THE WORLD'S SHORTEST PROLOG COMPILER

Charlie Dolan
Michael G. Dyer

April 1986
CSD-860068

The World's Shortest PROLOG Compiler?*

Charlie Dolan
UCLA Artificial Intelligence Laboratory
Hughes Aircraft AI Center

Michael G. Dyer
UCLA Artificial Intelligence Laboratory

ABSTRACT

This paper presents a compiler for TLOG, an implementation of TLOG in T [Rees et. al. 1983], a version of SCHEME [Abelson and Sussman 1984]. Itself is a dialect of LISP. In this compiler TLOG clauses are translated into T code. The compiler is presented as a simple efficient way of doing logic programming in LISP. The compiler uses the continuation passing method for building the proof tree on the execution stack [Kahn and Carlson 1984, Mellish and Hardy, 1984]. Several aspects of compilation are discussed: variables, cut (!), disjunction, structure copying, and unification. The code for implementing all but the last is given in the appendix. This compiler is part of the UCLA TLOG project. TLOG [Read et. al. 1984] is logic programming in T. Performance results for compiled vs. interpreted TLOG are given.

* This work was supported in part by a grant from the Hughes Aircraft AI Center and in part by a grant from the Keck Foundation.

The World's Shortest PROLOG Compiler?

Charlie Dolan
UCLA Artificial Intelligence Laboratory
Hughes Aircraft AI Center

Michael G. Dyer
UCLA Artificial Intelligence Laboratory

1. Why compile into T?

There are three reasons for compiling TLOG into T.

(1) It simplifies the problem of compilation. If you already have a LISP compiler, you don't have to write a code generator. This paper is somewhat inspired by Nilsson's "shortest" PROLOG interpreter in LISP [Nilsson, 1984]. The code presented here will give you reasonable PROLOG compiler kernel to build on given that the your LISP has all the needed features: closures and **CATCH** (e.g. Zetalisp [Weinreb and Moon 1981], COMMON LISP [Steele 1984], SCHEME).

(2) Is that there are already large programming environments for LISP, and much software has been committed to those environments (e.g. graphics, editors, programming tools). Implementing logic programming in LISP means that we don't need to build all these tools for a logic programming language [Robinson and Sibert, 1982].

(3) If you are programming primarily in LISP you want the interface between LISP and PROLOG (in our case T and TLOG) to be as efficient and transparent as possible. The best way to do this is to turn PROLOG into LISP. In the TLOG system a set of macros is provided which insert the user's code into the compiled TLOG. This way the user has full access to all the variables in the TLOG code.

2. T features used

The main feature of T that is used in the compiler are closures. A closure is a procedure (lambda expression) and an environment in which the procedure is to execute. Closures are used to implement the continuations discussed in the next section. Here is a brief example of a closure. Given procedures,

```
(DEFINE (PROC1)
  (LET ((X 3))
    (PROC2 (LAMBDA () X))))

(DEFINE (PROC2 CONT)
  (LET ((X 4))
    (CONT)))
```

PROC1 will return 3 because **X** is bound in the environment of **PROC1** and not in **PROC2**. In a LISP without closures **PROC2** would return 4 because 4 is the most recent value assigned to **X**.

Another feature of T that is used is **CATCH**. **CATCH** allows a procedure to **THROW** beyond its normal return point. Consider:

```
(DEFINE (PROC1)
  (CATCH CUT-PROC
    (PROC2 (LAMBDA () (CUT-PROC NIL))) T))

(DEFINE (PROC2 CONT)
  (CONT))
```

If **PROC2** had not invoked **CONT**, **PROC1** would have returned **T**, the last expression in the **CATCH**. But **PROC2** invoked **CONT** which contained an invocation of **CUT-PROC** set up by the **CATCH**. This forces the entire **CATCH** expression to be terminated and the value with which the **THROW** procedure, **CUT-PROC**, was invoked to be returned.

The last feature of T used is the macro facility. TLOG* procedures are declared with the macro

* TLOG is a superset of PROLOG imbedded in T. For instance a PROLOG clauses like: **relation(X) :- body** become in TLOG: **(:- (RELATION ?X) BODY)**. TLOG also has a front end which will handle PROLOG syntax in addition to TLOG syntax.

DEFINE-TLOG.

```
; name(PATTERN1) :- body1.
; name(PATTERN2) :- body2.
;     .
;     .
;     .
(DEFINE-TLOG (name arg1 arg2 ... argn)
       (:- (name PATTERN1) body1)
       (:- (name PATTERN2) body2)
                .
                .
                .
)
```

where each $body_i$ is a goal. A goal can be a TLOG procedure call, (AND $goal_1$ $goal_2$ ... $goal_m$), or (OR $goal_1$ $goal_2$ ... $goal_m$). The entire body gets turned into a procedure definition with $n+1$ arguments, where the last argument is the continuation.

3. Downward success continuation

In downard success continuation, a procedure succeeds by calling its continuation. The user must then specify, as part of the call to prove a goal, what is to be done if the goal is true. Returning to the caller is failure. When compiling TLOG, this means that proving subgoals consists of constructing nested continuations to call them. Consider the following,

```
; p1.
; p1 :- p2.
; p1 :- p3,p4.
(DEFINE-TLOG (P1)
    (:- (P1))
    (:- (P1) (P2))
    (:- (P1) (AND (P3) (P4)))
)
```

Which compiles into**,

** Here and in the remainder of this paper, object code produced by the compiler is simplified for clairity. In particular, code for resetting variables bound during evaluation of a failed goal is omitted.

```
1       (DEFINE (P1/0 *CONTINUATION*)
2         (*CONTINUATION*)
3         (P2/0 *CONTINUATION*)
4         (P3/0 (LAMBDA () (P4/0 *CONTINUATION*)))
5       )
```

It is assumed that when the user calls a top-level goal from a T program, a suitable continuation is supplied. In most cases the continuation is constructed by the macros for calling TLOG from T presented later in this paper.

We see 4 things demonstrated here:

(1) As in line 2, success consists of calling the continuation.

(2) In line 3, proving subgoals is simply calling the procedure for the sub-goal.

