

[Manna and Waldinger 86]

Z. Manna and R. Waldinger, "Special relations in automated deduction," Journal *of the ACM,* January 1986.

[Martelli and Montanari 82]

A. Martelli and U. Montanari, "An efficient unification algorithm," *ACM Transactions* ***on Programming Languages and Systems,*** **Vol. 4, No. 2, pages 258-282,** April 1982.

[McCarthy 82]

J. McCarthy, "Coloring maps and the Kowalski doctrine," Technical Report STAN-CS-82-903, Department of Computer Science, Stanford University, April 1982.

[McCarthy and Talcott 80]

J. McCarthy and C. Talcott, *LISP Programming and Proving,* Stanford University, 1980.

[Milner 84]

R. Milner, "A proposal for standard ML," *Proceedings of the ACM Symposium* ***on Lisp and Functional Programming,*** **Austin, Texas, August 1984.**

[Murray 82]

N. V. Murray, "Completely nonclausal theorem proving," *Artificial Intelligence,* Vol. 18, No. 1, pages 67-85.

[O'Donnell 77]

M. J. O'Donnell, *Computing in Systems Described by Equations,* **Springer-**Verlag, New York, 1977.

[O'Keefe 85]

R. O'Keefe, "Alpine Club problem," *PROLOG Digest,* Volume 3, Issue 4, 11 February 1985.

[Paterson and Wegman 78]

M. S. Paterson, and M. N. Wegman, "Linear unification," *Journal of Computer and System Sciences,* Vol. 16, No. 2, pages 158-167, April 1978.

[Pereira and Warren 82]

L. M. Pereira and D. H. D. Warren, DEC*system-10 PROLOG User's Manual*, Department of Artificial Intelligence, University of Edinburgh, Scotland, November 1982.

[Reddy 85]

U. Reddy, "On the relationship between logic and functional languages," in [DeGroot and Lindstrom 85].

[Reddy 85b]

U. Reddy, "Narrowing as the operational semantics of functional languages," in *Proceedings of the 1985 Symposium on Logic Programming*, Boston, Massachusetts, July 15–18, 1985.

[Robinson 65]

J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, Vol. 12, No. 1, January 1965, pages 23–41.

[Robinson and Sibert 82]

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 editors, Ellis Horwood Ltd., Chichester, 1982.

[Robinson 85]

J. A. Robinson, "New-generation knowledge processing, Syracuse University Parallel Expression Reduction," Progress Report, Syracuse University, Syracuse, New York, December 1984. To appear in *Machine Intelligence 13*, Ellis Horwood Ltd., Chichester, 1985.

[Sato and Sakurai 84]

M. Sato and T. Sakurai, "Qute: A Functional language based on unification" in *Proceedings of the International Conference on Fifth-Generation Computer Systems*, 6–9 November 1984, Tokyo, Japan. Also in [DeGroot and Lindstrom 85].

[Shapiro 83a]

E. Y. Shapiro, "A subset of CONCURRENT PROLOG and its interpreter," Technical Report CS83-06, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, February 1983. Also available as: Technical Report TR-003, ICOT Institute for New Generation Computer Technology, Tokyo, Japan.

[Shapiro 83a]
E. Y. Shapiro, "Lecture notes on The Bagel—a systolic CONCURRENT PROLOG machine," Technical Report TM-0031, ICOT Institute for New Generation Computer Technology, Tokyo, Japan, November, 1983.

[Sterling and Bundy 82]
L. Sterling and A. Bundy, "Meta-level inference and program verification," in **Lecture Notes in Computer Science 138:** ***6th Conference on Automated Deduction,*** D. W. Loveland (editor), pages 144-150, Springer-Verlag, 1982. Also available as Research Paper 168, Department of Artificial Intelligence, University of Edinburgh, Scotland (December, 1981).

[Stickel 82]
M. E. Stickel, "A nonclausal connection-graph resolution theorem-proving program," **in** ***Proceedings of the National Conference on Artificial Intelligence,*** pages 229-233, Pittsburgh, Pennsylvania, August 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.

[Tärnlund 81]
S.-Å. Tärnlund, "A Programming Language Based on a Natural Deduction System," Technical Report, Computer Science Department, Uppsala University, 1981.

