Linear and Temporal Logic Programming Language

ARNOŠT VEČERKA
Department of Computer Science
Palacky University
Tomkova 40, 77900 Olomouc
CZECH REPUBLIC
http://www.inf.upol.cz/vecerka/

Abstract: - Recent logic calculi open new possibilities for logic programming. Apparently the most important of them is a linear logic which makes possible to solve problems by resources treatment. The linear logic has already been used for several proposals of logic programming language. But authors of these languages mostly focused on the proposal of language and somewhat turned aside its efficient implementation. Programs written in these languages are mostly interpreted by interprets written in Prolog. This is relatively simple method, but performance of such system is poor and limits usability of these languages. The only useful language is LLP [3] which has its compiler system. Programs written in LLP are compiled into an internal form and then interpreted by extended WAM. In this paper we describe another programming language which has its compiler. This language is based on a linear logic and on a temporal logic as well.

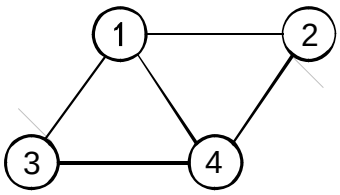
Key-Words: Linear logic, Fact, Rule, Temporal logic, Transition relation, Model

1 Introduction

A traditional logic programming language provides two types of logic formulas:

- Facts
- Rules

The facts describe individuals or describe a relation that holds between individuals. An example is a graph. The graph is defined by nodes and edges. The nodes are individuals and can be represented by unary facts in a logic program. The edges mean relation of adjacency between nodes and can be represented by binary facts in the logic program.
The following graph



can be in classic logic programming language Prolog represented by facts

```
node(1). node(2). node(3). node(4).
edge(1,2).  edge(1,3). edge(1,4).
edge(2,4).  edge(3,4).
```

Prolog is a programming language based on a standard predicate logic and every formula in Prolog program can be used a number of time during program execution. It is disadvantageous for problems in which facts represent resources.

An example can be a graph problem in which we search an important set of nodes (dominating set, independent set, and other). Every node of the graph can be put to the set only once. If we wrote a program for such problem in Prolog we should use a list for storing nodes already added to the set. Before adding next node to the set we need to seek through the list in order to ensure that the node is not in the set already. It makes the program more complex and frequent searching through the list slows down a program execution.
It is far better to use a programming language based on a linear logic calculus for such problems. In the following sections we shall show that the linear logic makes possible to write programs that have more expressive logical structure and less frequently use lists.

2 The programming language

The programming language described in this paper is based on a linear logic calculus created by J. Y. Girard [1]. In the language that is described in this paper the two following linear logic connectives are implemented.

- The connective of multiplicative conjunction $\otimes$, in the language it is written by the symbol *.
- The connective of additive disjunction $\oplus$, in the language it is written by the symbol +.

The connectives can be used in rules and in a goal.

2.1 Linear formulas

In a linear logic calculus formulas are linear by default.
The property of linearity results from axioms of the calculus and means that every formula can be used only once in an inference process. But for a logic program, whose most formulas are usually used repeatedly during execution, it is not convenient. Hence in the proposed language the formula is not linear by default. The formula that is linear is explicitly marked by the keyword *lin* at its beginning. Linear formulas are facts usually. In the described language the linear formula can be also a rule.
The following simple program and goal demonstrate usage of mentioned connectives and usage of a linear formula.

```
Program: a(1).
         lin a(2).
Goal:    (a(X) * a(Y)) + a(Z).
```

The connective of additive disjunction + means that either the first subgoal *a(X)*a(Y)* or the second subgoal *a(Z)* must be executed.
The connective of multiplicative conjunction * means that both the two subgoals *a(X)* and *a(Y)* must be executed.
The first fact *a(1)* is not linear and can be used a number of time. The second fact *a(2)* is linear and can be used only once. There are five possible executions of the goal:

1. X = 1, Y = 1
2. X = 1, Y = 2
3. X = 2, Y = 1
4. Z = 1
5. Z = 2

2.2 Brief description of the language

As the language is rather complex only the most important its parts are described in this section.
A fact has form similar to its form in Prolog. It is a predicate. Its syntax is:

[lin] *Predicate*.

A rule has the following syntax:

[lin] *Predicate* :- *Formula*.

