TLOG -- YET ANOTHER LOGIC SYSTEM IN LISP?

Walter Read
Michael G. Dyer

April 1986
CSD-860070

TLOG -- Yet Another Logic System in Lisp?

Walter Read
California State University, Fresno
UCLA Artificial Intelligence Laboratory

Michael G. Dyer
UCLA Artificial Intelligence Laboratory

Abstract

This paper describes TLOG, an implementation of logic programming in T, an object-oriented, lexically-scoped Lisp in the Scheme family. TLOG is designed to support AI research in the Artificial Intelligence Laboratory at UCLA. It is, as far as we know, the only object-based logic programming system implemented directly via objects. This object-orientation allows the development of useful experimental features, including trace/debug and compilation techniques for logic programs.

TLOG -- Yet Another Logic System in Lisp?

Walter Read
California State University, Fresno
UCLA Artificial Intelligence Laboratory

Michael G. Dyer
UCLA Artificial Intelligence Laboratory

1. Introduction

At the UCLA AI lab, like other labs, we have felt the need for a logic programming facility. Our environment currently consists of two dozen Apollo workstations running our internally developed tools environment called GATE (Graphical AI Tools Environment) [Mueller & Zernik 1984] a graphics and demon-based [Dyer 1983] facility built on top of T [Rees & Adams 1982]. T is a member of the Scheme family of lexically scoped Lisps [Abelson and Sussman, 1984], but also has a number of features that go beyond Scheme, including explicit objects constructs with message-passing and also named environments [Rees, Adams & Meehan 1983].

1.1. Goals

We want our logic programming system to be fully embedded in T so that T functions, demons and our other entities can freely be defined in terms of, and have access to, one another. This requires that (1) the database be represented and manipulated in T, (2) the theorem prover be run from T, and (3) that T code be called from logic clauses.

TLOG is intended to be a superset of Prolog were the user can experiment with alternate interpretation disciplines. However, TLOG also currently provides a standard Prolog syntax and semantics. The main goals of the TLOG project are: (1) to provide a useful and attractive logic programming system for AI researchers, (2) to explore the potentials of T as a language for implementing logic programming, and (3) to serve as a testbed for experiments in logic programming. The initial objectives were to lay the foundations for future development and to provide a full-featured logic programming system, one that would give the user much of the flexibility and support provided by the environment already in use. This required good trace/debug and edit facilities. TLOG has now completed its first stage of development and is currently in use in the lab, running on Apollos and also on Vaxen under Unix.

1.2. Some T Background

There are several extensions to SCHEME in T. The extensions which are used most heavily in the implementation of TLOG are **OBJECTs** and **SETTABLE** access methods.

T provides a convenient syntax for declaring **OBJECTs** (i.e. **LAMBDA** closures). The way a user makes a new **OBJECT** type in T is demonstrated by the following example:

```
(DEFINE (MAKE-RECTANGLE length width)
   (OBJECT ()
     ((GET-WIDTH self) width)
     ((GET-LENGTH self) length)
     ((GET-AREA self) (* width length))
     ((PRINT self stream)
      (PRINT (LIST 'RECTANGLE a1 a2) stream))))
```

where `(OBJECT ...)` introduces an object definition. Each of the method clauses, `((method-name . args) code)`, defines a method on the object, which is executed for example by executing the operation `(GET-AREA rectangle)`. Key system functions, like **PRINT**, are also implemented as operations so that users can write object definitions which integrate well with the rest of the T environment.

Users can also write accessor methods which are **SETTABLE**. For example, one can can say `(SET (GET-WIDTH rectangle) new-width)` if the object is defined as below:

```
(DEFINE (MAKE-RECTANGLE length width)
   (OBJECT ()
     ((GET-WIDTH self) width)
     (((SETTER GET-WIDTH) self value)
      (SET length (* (/ length width) value))
```

```
        (SET width value)) ...)
```

This defines a **SETTABLE** accessor method for **'width'** which preserves the aspect ratio of the rectangle.

T also provides named environments. This makes it easy to access bindings in other environments while at the same time maintaining a proper discipline of procedure and data abstraction.