[Traugott 85]
J. Traugott, "Nested resolution," unpublished report, Computer Science Department, Stanford University, 1985.

[Voda 84]
P. J. Voda and B. Yu, "RF-Maple: a logic programming language with functions, **types, and concurrency," in** ***Proceedings of the International Conference on Fifth*** *Generation Computer Systems,* 6-9 November 1984, Tokyo, Japan.

[Warren 77]
D .H. D. Warren, "Implementing Prolog, Compiling Predicate Logic Programs," Research Reports 39 & 40, Dept of Artificial Intelligence, University of Edinburgh, 1977.

[Winston and Horn 81]
P. H. Winston and B. K. P. Horn, *LISP,* Addison-Wesley, 1981.

[Yellin 83]
F. Yellin, "PROLOG-based program synthesis," unpublished report, 1983.

APPENDIX A

TABLOG IN TABLOG

This appendix includes a TABLOG interpreter implemented in the TABLOG language. The purpose of this interpreter is two-fold: to demonstrate the expressive power of the language and to give the reader of this document and the TABLOG user a better feeling for the way the real sequential interpreter works.

The following TABLOG program is an approximation of the real TABLOG interpreter. The real interpreter is too low-level to be translated into TABLOG and included here. Unlike PROLOG or LISP, TABLOG distinguishes between predicate and function symbols. Since all the statements in a TABLOG program must be formulas in logic, we cannot nest predicates within each other. The interpreter included here uses specially defined symbols to represent the programs to be interpreted. Using these symbols we represent terms and formulas alike as terms. As with the unification program given in Chapter 5, this program depends on the order of evaluation of the TABLOG interpreter.

The special predicate assert is used to state the assertions of the interpreted program, while **eval** and **evalterm** are functions used to evaluate formulas and terms, respectively. The functions -1-1, AA, VV, <—<—, ≡= and cond are used to include the standard connectives in program assertions. Truth is represented by yes and falsity by no. In a program given to this interpreter however we write an assertion with no conditions as an implication with the antecedent *t* which evaluates to yes.

1. **eval(i)** = yes.

   True is represented by *t* in programs and should always be evaluated to yes.

The next two assertions are for evaluating the equality predicate. The terms on both sides of the equality are each evaluated by **evalterm** and the results are compared.

2. **eval(x** == *y)* = *if* **evalterm(x)** = **evalterm(y)** *then* yes *else* no.
3. **eval(-r-i(a:** == *y))* = *if* **evalterm(s)** = **evalterm(y)** *then* no *else* yes.

Each of the next groups of assertions evaluates one of the propositional connec-

each group is for the failure case when none of the other assertions can be successfully used. Of course this makes the program order dependent.

4. $\mathbf{eval}(p \vee\!\vee q) = \text{yes} \leftarrow \mathbf{eval}(p) = \text{yes} \vee \mathbf{eval}(q) = \text{yes}.$
5. $\mathbf{eval}(p \vee\!\vee q) = \text{no} \leftarrow \mathbf{eval}(p) = \text{no} \wedge \mathbf{eval}(q) = \text{no}.$
6. $\mathbf{eval}(p \vee\!\vee q) = \text{fail}.$

7. $\mathbf{eval}(p \wedge\!\wedge q) = \text{yes} \leftarrow \mathbf{eval}(p) = \text{yes} \wedge \mathbf{eval}(q) = \text{yes}.$
8. $\mathbf{eval}(p \wedge\!\wedge q) = \text{no} \leftarrow \mathbf{eval}(p) = \text{no} \vee \mathbf{eval}(q) = \text{no}.$
9. $\mathbf{eval}(p \wedge\!\wedge q) = \text{fail}.$

10. $\mathbf{eval}(p \rightarrow\!\rightarrow q) = \text{yes} \leftarrow \mathbf{eval}(p) = \text{no} \vee \mathbf{eval}(q) = \text{yes}.$
11. $\mathbf{eval}(p \rightarrow\!\rightarrow q) = \text{no} \leftarrow \mathbf{eval}(p) = \text{yes} \wedge \mathbf{eval}(q) = \text{no}.$
12. $\mathbf{eval}(p \rightarrow\!\rightarrow q) = \text{fail}.$