(3) As in line 4, proving a conjuction of sub-goals involves calling the first conjunct and passing to it a continuation to call the remaining conjuncts.

(4) In line 5 failure is simply returning from a procedure.

The implementation of continuations is extremely efficient in T because T was designed with the goal of efficient implementation of **LAMBDA** closures. Besides the code itself, the overhead for using a **LAMBDA** closure in T is one **CONS** cell plus one more for each variable in the calling procedure. [Rees et al 1983].

4. Implementing logical variables in T

T only has normal programming variables, so we need to create a new type of structure to represent logical variables. In this implementation TLOG variables are implemented with *structures* [Rees et all 1983] we call **PROVARs**. These are record structures whose value component is accessed with the selector, (**PROVAR-VALUE** *obj*). New logical varaibles are created with a call to (**MAKE-PROVAR**). The value component is initially the atom ***UNDEF***. With these logical variables we can now compile func-

tions which have variables. For instance,

```
; append([],L,L).
; append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
(DEFINE-TLOG (APPEND A1 A2 A3)
   (:- (APPEND () ?L ?L))
   (:- (APPEND (CONS ?X ?L1) ?L2 (CONS ?X ?L3))
       (APPEND ?L1 ?L2 ?L3))
)
```

becomes

```
1     (DEFINE (APPEND/3 A1 A2 A3 *CONTINUATION*)
2       (LET ((?L A2))
3         (UNIFY (LIST A1 A2 A3)
4                (LIST () ?L ?L)
5                *CONTINUATION*)))
6       (LET ((?X (MAKE-PROVAR) (?L1 (MAKE-PROVAR))
7             (?L2 A2) (?L3 (MAKE-PROVAR)))
8         (UNIFY (LIST A1 A2 A3)
9                (LIST (LIST 'CONS ?X ?L1) ?L2 (LIST 'CONS ?X ?L3))
10               (LAMBDA () (APPEND/3 ?L1 ?L2 ?L3 *CONTINUATION*)))))
```

In the first and second clauses of **APPEND**, **?L** and **?L2** (lines 2,7) are local variables [Warren 1977]. Local variables are needed only to pass procedure arguments to other parts of the procedure. They are not used in structure selection and construction. In these cases no logical variables need to be constructed. For the other variables **MAKE-PROVAR** is used to make a new global logical variable, as in lines 6-7. Logical variables in the TLOG clauses are mapped to T variables which are bound to either procedure arguments or unbound logical variables.

Note here is that the unifier (lines 3 and 8) takes three arguments: two lists of terms to unify, and a continuation. If the unification is successful, the continuation is called. If the unification fails or if the continuation fails (.i.e. returns), then **UNIFY** unbinds all the variables which were bound during the unification process.

In the second clause of **APPEND/3**, line 9, we can see that the compiler also creates code for do-

ing the structure copying, with **LIST**.

5. Compiling cut (!)

There are two ways to implement cut (!) using **CATCH**. The first is straightforward to implement, but does not have the desirable attribute of cut (!) i.e. that it removes frames from the stack. The second requires more work by the compiler but preserves both the semantics and the desirable side effects of cut (!). As an example look at the procedure below.

```
(DEFINE-TLOG (P1)
    (:- (P1) (AND (P2) (!) (P3)))
    (:- (P1) (P4)))
```

The first way to implement this is to use a built-in function for cut (!), and to have this built-in procedure **THROW** back to the original procedure. This requires that we use the T dynamic binding mechanism **BIND** [Rees et al, 1983] in order to defeats the lexical scoping rules.

```
1    (DEFINE (P1/0 *CONTINUATION*)
2       (CATCH THROW
3         (BIND ((*CUT* THROW))
4           (P2/0 (LAMBDA () (!/0 (LAMBDA () (P3/0 *CONTINUATION*)))))
5           (P4/0 *CONTINUATION*))))
```

where **!/0** is defined as,

```
(DEFINE (!/0 *CONTINUATION*)
    (*CONTINUATION*)
    (*CUT* NIL))
```

Here is how it works. If **P2/0** succeeds (line 4), it will call its its continuation which is a call to **!/0**. This will immediately call **P3/0** with the original continuation. If **P3/0** fails, i.e. returns, **!/0** will **THROW** out of **P1/0**, not trying **P4/0**, the code for the second clause. If **P2/0** fails, **!/0** will not get called and **P4/0** will get called. This works fine and gives us a correct implementation of cut (!). However with this implementation, programmers cannot use cut (!) to make their programs more efficient, only to change the way they execute. The reason is that after the cut is encountered, the frame for the call to **P2/0** will still be on the stack. If we want cut (!) to make programs more efficient we should instead use

the following implementation,

```
1      (DEFINE (P1/0 *CONTINUATION*)
2          (CATCH FAIL
3           (CATCH NO-CUT
4               (CATCH CUT
5                   (P2/0 (LAMBDA () (CUT NIL)))
6                   (NO-CUT NIL))
7               (P3/0 *CONTINUATION*)
8               (FAIL NIL)))
9           (P4/0 *CONTINUATION*))))
```

What has been done here is to wrap the entire function in **CATCH/FAIL** (at lines 2 and 9). If we ever want to cut of out other clauses we just **(FAIL NIL)**. Since this is a local **CATCH** it just turns into a jump when compiled by the T compiler [Rees et. al. 1983]. **CATCH/CUT** (lines 4-6) is used to trap the success of **P2/0**. A **(CUT NIL)** is passed as the continuation to **P2/0**, line 5. This means that when **P2/0** succeeds, it will **THROW** and will therefore get removed from the stack, then **P3/0** gets called with the continuation. If **P3/0** fails, the procedure will **THROW FAIL** and that will fail the entire procedure. If **P2/0** fails the procedure will **THROW NO-CUT** and will go to the next clause, the code for invoking **P4/0** at line 9.

Besides taking frames off the stack, this is more efficient in terms of the the code the T compiler produces. The **BIND** implementation causes the creation of one non-local **CATCH** which takes 1 **CONS** cell, but the use of **BIND** causes the creation of 3 **LAMBDAs** which take 2 **CONS** cells each for a total of 7 cells [Rees, 1984]. The second implementation has a non-local **CATCH** which takes 1 **CONS** cell. This makes the second implementation much more attractive.