2. Basics of the Implementation

a. Top Level Embedding of TLOG in T: When TLOG is loaded at the beginning of a session, a special environment is created and all TLOG functions are loaded into that environment. Only those functions needed are made available to (i.e. exported to) the user's own environment. The T read-eval-print loop READ and EVAL are then redefined so that T can include TLOG assertions or queries as part of the function definition. The **TLOG-READ** peeks at the first character of the input to determine whether Prolog or T syntax follows. In either case the output of **TLOG-READ** is conventional T code. The **TLOG-EVAL** then decides, based on the function name, whether to interpret the expression as a TLOG clause or as ordinary T code. This allows TLOG to be fully embedded in T and permits the user to intermix T and TLOG code.

b. Clauses: TLOG clauses are represented as T objects, with a variety of methods. In particular, the head and body of each clause are stored under settable access functions. Actually, there are two copies each of the head and body. One is generally left alone except while editing. The other is the "renamed" version used by the unifier. The clauses themselves are indexed fully by clause head within a discrimination net (d-net). The clause object also keeps a number used by the d-net to keep clauses in the order specified by the programmer for when standard Prolog interpretation is desired. The clause object also maintains a flag for deletion used by the discrimination net **FETCH** to note what clauses are still considered "in" the database. When the clause is loaded into the database the set of variables in the clause is computed for use by the "rename-variables" function.

c. The database: In TLOG the database is organized as a discrimination net with discrimination on the head of the clause. In order to keep the fetching efficient, whenever the number of sons passes a cutoff number, the discrimination net creates a hash table for that set of sons. The d-net also keeps track of the number of sons at each node.

Each clause is given a precedence number when it is added to the database. This allows the fetch to return clauses in the order in which they were entered into the database. This also allows TLOG to implement **ASSERTA** even though d-net organization is not by order of loading.

The decision to implement a full d-net with dynamic hashing, precedence ordering, and other structural information was based on three goals:

(1) Use the d-net to support unification.

The d-net already provides some support by filtering out many impossible unifications that would be considered in other Prolog systems. There are plans to better the cooperation between discrimination net and unifier: Optimally, either the unifier should make use of bindings produced by the result of a **FETCH**, or unification should be so integrated with indexing that unification is already completed by the time **FETCH** is done.

(2) Use the d-net to direct literal selection within a clause or clause selection within a procedure.

In Prolog, an 'incorrect' ordering of goals can result in very inefficient execution. By adding information at each node (nodes themselves are implemented as objects) it becomes possible to dynamically select the goal requiring the least amount of potential backtracks.

(3) Use the d-net to make complex structures an attractive option.

Most Prologs index on the predicate and perhaps the first argument. This seems somewhat ad hoc. For instance, imagine implementing one or more a logic interpreters in TLOG. This could result in many clauses each with the predicate **CLAUSE**. In systems that access the database by indexing on the predicate name efficient database access is defeated. As a more complicated example, suppose the database information is a story represented as many instantiated and interconnected frames [Dyer 1983]. If we need to find all occurences of something said to Mary, then the discrimination net will filter out all but the possible clauses. Simple indexing will only find all clauses that are indexed under the concept (predicate) representing "saying" and the unifier will have to check each one. The advice that such frame structures be represented as a series of binary relations [Kowalski 1979; Nilsson 1980] seems to require a very large database and an unnecessarily complicated search. Either we have to look for every clause that involves "saying" and then see if the object was Mary or we have to look at every clause involving Mary as a target and then see if the action was "saying". What complex structures within clauses allow us to do is to specify the most relevant indices for accessing associated relations. The practice of mapping to simple binary relations appears to be influenced as much by the limitations inherent in most Prolog indexing schemes, as by purely theoretical considerations.

d. Renaming variables during unification: TLOG renames variables for unification by getting the set of variables from the clause object, generating new names and mapping the resulting bindings into the clause object. The more common method of searching a clause and doing renaming requires that the program check at each stage whether the variable currently under consideration has already been renamed. By using the fact that a variable set has already been generated, the renaming can be done once and, when the clause is searched later, only the substitutions need be done. This efficiency in producing variant closures is important in the efficiency of the overall system, ~~as~~ variants have to be produced at each unification [Kahn and Carlsson 1984, p. 119].