13. $\mathbf{eval}(\mathbf{cond}(p,q,r)) = \mathbf{eval}(q) \leftarrow \mathbf{eval}(p) = \text{yes}.$
14. $\mathbf{eval}(\mathbf{cond}(p,q,r)) = \mathbf{eval}(r) \leftarrow \mathbf{eval}(p) = \text{no}.$
15. $\mathbf{eval}(\mathbf{cond}(p,q,q)) = \mathbf{eval}(p).$
16. $\mathbf{eval}(\mathbf{cond}(p,q,r)) = \text{fail}.$

17. $\mathbf{eval}(\neg\neg(p \vee\!\vee q)) = \mathbf{eval}(\neg\neg p \wedge\!\wedge \neg\neg q).$
18. $\mathbf{eval}(\neg\neg(p \wedge\!\wedge q)) = \mathbf{eval}(\neg\neg p \vee\!\vee \neg\neg q).$
19. $\mathbf{eval}(\neg\neg(p \rightarrow\!\rightarrow q)) = \mathbf{eval}(p \wedge\!\wedge \neg\neg q).$
20. $\mathbf{eval}(\neg\neg\mathbf{cond}(p,q,r)) = \mathbf{eval}(\mathbf{cond}(p, \neg\neg q, \neg\neg r)).$
21. $\mathbf{eval}(\neg\neg p) = \mathbf{eval}(\neg\neg q) \leftarrow \mathbf{assert}(p \equiv\!\equiv q).$
22. $\mathbf{eval}(p) = \mathbf{eval}(q) \leftarrow \mathbf{assert}(p \equiv\!\equiv q).$
23. $\mathbf{eval}(p) = \text{yes} \leftarrow \mathbf{assert}(p \leftarrow\!\leftarrow q) \wedge \mathbf{eval}(q) = \text{yes}.$
24. $\mathbf{eval}(p) = \text{no} \leftarrow \mathbf{assert}(\neg\neg p \leftarrow\!\leftarrow q) \wedge \mathbf{eval}(q) = \text{yes}.$
25. $\mathbf{eval}(p) = \text{yes} \leftarrow \mathbf{assert}(\mathbf{cond}(q,p,r)) \wedge \mathbf{eval}(q) = \text{yes}.$
26. $\mathbf{eval}(p) = \text{no} \leftarrow \mathbf{assert}(\mathbf{cond}(q, \neg\neg p, r)) \wedge \mathbf{eval}(q) = \text{yes}.$
27. $\mathbf{eval}(p) = \text{yes} \leftarrow \mathbf{assert}(\mathbf{cond}(q,w,p)) \wedge \mathbf{eval}(q) = \text{no}.$
28. $\mathbf{eval}(p) = \text{no} \leftarrow \mathbf{assert}(\mathbf{cond}(q,w, \neg\neg p)) \wedge \mathbf{eval}(q) = \text{no}.$
29. $\mathbf{eval}(p) = \text{fail}.$

The next part of the interpreter defines the evaluation of terms in a recursive way.

30. $\mathbf{evalterm}(u) = u \leftarrow \mathbf{atomp}(u).$
31. $\mathbf{evalterm}(u_1 \circ u_2) = \mathbf{evalterm}(u_1) \circ \mathbf{evalterm}(u_2).$

32. $\mathbf{evalterm}(u) = \mathbf{evalterm}(v_1) \leftarrow \mathbf{assert}(u == v_1 \leftarrow\!\!\leftarrow w) \wedge \mathbf{eval}(w) = \text{yes}.$
33. $\mathbf{evalterm}(\mathbf{cond}(p, u, v)) = \mathbf{evalterm}(u) \leftarrow \mathbf{eval}(p) = \text{yes}.$
34. $\mathbf{evalterm}(\mathbf{cond}(p, u, v)) = \mathbf{evalterm}(v) \leftarrow \mathbf{eval}(p) = \text{no}.$
35. $\mathbf{evalterm}(u) = u.$

And here a version of the quicksort program coded for this interpreter.