The part *Formula* is defined by the following rules:

Predicate is *Formula*
Condition is *Formula*
Unification is *Formula*
If α and β are the *Formulas* the *Formula* also is:
$\alpha * \beta$
$\alpha + \beta$
(α)
once α
not α

The syntax of the condition is:

(*Conditional_Expression*)

The unification can have the following two forms of syntax:

(*Variable* = *Expression*)
($Variable_1$ = $Variable_2$)

The conditions and unifications are enclosed in parentheses in order that a compiler can easily recognise them during program compilation.
The keyword *once* has similar purpose as the Prolog cut predicate (!). The formula which is after this keyword is executed only once. It prevents another execution of the formula after a backtracking.
The *not* is a negation-as-failure. It has the same meaning as the *not* in Prolog.
A goal has similar syntax as the *Formula* at right side of rule, but the goal can not contain conditions and unifications.
As an example of program written in the proposed language we present program solving knight's tour problem. The goal is to find a tour of chess knight on a chess board starting from arbitrary square. The knight must visit every square of board exactly once.
The squares of board are represented by linear facts. A move of knight to the next square is in the program written by a rule. At right side of the rule there is a predicate with which is unified the fact that represents visited square. The fact is linear so that the knight cannot visit the same square twice.

Program 1

```
n=8.
n1=n*n-1.
lin board(1;n,1;n).
goal (I,J) :- board(I,J) * tour(n1,I,J).
tour(0,_,_).
tour(N,I,J) :- ((I1=I-2) * ((J1=J-1) + (J1=J+1)) +
                (I1=I-1) * ((J1=J-2) + (J1=J+2)) +
                (I1=I+1) * ((J1=J-2) + (J1=J+2)) +
                (I1=I+2) * ((J1=J-1) + (J1=J+1))) *
               board(I1,J1) *
               (N1=N-1) * tour(N1,I1,J1).
```

Each square of board is represented by one linear fact. Hence a standard 8×8 board is represented by 64 facts. In the proposed language there is a possibility to write facts as an array to avoid writing so many facts. The following array of linear facts

```
lin board(1;8,1;8).
```

is equivalent to 64 linear facts

```
lin board(1,1). lin board(2,1). ... lin board(8,8).
```

The language also has possibility to define constants so that writing of program may be easier. A definition of constant has the syntax:

name = *constant_expression*.

The constant name has the same syntax as a name of predicate. The definition of constant can also be in a goal. This definition has precedence before the definition of constant with the same name in program. For example the goal

```
n=6, goal(1,1).
```

means that the previous program is executed for a board 6×6 and the tour of knight starts on a square in corner of board.

2.3 Temporal logic

In the previous section we showed that the linear logic makes possible to replace usage of list by more effective facts. Nevertheless, when number of facts is considerable it also has an unfavourable influence on execution time.

For example, the previous program contains 64 linear facts which represent squares of board. When the knight steps upon any square the corresponding linear fact is removed from formulas of program. Hence the number of available facts is from 64 to 1 in accordance with the length of up to now stepped part of tour. A temporal logic model can significantly decrease the number of available facts in problems like this.

In a classical temporal logic the model is defined as the structure

$M = (S, \rightarrow, L)$

where S is a set of states, $\rightarrow$ is a transition relation (a binary relation on S, saying how to move from current state to next state), L is a function which associates each state with the set of atomic formulas which are true at that state.

A linear logic is a logic of resources and this function will determine available formulas in individual states in place of true formulas.

A transition relation formula in the proposed language has the syntax:

Predicate -> *Formula*.

The part *Formula* of transition relation is defined by the following rules:

Predicate is *Formula*,

If α and β are the *Formulas* the *Formula* also is:

$\alpha * \beta$

α , β

Condition * α

Unification * α

(α)

The comma (,) is a separator in a list of formulas. For example the relation formula

Predicate -> $Formula_1$, $Formula_2$, $Formula_3$.

is equivalent to the following three relation formulas

Predicate -> $Formula_1$.
Predicate -> $Formula_2$.
Predicate -> $Formula_3$.

An example is a transition relation for a temporal logic model of the graph mentioned in introductory section.

```
lin node(1;4).
node(1) -> node(2), node(3), node(4).
node(2) -> node(1), node(4).
node(3) -> node(1), node(4).
node(4) -> node(1), node(2), node(3).
```

It is also possible to use facts for a definition of transition relation. An example is an alternative method for defining a transition relation for the previous graph.

```
lin node(1;4).
edge(1,2). edge(1,3). edge(1,4).
edge(2,4).  edge(3,4).
node(I) -> (edge(I,J), edge(J,I)) * node(J).
```

The temporal logic contains three operators that have meanings *always*, *eventually*, and *next*. In the proposed language the operator *next* is implemented from these operators. This operator can be used at a predicate in rule or in goal. The operator designates that only the fact which is from next state with regard to current state can be unified with this predicate.

It is necessary to determine which of states will be an initial one. In the proposed language there is the keyword *init* for this purpose. This keyword can be used at a predicate. A state with the fact which will be unified with this predicate will be the initial state.

An example how to use the keyword *init* and operator *next* is the following program. The program searches for Hamiltonian cycle in a graph.

Program 2

```
n = number_of_nodes.
lin node(1;n).
lin node(1).
… facts for edges ….
node(I) -> (edge(I,J), edge(J,I)) * node(J).
n1 = n-1.
cycle :- init node(1) * adjac(n1).
adjac(0) :- next node(1).
adjac(N) :- next node(X) * (N1=N-1) * adjac(N1).
```

Hamiltonian cycle is a closed path which contains all graph nodes. The program searches it as a path from node *1* to node *1*. The fact *node(1)* is used two times. That is why one more linear fact *lin node(1)* is added to facts in the program.