3. Some Comments on Compiling Logic Programs

There is no general agreement on what "compiling" a logic program should be, except that it should result in the program running faster, possibly at a cost of time while loading. The common approach is to translate clauses or predicates into a procedural language. This may be either an existing language or an abstract machine language [Warren 1977].

If we want to consider compilation within the paradigm of logic programming, we must consider where to improve the performance of common functions in the interpretations of logic. Two of the most important places where improvement can be made are unification and renaming variables. Speeding up renaming depends partly on the representation used for variables but TLOG's way of handling renaming shows a general method that should improve performance no matter how variables are represented.

TLOG also supports unification by using the discrimination net. The net "compiles out" many non-unifications that would otherwise have to be checked by the unifier. Indexing clauses, as is done in many Prologs, achieves some of the same effect. In either case, structuring the database contributes to compilation.

4. "Society of Experts" Programming

The "Society of Experts" approach refers to a general model of cooperative problem solving in which different parts of the overall system are seen as "experts", each knowing how to solve a piece of the problem and each able to communicate questions and answers to other parts of the system. As such it is closely related to the idea of procedure abstraction, in that details of an individual expert's reasoning are hidden from other experts. This type of factoring requires that other experts need know only the form of the interface to their 'colleagues'.

TLOG currently supports this model of programming through T's named environments and a built-in TLOG predicate called **REQUEST**. The database corresponding to a given expert's knowledge can be created within an environment named for that expert. If, in the course of answering a query, another expert needs information from that expert, it can issue a **REQUEST**, which takes as arguments a query and a named environment. TLOG then runs the query in the named environment and passes the answer back to the calling environment as the value of the predicate (i.e. as either a binding list or the **'FAIL** signal).

As a simple example, we constructed a student database, whose knowledge is of the form "if you are asked a physics question, then ask a physicist". Clauses in this database would just call **REQUEST** referring to another (i.e. the physicist) database. The physicist database hase clauses describing solution methods for such queries. In our example, the physicist database also knows how to do elementary arithmetic, but to take square roots it needs a calculator. Thus some of the clauses in this database have a **REQUEST** to a calculator database. The user queries the student, who may then query either the physicist or the calculator, and so on. The system makes the necessary connections and the answer is returned in the original environment.

If the calling expert needs an explanation, either of a success or a failure, it can call the advanced debugging features of TLOG within the consulted expert's environment. In this way the called expert can either show its reasoning in case of success or show the failure points in case of failure.

This can be done somewhat awkwardly in Prolog by, for example, representing clauses in a given database, say `'db1'`, as `'db1(clause)'`. But this fails to give the abstraction provided by TLOG and may in large part defeat the indexing commonly used in Prolog by forcing each clause to have a new predicate. Other approaches involve building an entire object-oriented semantics on top of Prolog [Kahn 1982].

In TLOG distributed theorem-proving is almost trivially implemented by having **REQUEST** call a simple 3-line-long T function. This is because T allows setting variable values and evaluating expressions in named environments. Futhermore, since named environments are system objects, jumps to these environments are very efficient.

5. Special Features of TLOG

a. Integration: TLOG is fully embedded in T in a way that allows the user: (1) to treat TLOG as a "stand-alone" programming language, (2) to call TLOG as a theorem prover or deductive retrieval system from T, or (3) to call T code from TLOG. Since TLOG accepts Prolog syntax it can also be used immediately by someone familiar only with Prolog.

b. Predefined predicates: the TLOG-to-T connection: TLOG provides the user with a basic set of predefined (evaluable or built-in) predicates for arithmetic, I/O, file manipulation, etc. But TLOG, unlike most Prologs, allows the programmer to extend this list of built-in predicates arbitrarily by a simple procedure: It is only necessary to build a list associating the predicate with an appropriate T function and these new predicates will be defined as built-in.

This is an important feature of TLOG, especially for the uses for which it was designed, because it is through the predefined predicates that the TLOG-to-T part of the integration is implemented. This allows the TLOG programmer to call graphics functions [Mueller & Zernik 1984], spawn demons for parsing [Dyer 1983], or direct customized output to a newly opened window. The extendible predefined predicates allow T and TLOG to interact fully.

c. Trace/debug facilities: TLOG has a trace package modelled on the trace in Edinburgh Prolog and C-Prolog and thus provides the usual options (i.e. creep, skip, quasi-skip, leap, retry, etc.). TLOG also provides trace options that go beyond most logic programming systems. These options include algorithmic debugging and editing.