36. $\mathbf{assert}(\mathbf{quicksort}(u) == \mathbf{qsort}(u, [\,]) \leftarrow\!\!\leftarrow t).$
37. $\mathbf{assert}(\mathbf{qsort}(x \circ u, r) == \mathbf{qsort}(sm, x \circ \mathbf{qsort}(lr, r)) \leftarrow\!\!\leftarrow \mathbf{partition}(u, x, sm, l$
38. $\mathbf{assert}(\mathbf{qsort}([\,], r) == r \leftarrow\!\!\leftarrow t).$
39. $\mathbf{assert}(\mathbf{partition}(x \circ u, y, sm, x \circ lr) \leftarrow\!\!\leftarrow \mathbf{partition}(u, y, sm, lr)) \leftarrow y < x.$
40. $\mathbf{assert}(\mathbf{partition}(x \circ u, y, x \circ sm, lr) \leftarrow\!\!\leftarrow \mathbf{partition}(u, y, sm, lr)) \leftarrow y \geq x.$
41. $\mathbf{assert}(\mathbf{partition}([\,], u, [\,], [\,]) \leftarrow\!\!\leftarrow t).$

This program will be called with a goal like

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

and will bind z to the sorted list $[1, 2, 3]$. Note that there is a some cheating in this quicksort example: the tests $y < x$ and $y \geq x$ are not written inside the **assert**; this causes their evaluation directly by the real TABLOG interpreter and not the one demonstrated here. This short-cut is taken to make the example shorter.

APPENDIX B

HOW TO RUN TABLOG

This appendix describes how to use the TABLOG interpreter currently available. The system is implemented in MACLISP and it runs only on the SAIL (Stanford Artificial Intelligence Laboratory) computer system under the WAITS operating system.

Section 1 of this appendix gives general information about the use of the interpreter and in particular about the modes of communication between the user and the system. The following section describes the details of syntax of commands and the programs that are acceptable by the interpreter and how to declare new objects in the language. Later sections overview other commands that can be issued to the system, including reading and writing of files and the manipulation of program tableaux.

Since this appendix is written as a user manual second person is used.

B.1 General information

The first thing to do is to type to the SAIL monitor

```
r tablog
```

this will invoke the TABLOG interpreter.

B.1.1 Modes

The interpreter can be in one of four modes:

Command-mode: in this mode the interpreter expects a TABLOG command. This mode is recognized by the prompt message

```
tablog>.
```

The various commands will be described later. Most commands have their arguments on the same line as the command but will prompt the you for missing arguments.

If the line starts with a non-atomic S-expression, the TABLOG interpreter will pass it to the LISP interpreter. This is merely an aid in debugging and not part of the TABLOG language.

Formula-mode: In this mode, the interpreter expects a formula in logic to be entered as an entry (assertion or goal). The identifying message for this mode is

».

Type a logic formula using the syntax described below. Remember however, that the formula parser expects a period at the end of each formula; this allows you to enter multi-line assertions or goals.

Stepping-mode: This mode is used when debugging a TABLOG program. It is recognizable by the herald

STEP>

In most cases you will just type a positive integer to specify the number of steps to be executed before the next stop. Other options in this mode will be described later.

Lisp-mode: Sometimes (because of error and, hopefully, rarely) you will find yourself in a bare LISP read-eval-print loop. You can recognize this mode by the prompt character

•-

of the LISP editor or the absence of any prompting message. Usually you will want to go back to *command-mode*, which you can do by typing

tablog

If this does not help, try repeating it after typing ~G (i.e., control-G) and getting the «- prompt.

To simplify usage, the system recognizes any unambiguous abbreviation of the commands. When a command is typed without the expected arguments, the system will prompt you for the specific missing arguments.

In the examples (as in the actual output of the system), lower case letters represent individual constant symbols while upper case letters represent function symbols or individual variables. The input is, however, case insensitive.

You have to define the syntactic role of the symbols of the formal language used in the program. Although the system prompts you for declarations at the beginning of a new program, constant, variable, function, and predicate symbols can be also declared after the program is entered.

Section 2 includes the list of all the built-in declarations, which let the system know about basic constants, functions, and predicates. Variables as well as predicates and functions not built-in must be declared explicitly. Undeclared objects default to functions or constants according to the way they are used in the input.