6. Compiling disjunction

Disjunction introduces an interesting problem in construction of continuations. Because an **OR** may be contained in an **AND**, we have to traverse the entire AND/OR tree and construction the con-

tinuations for the conjuncts after the **OR**. As an example, look at the procedure,

```
(DEFINE-TLOG (P1)
    (:- (P1) (AND (P2)
                  (OR (AND (P3) (P4))
                      (P5))
                  (P6))))
```

which compiles into

```
1     (DEFINE (P1/0 *CONTINUATION*)
2       (P2/0 (LAMBDA ()
3               (P3/0 (LAMBDA () (P4/0 (LAMBDA () (P6/0 *CONTINUATION*)))))
4               (P5/0 (LAMBDA () (P6/0 *CONTINUATION*))))))
```

We see that the compiler functions depth first, making the continuation for the last conjunct, **P6/0** before it compiles the disjunction. The call to **P6/0** becomes part of the continuation for both disjuncts (lines 3 and 5). The disjunction itself simply involves sequential statements in the **LAMBDA** expression (lines 3-4) to try the different alternatives.

7. Structure copying

One of the advantages of compilation is that we can compile away almost all of the overhead of interpreted structure copying. The TLOG code,

```
(F1 ?X (F2 A B) (F3 ?Y))
```

turns quite straightforwardly into,

```
(LIST 'F1 ?X (LIST 'F2 'A 'B) (LIST F3 ?Y)).
```

As part of the interface to the host language, T, the compiler also understands the construct **(EVAL** *obj*). This tells the compiler not to quote or further compile *obj*. **EVAL** can be used for inserting global constants into TLOG programs. An example of this would be:

```
(MEMBER ?X (EVAL PROPER-NAME-LIST))
```

where **PROPER-NAME-LIST** is maintained by the T program. This gets translated simply to:

```
(MEMBER/2 ?X PROPER-NAME-LIST)
```

So **EVAL** is a flag to the TLOG compiler which tells it to pass an expression along, unaltered to the T compiler.

8. Compiling unification

The primary place where compilation helps in logic programming is unification. This is because PROLOG-style languages spends almost all of their time doing unifications. Here we see what **APPEND** would look like if all the unification code were included in line.

```
(DEFINE-TLOG (APPEND A1 A2 A3)
   (:- (APPEND () ?L ?L))
   (:- (APPEND (CONS ?X ?L1) ?L2 (CONS ?X ?L3))
       (APPEND ?L1 ?L2 ?L3))
)
```

which turns into (with unification compilation)

```
1     (DEFINE (APPEND/3 A1 A2 A3 *CONTINUATION*)
2       (LET ((?L NIL))
3         (AND (OR (EQ? A1 ())
4                  (AND (UNDEF? A1) (SET (PROVAR-VALUE A1) ())))
5              (SET ?L A2)
6              (UNIFY A3 ?L *CONTINUATION*)))
7       (LET ((?X NIL)) (?L1 NIL)
8            ((?L2 NIL) (?L3 NIL))
9         (AND (OR (AND (UNDEF? A1)
10                      (SET ?X (MAKE-PROVAR))
12                      (SET ?L1 (MAKE-PROVAR))
13                      (SET (PROVAR-VALUE A1) (LIST 'CONS ?X ?L1)))
14                 (AND (EQ? (FUNCTOR A1) 'CONS)
15                      (SET ?X (ARG 1 A1))
16                      (SET ?L1 (ARG 2 A1))))
17             (SET ?L2 A2)
18             (OR (AND (UNDEF? A3)
19                      (SET ?L3 (MAKE-PROVAR))
20                      (SET (PROVAR-VALUE A3) (LIST 'CONS ?X ?L3))
21                      (*CONTINUATION*))
22                 (AND (EQ? (FUNCTOR A3) 'CONS))
23                      (SET ?L3 (ARG 2 A3))
24                      (UNIFY (ARG 1 A3) ?X *CONTINUATION*))))
```

Lines 3-4 show that the unification consists of **AND**ing together the results of matching all the terms in the clause head. For any argument where there is a choice, we construct an **OR** as in lines 9-16. One branch of the **OR** is for the parameter being unbound, in which case we allocate new logical variables, and construct the term, line 13. Otherwise we test the functor of the argument with functor of the pattern, line 14, and if it is equal, we set **?X** and **?L1** to the arguments of the functor.

9. Calling TLOG from T

One of the benefits of compiling TLOG into T is that it makes calling TLOG from T very simple. All that the user needs is some syntactic sugar to construct the required continuations. The macro provided is **TLOG-LOOP**. The syntax is

(TLOG-LOOP *name Tlog-goal*
*T-code***)**

The first argument is similar to the **THROW** tag in a **CATCH**. It provides the equivalent to a PROLOG **fail** from the *T-code*. It is called when another solution to *Tlog-goal* is required. This means that

TLOG-LOOP can be used as a looping construct to process all the solutions to a goal. Here is an example of the use of **TLOG-LOOP** to print out all the solutions of the TLOG query: **(APPEND ?L1 ?L2 [A B C])**.

```
(SET LST '(CONS A (CONS B (CONS C ()))))
(TLOG-LOOP APP (APPEND ?L1 ?L2 (EVAL LST)
   (PRINT ?L1 (STANDARD-OUTPUT))
   (NEWLINE (STANDARD-OUTPUT))
   (PRINT ?L2 (STANDARD-OUTPUT))
   (NEWLINE (STANDARD-OUTPUT))
   (NEWLINE (STANDARD-OUTPUT))
   (APP))
```

which prints out the following,

```
()
(CONS A (CONS B (CONS C ())))

(CONS A ())
(CONS B (CONS C ()))

(CONS A (CONS B ()))
(CONS C ())

(CONS A (CONS B (CONS C ())))
()
```

which are all the lists which can be appended to make **LST**.