(1) Algorithmic debugging: In TLOG this facility is based on Shapiro's work in this area [Shapiro 1983]. It is basically an explanation system. In case of unanticipated success TLOG can be queried as to its reasoning and it will backtrack from the goal, asking the user at each stage whether the predicate it is using is true or not. If a predicate is false, TLOG asks for a ground instance which is false. It continues until it finds a clause in the database which is false. In case of unexpected failure, TLOG

switches to a special purpose trace monitor and reexamines its attempt at a proof. The new monitor reports to the user each point at which the proof attempt failed. Once a failure is reported, remaining backtrack failures are ignored until another line of proof is started. These features, corresponding to Shapiro's false-procedure and incomplete-procedure, are available both as T functions and as options in any running trace.

(2) Editing in TLOG: TLOG provides facilities for editing the database and for "in-core" editing of individual clauses. In addition to the usual **ASSERTA**, **ASSERTZ** and **RETRACT**, TLOG provides **REMOVE** and **RESTORE**. REMOVE marks a clause as "removed" in such a way as to allow it to be restored later at the same position in the database. This gives the programmer the possibility of experimenting with the database, whether in debugging or in analyzing the code, without changing the structure of the database. Using **RETRACT** and **ASSERTA/Z** to do this sort of experiment by contrast makes significant and potentially serious changes in the program. **REMOVE** and **RESTORE** can be called both from T as functions and called as options in a running trace. This feature of TLOG uses the fact that in TLOG clauses are objects with a **DELETED?** method which can be set to **T** or **NIL**. Marking a clause as d eleted simply causes the fetch from the discrimination net to skip that clause.

To edit individual clauses, TLOG provides a **CLAUSE-EDIT** feature, which also is available either as a T function or as an option in a running trace. The programmer can modify the head, body or any part of the head or body and the resulting clause is stored in the database in place of the original clause. TLOG implements this editing by using the fact that in the clause the head and body are settable access functions of the clause-object.

6. Future Work.

There are several future projects for TLOG, some of which are already underway.

a.. Current projects:

(1) Compiling TLOG directly into T: Since T compiles very well (and version 3.0 of T is expected to produce significantly faster compiled code) this pre-compilation should speed up TLOG substantially. One approach to this is already implemented [Dolan & Dyer 1984] and this TLOG compiler has resulted in a speed improvement a factor of 3. We expect further improvements in this area.

(2) Continuation passing: There is current work on building a continuation passing theorem prover [Mellish & Hardy 1984]. We expect that this will speed up TLOG as well as being of research interest.

(3) Graphical trace/debug: The AI Laboratory at UCLA is continuing development on its Graphical AI Tools Environment (GATE [Mueller & Zernik 1984]). E.g. we are current working on adding a graphical trace facility to the existing trace. Eventually, we hope to extend this to allow graphical interaction between a TLOG program and the programmer, including graphical editing and graphical display of unification and backtracking.

b. Longer-range projects include:

(4) Using the discrimination net to support unification: A fetch from the discrimination net currently does a "partial unification" by checking for all "constant-to-constant" matches and allowing variables to match anything. We are interested in developing a unifier that can use this information to improve its performance, possibly using extra information about the clause from the clause object. Alternatively, consideration has been given to integrating the unifier with the discrimination net so that the fetch does unification.

(5) Using the clause object to support unification: At the cost of taking more time to add clauses to the database, the clause object could store more information local to the clause. For example, the directed acyclic graph (dag) structure of the head or all or part of the body of the clause could be computed and used by a unifier based on the dag structure [Patterson and Wegman 1976].

(6) Metalevel control: TLOG, like Prolog, currently always chooses the next clause as listed in the database and the next literal in the body of the clause to be resolved on. Suggested methods for giving the programmer more control over these selection rules include annotating the variables and extending the interpreter to use "control clauses", instructions with the syntax of Prolog clauses which change the path of clause or literal selection [Gallaire and Lasserre 1982]. We are currently looking into adding some sort of metalevel control to TLOG.