As already mentioned, the scope of variable names is the entry or rule in which it appears; therefore the same name denotes two different variables in different entries. When applying the inference rules, the interpreter will rename variables to resolve any possible name collisions.

B.1.2 A sample session with Tablog

We start with the almost classical example of the **qsort** function. Here is the log of a session dialog:

```
Tablog> new qsort
Declarations (Variables, Constants, Functions, Predicates, CR to end)
declare (CR to end):  >> var u u1 u2 v x y z
declare (CR to end):  >> pred (partition)
declare (CR to end):  >> func (append qsort)
declare (CR to end):  >>
Enter program assertions.  Finish with "END".
>> partition(x,u,u1,u2) ↄ qsort(x⊛u)=append(qsort(u1),x⊛qsort(u2)).
A1 PARTITION(X,U,U1,U2) ↄ (QSORT(X⊛U) = APPEND(QSORT(U1),X⊛QSORT(U2)))
>> y≤x ∧ partition(x,u,u1,u2) ↄ partition(x,y⊛u,y⊛u1,u2).
A2 ((Y ≤ X) ∧ PARTITION(X,U,U1,U2)) ↄ PARTITION(X,(Y⊛U),(Y⊛U1),U2)
>> y>x ∧ partition(x,u,u1,u2) ↄ partition(x,y⊛u,u1,y⊛u2).
A3 ((Y > X) ∧ PARTITION(X,U,U1,U2)) ↄ PARTITION(X,(Y⊛U),U1,(Y⊛U2))
>> partition(x,[],[],[]).
A4 PARTITION(X,[],[],[])
>> qsort([])=[].
A5 QSORT([]) = []
>> append([],v)=v.
A6 APPEND([],V) = V
>> append(x⊛u,v)=x⊛append(u,v).
A7 APPEND((X⊛U),V) = (X⊛APPEND(U,V))
>> end
Goal >> z=qsort([2,1,4,3]).
G8 [None]                Z = QSORT([2,1,4,3])                    Z
```

After an assertion or a goal is entered into the tableau it is echoed onto the terminal screen.

Because '⊛' (which represents the insertion operator) is predefined, the parser did not complain that it was not declared; the conversion between the standard representation of lists using the `[ , , ]` construct and the decomposition-oriented representation using the ⊛ operator is done on input and output whenever appropriate.

Once a program and a goal are introduced to the system, the interpreter can execute the goal as a call to the program.

B.2 Declaring objects

In the example we saw how to declare objects when entering a new program; the `DECLARE` command can be used to introduce new symbols into the formal language of the proof system before or after a new program is entered. The declaration command has two forms:

`DEClare {Variable, CONStant, PROposition} <objects>`

`DEClare {Function, PREdicate} <objects> <options>`

When in *declaration mode* (as in last section's example) the system expects to get the arguments of a `declare` command which is exactly the rest of the line that would otherwise be typed with this command.

The syntactic objects that can be declared are:

`CONStant` – an individual constant.

`Variable` – an individual variable.

`PROposition` – a proposition (or logical constant). Initially, *true* and *false* are the only symbols in this category.

`PREdicate` – a predicate symbol.

`Function` – a function symbol.

The `<options>` field specifies some additional properties of the symbols being declared.

`<objects>` represents either one symbol or a list (in parentheses) of symbols to be declared; the declaration is applied to all the symbols in the list. When the first form of `<declare>` is used, `<objects>` does not have to be parenthesized even if it contains more than one object name.

For functions, `<options>` may contain any of the following in any order:

`Binary`, `INfix`, `PRefix`, `POstfix` or `Unary` – specifying the position of the operator in respect to its operands. `Binary` and `infix` are aliases; unary and postfix (e.g., $x!$) do not require parentheses around the arguments while `prefix` does.

`Nullary` – no argument operator; a nullary function is defined to be a constant.

`Left`, `Right`, or `Associative` – associativity property of the operator.

`<precedence>` – a natural number in the range 100 to 900 that specifies the relative binding power of the operator within its category. Examples are given below.