10. Calling T from TLOG - IS

As in PROLOG, the main interface to expression evaluation in TLOG is through **IS** which evaluates its second argument and unifies it with the first. To do this it simply de-references all the terms in the second arguments and assumes that all functors are defined T functions. For example,

```
(IS ?X (* ?Y (+ ?Z 3)))
```

becomes

```
(LET ((VALUE (* (DE-REF ?Y) (+ (DE-REF ?Z) 3))))
  (UNIFY VALUE ?X *CONTINUATION*))
```

The function **DE-REF** is used to de-reference TLOG variables to either a compound term or an unbound logical variable. What is especially nice about this method of implementation is that the user does not

have to do anything special to import T functions into TLOG.

11. Conclusions and future work

The current version of the TLOG compiler does not compile unifications, but does distinguish between local and global variables. Without the compilation of unification, compiled TLOG runs almost 3 times faster than interpreted TLOG. On an Apollo DN300, a 68000 based workstation, compiled TLOG runs at 100 LIPS. University of New Hampshire interpreted PROLOG runs at 250 LIPS on the same workstation. This seems unbalanced at first, when considering that TLOG is compiled, but not as much once we consider that TLOG is imbedded in a much larger programming environment. Things which are very inefficient in PROLOG can be done in TLOG by implementing them in T. Also preliminary results with hand-compiled code show that by compiling unification a speed-up to 200-300 LIPS can reasonably be obtained

One of the biggest advantages of using TLOG for logic programming is that it is easier to experiment with embedding other programming paradigms into logic programming. TLOG already has functional programming built in due to T. Also since T is an object-oriented LISP, it is easy to put in message passing; simply use T **OBJECT**s as terms, and have them respond to the message **UNIFY-TERM**. The integration, with TLOG, of an existing frame system and a demon system already embedded in T are also underway.

Since the TLOG unifier is relatively small and easy to modify, it is also more feasible to experiment on a large scale with equality and typed logics. Experimentation can be done on a larger scale because the user doesn't have to run his code through an interpreter which has a high overhead. For example, a typed logic can be implemented by modifying the unifier, written in T, to respect variable

types.