(7) Shared TLOG code: At the moment, when TLOG is running in several environments the full TLOG code is copied into each environment. We would like to have the core code running in a single environment using the various distributed databases.

(8) Using the discrimination net for literal selection: Consideration is being given to storing extra information in the discrimination net, for example, the number of clauses under a given node. Then literals could be chosen in a "fewest possible unificants first" manner.

(9) Concurrent TLOG: Of course, many people are working on concurrent logic programming. We expect that the message passing object semantics of T along with named environments will open up possibilities for concurrency that would be more difficult in other languages.

7. Conclusions

Our initial goals for TLOG have largely been achieved. TLOG exists as a logic programming system fully embedded in T and provides users with features not generally available in other pure Prolog systems. It is already serving as a base for further experiments in compilation, alternate theorem provers, and graphical trace/debugging.

The most interesting result may be the use of objects to implement logic. Treating clauses as objects, for example, allows the implementer to support the rename-variables function, database manipulation and in-core editing of clauses. The message-passing semantics of objects provides a uniform way of adding information to various entities in a logic system, such as clauses, d-net nodes and variables, and also in making that information available to other parts of the system.

As far as we know, TLOG is the only object-based logic programming system in existence*.

8. Acknowledgments

We would like to gratefully acknowledge contributions in both design and implementation made by Charlie Dolan, Maria Fuenmayor, Hamid Nabavi, Kurt Stoll and Scott Turner on the TLOG compiler, unifier, trace package, d-net, and Prolog front-end, respectively. In addition, we also want to thank Charlie Dolan for his valuable editorial and substantive aid in the preparation of this document.

9. References

Abelson, H., Sussman, G. J. with Sussman, J., *Structure and Interpretation of Computer Programs.* MIT Press, MA, 1985.

Dolan, C. and Dyer, M. G. "The World's Shortest PROLOG Compiler?" UCLA AI lab memo, 1984.

Dyer, M. G. *In-Depth Understanding.* MIT Press, MA, 1983.

Gallaire, H. and Lasserre, C. "MetaLevel Control for Logic Programs", in K. L. Clark and S. -A. Tarnlund (eds.). *Logic Programming.* Academic Press, NY, 1982.

Kahn, K. M. "INTERMISSION – Actors in Prolog", in K. L. Clark and S. -A. Tarnlund (eds.). *Logic Programming.* Academic Press, NY, 1982.

Kahn, K. M. and Carlsson, M. "How to implement Prolog on a LISP Machine", in *Implementations of PROLOG.* J. A. Cambell (ed.), Ellis Horwood Limited, NY. 1984.

Kowalski, R. *Logic for Problem Solving.* North Holland, NY 1979.

Mellish, C. and Hardy, S. "Integrating Prolog in the Poplog Environment", in *Implementations of PROLOG.* J. A. Cambell (ed.), Ellis Horwood Limited, NY. 1984.

Mueller, E. and Zernik, U. "GATE Reference Manual". UCLA AI lab, tools note 6, Oct. 1984.

*The next most likely candidate appears to be LM-Prolog, which is implemented in Zetalisp [Weinreb and Moon 1981] but its implementers apparently did not use Zetalisp flavors [Kahn and Carlsson 1984].

Nilsson, N. J. *Principles of Artificial Intelligence.* Tioga Press, Palo Alto, CA. 1980.

Paterson, M.S. and Wegman, M.N. "Linear Unification", *Proc. Annual ACM Symposium on Theory of Computing,* 1976.

Rees, J. A. and Adams, N. I. "T: a dialect of LISP, or lambda: the ultimate software tool", in *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming.* August, 1982.

Rees, J. A., Adams, N. I. and Meehan, J. R. *The T Manual,* Computer Science Dept. Yale University, 1983.

Shapiro, E. Y. *Algorithmic Program Degubbing.* MIT Press, MA, 1983.

Warren, D. H. D. *Implementing Prolog -- Compiling Logic Programs,* Vol 1 & 2, D. A. I. Research Reports 39 & 40, University of Edingurgh. 1977.

Weinreb, D. and Moon, D. *LISP Machine Manual.* MIT, 4th edition, 1981.