`Commutative` – to specify that the operator is commutative.

`IDempotent` – to specify that the operator is idempotent.

When declaring a predicate symbol there are two possible arguments: a position specification in the same form as described above for functions, and a commutativity (or symmetry) specification using the argument `commutative` (or `symmetric`), or any prefix of them.

The default values are:

position – `PREFIX`

associativity – `LEFT` for `binary` and `postfix`
`RIGHT` for `unary`
none for `prefix`

precedence – 900 for `unary` and `postfix`
400 for `binary`

For the special properties like idempotence the default is that the property does not hold.

Any element in `<options>` that does not match any of the options for the syntactic category being defined is ignored.

Examples:

```
declare cons (a b c d)
declare var r s t u v w
decl func !  postfix 600
dec pred (≥ ≤ < >) 400 binary
dec pred = symmetric binary 400
```

The first and second examples declare some new constants and variables respectively.

The later declarations introduce the equality and inequality predicates.

The relative precedence values are effective only within the syntactic category.

The properties assigned to the predefined objects of the language are equivalent to the following list of declarations. Note that these declarations also include declarations of connectives, which is not supposed to be done by the TABLOG programmer because the simplifier has to know about connectives while declarations only teach the parser about them.

```
declare connective (∨ or)  binary assoc 700 (1) commut idempotent
declare conn (∧ and &) bin assoc 800 (1) com idem
declare conn (¬ ~ not) unary right 900 (-1)
declare conn (≡ iff) bin left 600 (0) commut
declare conn (⊃ → implies) binary left 500 (-1 1)
declare conn (← :-) binary left 500 (-1 1)
declare function (^) binary right 800
declare funct * binary assoc 700 commut
declare func (// div) bin left 700
declare function + bin assoc 600 commut
declare fun - bin left 600
decl function (⊗ ⓪) binary right 300
decl function (min max) prefix
dec pred (> < ≥ >= ≤ =< ∈ in odd even) binary
dec pred (= ≠ /= \=) binary symmetric
dec proposition (true false)
```

The built-in syntactic objects initially known to the system includes the logical connectives as well as basic primitive functions and predicates.

This includes

- The truth values: `true, false`.
- The connectives: `∧, ∨, ¬, ≡, →, ←, if_then_else`.
  (Alternative notations for the syntactic symbols mentioned here are described below).
- The constant `[]`, and all the integers as constants.
- The predicates `=, ≥, ≤, >, <, even, odd`.

- The functions `min, max, +, -, *, /, ^, ⊛.`

  The last two stand for exponentiation and list insertion, respectively.

Alternative notations for predefined objects:

disjunctions: `∨ or`
negation: `¬ not ~`
conjunction: `∧ & and`
implication: `⊃ implies →`
reverse implication: `← :-`
equivalence: `≡ iff`
unequality: `≠ \=`
bigger-than: `≥ >=`
less-than: `≤ =<`
list insertion: `⊛ @`
universal quantification: `\forall forall`
existential quantification: `\exists forsome exist.`

B.3 Program tableaux

A program is stored in a deductive tableau and can define more than one function or predicate.

The system enables you to have several tableaux concurrently. Each tableau has its own name, which does not have to be the same as the name of the programs it defines. There is always one current tableau to which the various operations are applied. The command `NEW` starts a new tableau and makes it the current one while the command `SWITCH` makes its argument the current tableau.

For example, the command

```
NEw foo
```

will start the tableau `FOO` after checking if it already exists. If this tableau already exists you will be prompted for more input and will be able to choose between overwriting the existing tableau or to supplying another name for the new tableau to be initialized. When starting a new tableau, you will be given the opportunity to declare new objects in the language.

To select another tableau, say `BAR`, type:

```
SWitch bar
```

If the tableau with this name does not exist, it will be created as if the command `New bar` were given. All the information about the current tableau is stored before switching.

Any part of the current tableau can be viewed by using the `Print` command, in one of the forms:

```
Print Tableau
Print Names
Print Entry <range>
Print Dependency <range>
```

`Tableau` will cause the whole (current) tableau to be shown. `Entry` lets the specified range of the tableau entries to be viewed. `Dependency` refers to the list of dependencies among entries in the tableau. `Names` is used to see what are the names of the existing tableaux.