Appendix A: Compiler Code

The macros for creating Tlog procedures

```
(HERALD COMPILER_MACROS (ENV T))

; This file contains the macros for a short Tlog compiler
; for T. The way to define Tlog procedures in T is the
; DEFINE-TLOG macro. The syntax for that macro is described
; below.

; A T STRUCTURE is used to represent variables. The structure
; is called PROVAR and has one component, VALUE. The following
; functions are defined,
;
;       (MAKE-PROVAR) makes a new unbound variable
;       (PROVAR? obj) is a predicate true on PROVARs
;       (PROVAR-VALUE provar) yields the VALUE component
;
; The VALUE component is initialized to *UNDEF*. This is the
; flag for an unbound Tlog variable.
;
; The the default print method for the PROVAR structure
; is changed so that it will automatically de-reference itself
; if it is bound. Otherwise, it prints its internal system
; number.

(DEFINE-STRUCTURE-TYPE PROVAR VALUE)
(SET (PROVAR-VALUE (STYPE-MASTER PROVAR-STYPE)) '*UNDEF*)
(DEFINE-METHODS (STYPE-HANDLER PROVAR-STYPE)
    ((PRINT SELF STREAM)
        (COND ((EQ? (PROVAR-VALUE SELF) '*UNDEF*)
               (FORMAT STREAM "#{ A}" (OBJECT-HASH SELF)))
              (T (PRINT (PROVAR-VALUE SELF) STREAM)))))

; The following functions are defined on expressions which are
; represented during compilation as lists.
```

```
(DEFINE (FUNCTOR:TERM TERM) (CAR TERM))
(DEFINE (ARGLIST:TERM TERM) (CDR TERM))
(DEFINE (ARG:TERM N TERM) (NTH (ARGLIST:TERM TERM) (- N 1)))

(DEFINE (CLAUSE? LST) (EQ? (CAR LST) ':-))
(DEFINE (HEAD:CLAUSE CLAUSE) (CADR CLAUSE))
(DEFINE (BODY:CLAUSE CLAUSE) (CADDR CLAUSE))

(DEFINE (BODY:LAMBDA LAMBDA-EXPR) (CDDR LAMBDA-EXPR))


; This is the macro for defining Tlog procedures. The syntax is:
;
;       (DEFINE-TLOG (name . args)
;               clause1 ... clauseN)
;
; Any of the clauses which start with (:- ...) are turned into
; Tlog code. Anything else is just inserted in line into the procedure.
; The syntax for clauses is,
;
;       (:- head goal) where
;
;       'goal' can be - a Tlog procedure invocation
;                       (AND goal1 ... goaln)
;                       (OR goal1 ... goaln)
;
; The BACKQUOTE macro (`) is used to create the code for defining
; the procedure.
;
; The name gets change from NAME to NAME/arity by the function
; TLOG-FUNC.
;
; In addition to the arguments declared, Tlog procedures get an
; extra argument *CONTINUATION* which is the function to call
; upon success.
;
; OLD-RESET-LIST is used to maintain a pointer to the variable
; stack where the procedure was called. It is used to clean up
; the bindings after a CUT exit.
;
; The CATCH/FAIL is to be used by procedures which fail after
; encountering a cut (!).
;
; We insert the code for each of the clauses in order. Things
; which are not Tlog clauses get inserted also.
;
; If a clause cuts and fails we have to restore the variables that
; were bound. This is done with the call to RESET-GLOBAL-LIST at the
; end of the procedure.

(DEFINE-SYNTAX (DEFINE-TLOG PATTERN . CLAUSES)
    `(DEFINE (,(TLOG-FUNC PATTERN) ,@(ARGLIST:TERM PATTERN) *CONTINUATION*)
```

```
        (LET ((OLD-RESET-LIST *GLOBAL-RESET-LIST*))
             (CATCH FAIL
                    ,@(MAP (LAMBDA (CLAUSE)
                                   (COND ((CLAUSE? CLAUSE)
                                          (MAKE-INVOCATION PATTERN
                                                           CLAUSE))
                                         (T CLAUSE)))
                           CLAUSES))
             (RESET-GLOBAL-LIST OLD-RESET-LIST))))

; MAKE-INVOCATION takes a single clause and sets up the code for it.
; It scans the clause (with UNIQ-VAR) to get a list of the variables
; it needs. It uses MAKE-CONTINUATION to create the code.
;
; Most variables get bound to (MAKE-PROVAR). However, if a variable
; is local it can be directly bound to the corresponding argument of the
; procedure.

(DEFINE (MAKE-INVOCATION PATTERN CLAUSE)
    (LET ((VAR-LIST (UNIQ-VAR CLAUSE))
          (ACTUAL-ARGS (ARGLIST:TERM PATTERN))
          (FORMAL-ARGS (ARGLIST:TERM (HEAD:CLAUSE CLAUSE)))
          (HEAD (HEAD:CLAUSE CLAUSE))
          (BODY (BODY:CLAUSE CLAUSE)))
         `(LET (,@(MAP
                    (LAMBDA (VAR)
                            (COND ((LOCAL? VAR HEAD)
                                   `(,VAR ,(FIND-ARG VAR HEAD PATTERN)))
                                  (T `(,VAR (MAKE-PROVAR)))))
                    VAR-LIST))
               ,(MAKE-CONTINUATION ACTUAL-ARGS FORMAL-ARGS
                    BODY '*CONTINUATION*))))

; MAKE-CONTINUATION starts making the code.
;
; If there are no actual args, then just make the goal continuation.
;
; Otherwise, set up the call to UNIFY for the actuals and the formals,
; and pass UNIFY the goal continuation.

(DEFINE (MAKE-CONTINUATION ACTUAL-ARGS FORMAL-ARGS GOAL CONT)
    (COND ((NULL? ACTUAL-ARGS)
           (CONS 'BLOCK (BODY:LAMBDA (MAKE-GOAL-CONTINUATION GOAL CONT))))
          (T `(UNIFY (LIST ,@ACTUAL-ARGS)
                     (LIST ,@(MAP MAKE-CONSTRUCTOR FORMAL-ARGS))
                     ,(MAKE-GOAL-CONTINUATION GOAL CONT))))))

; This is the real code constructor, MAKE-GOAL-CONTINUATION. The special
; cases it needs to handle are:
;       !
;       IS
;       AND
```

```
;       OR
;
; The cases,
;       No goals, just return the continuation.
;
;       A cut, return a procedure which calls the continuation and
;       throws to the CATCH/FAIL.
;
;       IS needs a procedure which evaluates the expression (after suitable
;       syntactic massaging by MAKE-EXPR) and unifies it with first argument,
;       passing the original continuation along to UNIFY.
;
;       conjunction: see MAKE-CONJ-CONTINUATION
;
;       disjunction: MAKE-DISJ-LIST returns a list or forms to be run.
;       Here we wrap a LAMBDA around it.
;
; Anything else gets turned into a procedure call which gets passed
; the continuation. MAKE-CONSTRUCTOR creates the structure copying
; code.

(DEFINE (MAKE-GOAL-CONTINUATION GOAL CONT)
 (LET ((ARGLIST (ARGLIST:TERM GOAL)))
    (COND ((NULL? GOAL) CONT)
          ((ALIKEV? GOAL '(!))
           `(LAMBDA () (,CONT) (FAIL NIL)))
          ((EQ? (FUNCTOR:TERM GOAL) 'IS)
           `(LAMBDA ()
               (LET ((VALUE ,(MAKE-EXPR (ARG:TERM 2 GOAL))))
                    (UNIFY ,(ARG:TERM 1 GOAL) VALUE ,CONT))))
          ((EQ? (FUNCTOR:TERM GOAL) 'AND)
           (MAKE-CONJ-CONTINUATION ARGLIST CONT))
          ((EQ? (FUNCTOR:TERM GOAL) 'OR)
           `(LAMBDA () ,@(MAKE-DISJ-LIST ARGLIST CONT)))
          (ELSE `(LAMBDA () (,(TLOG-FUNC GOAL)
                             ,@(MAP MAKE-CONSTRUCTOR ARGLIST)
                             ,CONT)))))))

; MAKE-CONJ-CONTINUATION has two important cases, CUT and no-CUT.
;
; If there is a cut...
; ...get the goals before and after it...
; We create two CATCHs. CATCH/NO-CUT is used when the goals
; before the CUT fail and we want to try other choices in the
; procedure.
; CATCH/CUT is used to pass as the continuation to the goals
; before the cut. If they succeed, it will be thrown, then the goals
; after the cut will be tried. If the goals after cut fail, the
; CATCH/FAIL made at the top of the Tlog procedure will be thrown,
; and the rest of the clauses will be skipped.
;
; Otherwise we just make a plain conjunctive goal.
```

```
; NOTE: We make the continuation for the tail of the list first
; and then make a goal continuation for the first one. This is
; important for compiling disjunctions.

(DEFINE (MAKE-CONJ-CONTINUATION GOALS CONT)
 (LET ((BEFORE-CUT NIL) (AFTER-CUT NIL))
    (COND ((NULL? GOALS) CONT)
          ((HAS-CUT? GOALS)
           (SET BEFORE-CUT (FIND-BEFORE-CUT GOALS))
           (SET AFTER-CUT (FIND-AFTER-CUT GOALS))
           `(LAMBDA ()
              (CATCH NO-CUT
                (CATCH CUT
                  ,@(BODY:LAMBDA (MAKE-CONJ-CONTINUATION BEFORE-CUT
                                   '(LAMBDA () (CUT NIL))))
                  (NO-CUT NIL))
                ,@(BODY:LAMBDA (MAKE-CONJ-CONTINUATION AFTER-CUT CONT))
                (FAIL NIL))))
          (T (LET
               ((NEW-CONT
                    (MAKE-CONJ-CONTINUATION (CDR GOALS) CONT)))
               (MAKE-GOAL-CONTINUATION (CAR GOALS) NEW-CONT))))))

; MAKE-DISJ-LIST
; In a disjunction, nothing after a cut  at the top level will
; ever get executed, so why not remove it.

(DEFINE (MAKE-DISJ-LIST GOALS CONT)
    (SET GOALS (COND ((HAS-CUT? GOALS) (FIND-BEFORE-CUT GOALS))
                     (T GOALS)))
    (COND ((NULL? GOALS) NIL)
          (T
            (APPEND! (BODY:LAMBDA (MAKE-GOAL-CONTINUATION (CAR GOALS) CONT))
                (MAKE-DISJ-LIST (CDR GOALS) CONT)))))

; MAKE-CONSTRUCTOR compiles the structure copying code.
;
; EVAL is a flag to indicate that this argument is neither
; a logical variable nor a constant but to be evaluated by
; T

(DEFINE (MAKE-CONSTRUCTOR ARG-SPEC)
    (COND ((ATOM? ARG-SPEC)
           (COND ((IS-VAR-ATOM? ARG-SPEC) ARG-SPEC)
                 (T `',ARG-SPEC)))
          ((EQ? (FUNCTOR:TERM ARG-SPEC) 'EVAL) (ARG:TERM 1 ARG-SPEC))
          (T `(CONS ,(MAKE-CONSTRUCTOR (CAR ARG-SPEC))
                    ,(MAKE-CONSTRUCTOR (CDR ARG-SPEC)))))))


(DEFINE (UNIQ-VAR CLAUSE)
 (LABELS
```

```
   (((UNIQ-VAR1 CLAUSE LIST)
             (COND ((NULL? CLAUSE) LIST)
                   ((ATOM? CLAUSE)
                    (COND ((IS-VAR-ATOM? CLAUSE)
                           (COND ((MEMQ? CLAUSE LIST) LIST)
                                 (T (CONS CLAUSE LIST))))
                          (T LIST)))
                   (T (SET LIST (UNIQ-VAR1 (CAR CLAUSE) LIST))
                      (SET LIST (UNIQ-VAR1 (CDR CLAUSE) LIST))))))
    (UNIQ-VAR1 CLAUSE ())))

(DEFINE (IS-VAR-ATOM? ATOM)
    (AND (SYMBOL? ATOM) (EQ? (STRING-ELT (SYMBOL->STRING ATOM) 0) #?)))

(DEFINE (TLOG-FUNC INVOCATION)
    (SYMBOLCONC (CAR INVOCATION) '/ (- (LENGTH INVOCATION) 1))))

; LOCAL? looks at the head of a clause to see if a variable
; is local. A local variable is one which occurs at least once
; NOT inside a compound term.
(DEFINE (LOCAL? VAR HEAD)
    (COND ((NULL? HEAD) NIL)
          ((EQ? VAR (CAR HEAD)))
          ((LOCAL? VAR (CDR HEAD)))
          (T NIL)))

(DEFINE (FIND-ARG VAR HEAD PATTERN)
    (COND
          ((NULL? HEAD) (ERROR "ARG SPEC ERROR"))
          ((NULL? PATTERN) (ERROR "ARG SPEC ERROR"))
          ((EQ? VAR (CAR HEAD))
           (CAR PATTERN))
          (T (FIND-ARG VAR (CDR HEAD) (CDR PATTERN)))))

(DEFINE (HAS-CUT? GOALS)
    (COND ((NULL? GOALS) NIL)
          ((ALIKEV? (CAR GOALS) '(!)))
          (T (HAS-CUT? (CDR GOALS)))))

(DEFINE (FIND-BEFORE-CUT GOALS)
    (LABELS ((LST NIL)
             ((FIND-B1 GOALS)
              (COND ((ALIKEV? (CAR GOALS) '(!)) T)
                    (T (PUSH LST (CAR GOALS))
                       (FIND-B1 (CDR GOALS))))))
            (FIND-B1 GOALS)
            (REVERSE! LST)))

(DEFINE (FIND-AFTER-CUT GOALS)
    (COND ((ALIKEV? (CAR GOALS) '(!)) (CDR GOALS))
          (T (FIND-AFTER-CUT (CDR GOALS)))))
```

```
(DEFINE (MAKE-EXPR EXPR)
    (COND ((ATOM? EXPR)
           (COND ((IS-VAR-ATOM? EXPR) `(DE-REF ,EXPR))
                 (T `',EXPR)))
          ((EQ? (FUNCTOR:TERM EXPR) 'EVAL) (ARG:TERM 1 ARG-SPEC))
          (T `(,(FUNCTOR:TERM EXPR) ,@(MAP MAKE-EXPR (ARGLIST:TERM EXPR))))))
```

The unifer

```
(HERALD UNIFY (ENV T))

; This file contains the unifier for the TLOG compiler.

; Global reset list is used to reset logical variables after a cut.
(LSET *GLOBAL-RESET-LIST* NIL)

; *RESET-LIST* is used to keep track of variables which are bound
; by UNIFY1

(DEFINE (UNIFY TERM1 TERM2 CONTINUATION)
 (LABELS
  ((*RESET-LIST* NIL)
   ((UNIFY1 TERM1 TERM2)
    (SET TERM1 (DE-REF TERM1))
    (SET TERM2 (DE-REF TERM2))
    (COND ((EQ? TERM1 TERM2))
          ((PROVAR? TERM1)
           (SET (PROVAR-VALUE TERM1) TERM2)
           (PUSH *RESET-LIST* TERM1)
           (PUSH *GLOBAL-RESET-LIST* TERM1)
           T)
          ((PROVAR? TERM2)
           (SET (PROVAR-VALUE TERM2) TERM1)
           (PUSH *RESET-LIST* TERM2)
           (PUSH *GLOBAL-RESET-LIST* TERM2)
           T)
          ((NULL? TERM1)
           (COND ((NULL? TERM2))
                 (T NIL)))
          ((NULL? TERM2) NIL)
          ((ATOM? TERM1) (EQ? TERM1 TERM2))
          ((LIST? TERM2)
           (COND ((UNIFY1 (CAR TERM1) (CAR TERM2))
                  (UNIFY1 (CDR TERM1) (CDR TERM2)))
```

```
                (T NIL))))))
    (COND ((UNIFY1 TERM1 TERM2)
           (APPLY CONTINUATION ()))))
    (ITERATE RESET ((LST *RESET-LIST*))
        (COND (LST
                (SET (PROVAR-VALUE (CAR LST)) '*UNDEF*)
                (POP *GLOBAL-RESET-LIST*)
                (RESET (CDR LST)))
              (T NIL)))))

(DEFINE (DE-REF TERM)
    (COND ((NULL? (PROVAR? TERM)) TERM)
          ((EQ? (PROVAR-VALUE TERM) '*UNDEF*) TERM)
          (T (DE-REF (PROVAR-VALUE TERM)))))

(DEFINE (RESET-GLOBAL-LIST OLD-RESET-LIST)
    (ITERATE RESET ()
        (COND ((NULL? (EQ? *GLOBAL-RESET-LIST* OLD-RESET-LIST))
               (SET (PROVAR-VALUE (POP *GLOBAL-RESET-LIST*))
                    '*UNDEF*)
               (RESET))
              (T NIL))))
```