`Show` is an alias for `print`.

B.3.1 Entries: goals and assertions

Each tableau entry (assertion or goal) is a well-formed formula. To view any entry, the command `print` described above should be used. New assertions are introduced into the system as part of the program in the formula-mode

`>> <wff>.`

The `<wff>` expression is the well-formed formula of the entry to be added; for a goal there is an optional output value. The well-formed formula is entered in standard predicate logic as described below. This is demonstrated in section B.1.2; the programs in Chapter 5 can also be used as examples.

Goals (for new calls for the program can be entered using the `goal` command. When in command-mode the line

```
goal u=qsort([1,2,3]).
```

will enter the new goal and make it the current goal (note the period at the end of the goal; the `goal` command moves TABLOG into formula-mode).

Entries can be deleted by the command `DELETE`, which takes as an argument the object to be deleted. The system renumbers the entries after such a deletion and checks and updates the dependencies among the deleted entries and the remaining entries. If something depends on the deleted entries, the system will warn you before actually going ahead and deleting. For example,

```
del ent 6 last
```

will delete all the entries after entry 5.

B.4 Running programs

After you have defined a program you probably want to run it.

The command to run a program is

EXEC

which will take the current goal and reduce new goals until the goal true is reached, or no more deduction steps can be applied.

A program can be executed in a free-run or single-step mode. The command STEP (or SET STEP) enters single-step mode and the command NO STEP sets the free-run mode (which is the default mode).

The VERBOSE and TERSE commands change the output mode of the system. When running in verbose mode, every goal will be printed as it is generated, while in terse mode only the last goal (hopefully true, or a formula that cannot be further reduced by the system) will be printed.

When running a program in single-step mode, you get into the step-loop after successful reductions. In this mode you will get the system message:

STEP>

You can type quit to abort the execution; typing a positive number N will cause the next stop to be after N successful reductions. Typing anything else will be passed to the underlying MACLISP interpreter.

When the program terminates with the success goal **true** and prints out the binding of the output variables the you will have the option of rejecting this solution. The command

FAIL

which is aliased

RETRY

will force backtracking, causing the interpreter to look for further solutions.

B.5 Reading and writing files

The interpreter is basically an interactive system but it enables you to read and write files containing TABLOG programs.

To read a program from a file use the command read,

READ <name> [<ext>]

Where <name> is the file name and <ext> is an optional extension (note that no period is used!); if no <ext> is given and if the file <name> does not exist the file <name> .TPL will be loaded (if found).

A file loaded in this form should contain the commands that you would otherwise type yourself. Note that the read command can be used recursively from within files.

To record the session (your input and the system's output) into a file use the command

WRITE <name> [<ext>]

The record becomes permanent only after you issue the CLOSE command to close the output file. Note that QUIT does NOT close the file.

ECHO is a command that will write to a file only your input so you can load the file again next time.

B.6 Summary of commands

Here is a summary of the commands (typing the upper case letters part suffices)

Assertion — add a new assertion to the tableau.

CLose — close output file(s).

COMment — ignore this input; it's a comment. ; can be also used

DEClare {Variable, Constant, PREdicate, Function, PROposition} — declare a symbol to have some role in the formal language.

DELete {Assertion, Entry, Goal, Tableau} — delete an entry or a whole tableau from the system. The system updates the numbering of entries and the dependence relations among them.

DELete {Constant, Variable, Function, PREdicate, PROposition} — cancel the meaning of the argument as a special symbol of the language.

ECho — echo the user input to a file for future use.

EXEC — execute the current goal.

EXIt (or Quit) — exit from the program (after confirmation).

EVal — pass the rest of the input line to the Lisp evaluator.

Help (or ?) — print summary of commands.

Goal — add a new goal to the tableau. Make it the current goal

Write — record a log of the session in the file specified in the arguments

Print (or SHow) {All, Entry, Tableau, Property, Dependency, Names} — display to the user some information.

REAd — read commands from a file.

TErse — say less.

STep — get into single step mode.

SWitch — switch to another tableau.

TOp — quit to Maclisp top level.

Verbose — say more.