The T to Tlog interface code

```
(HERALD INTERFACE (ENV T))
; This file contains the code for T calling TLOG.

; The syntax for the TLOG-LOOP is:
;       (TLOG-LOOP name form . code)
; name - any name the user likes
; form - a TLOG goal
; code - a list of s-exprs in T.
;
; If 'name' is invoked as a procedure, the next solution to the
; TLOG goal is found. Otherwise the result of the last expression
; is returned. All the logical variables in the 'form' can be
; accessed by 'code'.

(DEFINE-SYNTAX (TLOG-LOOP NAME FORM . CODE)
    `(LET (,@(MAP
               (LAMBDA (VAR) `(,VAR (MAKE-PROVAR)))
               (UNIQ-VAR FORM))
           (OLD-RESET-LIST *GLOBAL-RESET-LIST*)
           (CODE-VALUE NIL))
```

```
        (SET CODE-VALUE (CATCH *CUT*
          ,@(CDDR (MAKE-GOAL-CONTINUATION FORM
                    `(LAMBDA () (CATCH *FAIL*
                                  (LET ((,NAME
                                          (LAMBDA () (*FAIL* NIL))))
                                    ,@(CODE-CONT CODE))))))))
        (RESET-GLOBAL-LIST OLD-RESET-LIST)
        CODE-VALUE)
)

; CODE-CONT makes the list of expressions for the goal continuation.
; A list of expressions, (EXPR1 EXPR2 ... EXPRn) becomes,
; (EXPR1 EXPR2 ... (*CUT* EXPRn))

(DEFINE (CODE-CONT CODE)
    (COND ((NULL? (CDR CODE)) `((*CUT* ,(CAR CODE))))
          (T (CONS (CAR CODE) (CODE-CONT (CDR CODE))))))
```

Appendix B: A Phrasal Parser in Tlog

This section is an extended example of how to use the Tlog with T. The interface issues covered are:

how to write Tlog predicates in T, predicates which can be backtracked over,
how to call T code from Tlog, and
how to call Tlog predicates from T.

The parser uses a database of patterns which are mapped to conceptual structures. An example entry in the database is,

```
(?X GAVE ?Y ?Z) ==> (ATRANS actor ?X
                           to ?Y
                           obj ?Z
                           time PAST)
```

ATRANS is the representation for **give**. The database of patterns is maintained by T. It is indexed by the non-variable entries of the patterns, **GIVE** in the example above. The T functions used to access the data base are,

(FETCH-PATTERN *data-base sentence*)
-fetches a list of possible patterns for *sentence*

(NEXT:PATTERN-BAG *pattern-bag*)
-the next element of a *pattern-bag* as returned from FETCH-PATTERN

(REST:PATTERN-BAG *pattern-bag*)
-the result of removing the next element of *pattern-bag*

(TEMPLATE:PATTERN *pattern*)
-the actual list structure for the pattern

(FIRST-WORD:TEMPLATE *template*)
-the first non-variable member of *template*

(HEAD:TEMPLATE *template*)
-the elements of *template* before the first word, NIL if no first word

(TAIL:TEMPLATE *template*)
-the elements of *template* after the first word, *template* if no first word

(FIND-HEAD *sentence word*)
- the words in *sentence* before *word*

(FIND-TAIL *sentence word*)
- the words in *sentence* after *word*

To get patterns from the database, we need a predicate which will yield a new pattern every time it is FAILed and which will fail only when no more patterns are available.

```
(DEFINE-Tlog (DATABASE SENTENCE PATTERN)
    (LET ((PATTERN-BAG (FETCH-PATTERNS *PATTERNS* (DE-REF SENTENCE)))
          (NEXT-PATTERN NIL))
      (ITERATE Do-Loop ()
          (COND ((NULL? PATTERN-BAG) nil)
                (T (SET NEXT-PATTERN (NEXT:PATTERN-BAG PATTERN-BAG))
                   (SET PATTERN-BAG (REST:PATTERN-BAG PATTERN-BAG))
                   (UNIFY PATTERN
                          NEXT-PATTERN
                          *CONTINUATION*)
                   (Do-Loop))))
           (SET (PROVAR-VALUE PATTERN) '*UNDEF*)))
```

The predicate defined by here, DATABASE/2, uses FETCH-PATTERNS to get a list of patterns which may be applicable to SENTENCE. It makes a call to UNIFY to match whatever the user passed as the PATTERN parameter (usually an unbound variable) to the next pattern. Each time *CONTINUATION* fails, a new pattern is unified with PATTERN. An example of using DATABASE/2 in the Tlog predicate PARSE/2 given below.

```
(define-Tlog (parse sentence result)
    (:- (parse ?sentence ?result)
        (AND (database ?sentence ?pattern)
             (IS ?template (template:pattern ?pattern))
             (match ?template ?sentence () ?bindings ())
             (IS ?result (instantiate ?pattern ?bindings))))
)
```

PARSE/2 uses **DATABASE/2** to get a stream of possible patterns. It uses the selector **TEMPLATE:PATTERN** to get the actual list of variables and words that make up the phrase. The **MATCH/5** predicate determines if the sentence is matched by a particular template. It takes as arguments, the template **?template**, the sentence, **?sentence**, the old variable bindings, **()** in this case, and it returns the new bindings, **?bindings**, and the portion of the sentence not matched, constrained to be **()** in this case. **PARSE/2** uses the T function **INSTANTIATE** to build the actual representation of the phrase parsed.

Next we will look at **INSTANTIATE** a T function called from Tlog. **INSTANTIATE** takes a pattern, as returned from **DATABASE/2** and constructs the representation according to a **LAMBDA** expression which is part of the pattern. **INSTANTIATE** has to look up the bindings for the variables on the binding list which is passed to it. Unfortunately this binding list was created by Tlog functions and may contain bound variables. Structures which contain bound variables are transparent to Tlog code but not to T code. The solution is to use a Tlog predicate to look up variables on the binding list. This demonstrates how to use the **TLOG-LOOP** construct.

```
(DEFINE (INSTANTIATE PATTERN BINDINGS)
   (LET ((VAR-LIST (VAR-LIST:PATTERN PATTERN))
         (LAMBDA-EXPR (LAMBDA:PATTERN PATTERN))
         (ARG-LIST nil))
    (Tlog-Loop NEXT (look-up-vars (EVAL var-list) (EVAL bindings) ?result)
               (PUSH ARG-LIST (DE-REF ?result))
               (NEXT))
    (APPLY LAMBDA-EXPR (REVERSE! ARG-LIST))))

(define-Tlog (look-up-vars var-list bindings result)
    (:- (look-up-vars (?var . ?rest) ?bindings ?result)
        (look-up-binding ?var ?bindings ?result))
    (:- (look-up-vars (?var . ?rest) ?bindings ?result)
        (look-up-vars ?rest ?bindings ?result))
)
```

```
(define-Tlog (look-up-binding var bindings result)
    (:- (look-up-binding ?var ((?var . ?result) . ?rest) ?result) (!))
    (:- (look-up-binding ?var (?head . ?rest) ?result)
        (look-up-binding ?var ?rest ?result))
)
```

First we will examine the Tlog predicates. **LOOK-UP-VARS/3** is a predicate which looks up a variable from **VAR-LIST** on the binding list. Each time it is backtracked, it returns the binding of the next variable in **VAR-LIST**. It uses **LOOK-UP-BINDING** to find the binding of a single variable. **INSTANTIATE** uses the **TLOG-LOOP** construct to execute **LOOK-UP-VARS/3**. Remember, **TLOG-LOOP** makes a continuation out of the body of loop, in this case,

```
(PUSH ARG-LIST (DE-REF ?result))
(NEXT)
```

and calls the goal,

```
(look-up-vars (EVAL var-list) (EVAL bindings) ?result)
```

with that continuation. Because **NEXT** is the label of the loop, executing it will fail the continuation and cause the goal to be backtracked. The result of this loop is that all the bindings get pushed onto **ARG-LIST** in the reverse order they were in **VAR-LIST**. In the last expression of **INSTANTIATE**, **LAMBDA-EXPR** is applied to the argument list to yield the result of the parse.

The code for the **MATCH/5** and **MATCH-SEQ/5** predicates is given below. It implements a simple recursive string matching algorithm. Since it uses the selectors given above, the only feature of Tlog which it uses heavily is backtracking. No further explanation of this code is given since it does not highlight any additional feature of Tlog.

```
(define-Tlog (match template sentence b-in b-out fragment)
    (:- (match () ?f ?b ?b ?f) (!))
    (:- (match () () ?b ?b ()) (!))
    (:- (match (?word) (?word) ?b ?b ()) (!))
    (:- (match (?word) (?word . ?fragment) ?b ?b ?fragment) (!))
    (:- (match ?template ?sentence ?b-in ?b-out ?fragment)
        (AND (IS ?first-word (first-word:template ?template))
             (IS ?first-word ()) (!)
             (match-seq ?template ?sentence ?b-in ?b-out ?fragment)))
    (:- (match ?template ?sentence ?b-in ?b-out ?fragment)
```

```
        (AND (IS ?first-word (first-word:template ?template))
             (IS ?head-temp (head:template ?template))
             (IS ?head-sen (find-head ?sentence ?first-word))
             (match-seq ?head-temp ?head-sen ?b-in ?b2 ())
             (IS ?tail-temp (tail:template ?template))
             (IS ?tail-sen (find-tail ?sentence ?first-word))
             (match ?tail-temp ?tail-sen ?b2 ?b-out ?fragment)))
)

(define-Tlog (match-seq var-seq sentence b-in b-out fragment)
    (:- (match-seq () ?f ?b ?b ?f) (!))
    (:- (match-seq () () ?b ?b ()) (!))
    (:- (match-seq (?var . ?rest) ?sentence ?b-in ?b-out ?frag)
        (AND (database ?sentence ?pattern)
             (IS ?template (template:pattern ?pattern))
             (match ?template ?sentence ?b-in ?b2 ?rest-sen)
             (IS ?value (instantiate ?pattern ?b2))
                 (IS ?name (name:var ?var))
             (match-seq ?rest ?rest-sen
                        ((?name . ?value) . ?b-in) ?b-out ?frag)))
)
```

References

1. Abelson, H., Sussman, G. J., and Sussman, J., "Structure and Interpretation of Computer Programs", MIT Press, 1985.

2. Kahn, K. M. and Carlsson, M., "How to Implement Prolog on a Lisp Machine" in *Implementations of Prolog*, J. A. Campbell (Ed), Ellis Horwood Ltd., 1984.

3. Mellish, C. and Hardy, S., "Integrating Prolog in the POPLOG Environment", in *Implementations of Prolog*, J. A. Campbell (Ed), Ellis Horwood Ltd., 1984.

4. Nilsson, M, "The World's Shortest PROLOG Interpreter?", in *Implementations of Prolog*, J. A. Campbell (Ed), Ellis Horwood Ltd., 1984.

5. Read, W., Stoll, K., and Fuenmayor, M., "The TLOG Manual", Tools Note, UCLA AI Laboratory, 1984.

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

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

8. Rees, J. A., Personal communication, November 9, 1984.

9. Robinson, J. A. and Sibert, E. E., "LOGLISP: An Alternative to PROLOG", in *Machine Intelligence 10*, Ellis Horwood Ltd, 1982.

10. Steele, G. L., *COMMON LISP, the Language*, Digital Press, 1984.

11. Warren, D., *Implementing Prolog*, Research Report 33, Dept of A.I., University of Edinburgh, 1977.

12. Weinreb, D. and Moon, D., *LISP Machine Manual*, MIT AI Laboratory.