As a complete example of using temporal logic model we again present program solving knight's tour problem.

Program 3

```
n=8.
lin board(1;n,1;n).
board(I,J) -> ((I1=I-2) * ((J1=J-1), (J1=J+1)),
               (I1=I-1) * ((J1=J-2), (J1=J+2)),
               (I1=I+1) * ((J1=J-2), (J1=J+2)),
               (I1=I+2) * ((J1=J-1), (J1=J+1))) *
              board(I1,J1).
n1=n*n-1.
goal (I,J) :- init board(I,J) * tour(n1).
tour(0).
tour(N) :- next board(I,J) * (N1=N-1) * tour(N1).
```

The main part of the program is a transition relation $\rightarrow$. A compiler takes this relation and the array of facts *board(1;n,1;n)* and creates a temporal logic model. This model contains n^2 states. In each of these states one linear fact *board(i,j)* is available.

3 The compiler

The compiler is a stand-alone program. It has three parts. The first part translates a goal formula and formulas of logic program into an internal form. The internal form has different structure than internal form used by WAM, which is an abstract machine designed for execution of Prolog programs. It has been also used for a construction of compiler system for a linear logic programming language LLP [3]. Authors of this language extended the WAM in order that it might execute linear formulas. Great advantage of this approach is a utilization of memory architecture, unification, backtracking, and other functionalities of WAM. It significantly simplifies a construction of compiler, but the WAM is designed for Prolog and its modification for a programming language based on linear logic may be less effective. That is why a unique internal form has been developed for the proposed language. The internal form has a simple tree-like structure. As this internal form retains a structure of logic formulas it is also convenient for a program tracing during execution.

The second part of compiler adds supplementary pointers to the internal form. The purpose of these pointers is to make a direct transition from a currently executed node to a next node to be executed. The pointers considerably reduce a traversal of internal form tree during a program execution. This part of compiler also creates list of formulas for every predicate which is on right side of a rule or in a goal. This list contains all facts and rules which can be unified with the predicate. During program execution the list makes selection of formula for unification with the predicate quick. It is simply taken from the list.

In this stage of compilation a temporal logic model is also created if the program contains transition relations.

The last part of compiler is an interpreter of the internal form that executes the program. The interpreter has been proposed with care because its quality affects execution speed. The interpreter is a machine with three stacks. The first stack is a main stack and contains runtime data of executed nodes. The second stack is for variables. If an instance of rule or fact is created its variables are located at this stack. The third stack contains data about performed unifications. These data make possible undo unifications during backtracking.

The compiler is completely written in C++. Its current version has a graphical user interface in operating system Windows. The compiler is available with some example programs at [2].

4 Performance evaluation

Performance of the compiler has been tested on knight's tour problem. This problem has been solved by five various programs.

- The first program is written in the described language and it is the example program 1 presented in subsection 2.2. The language described in this paper has name LTLL (Linear and Temporal Logic Language).
- The second program is also written in LTLL. This program uses a temporal logic model. It is the example program 3 presented in subsection 2.3.
- The third program is written in linear logic programming language LLP. The language has been proposed and its compiler system has been developed at Kobe University in Japan [3].
- The fourth program is written in Prolog. SWI-Prolog was used for its execution. It is very good implementation of Prolog and is available at [7].
- The fifth program is written in LTLL. It is written by the same manner as the Prolog program and uses neither linear formulas nor temporal logic model. The program uses a list with visited squares for checking if a particular square has not been yet visited. It is the following program.

Program 4

```
n = 8.
n2 = n*n.
n1 = n2-1.
goal(I,J,L) :- tour(n1,I,J,[I,J|L]).
tour(0,_,_,_).
tour(N,I,J,L) :- ((I>2)*(I1=I-2) * ((J>1)*(J1=J-1) +
                  (J<n2)*(J1=J+1)) +
                  (I>1)*(I1=I-1) * ((J>2)*(J1=J-2) +
                  (J<n1)*(J1=J+2)) +
                  (I<n2)*(I1=I+1) * ((J>2)*(J1=J-2) +
                  (J<n1)*(J1=J+2)) +
                  (I<n1)*(I1=I+2) * ((J>1)*(J1=J-1) +
                 (J<n2)*(J1=J+1))) *
                 not memb(I1,J1,L) * (N1=N-1) *
                tour(N1,I1,J1,[I1,J1|L]).
memb(X,Y,[X,Y|_]).
memb(X,Y,[_,_|Z]) :- memb(X,Y,Z).
```

A comparison of this program with the program written in Prolog is interesting because the LTLL compiler is based on a machine which is very different from the abstract machine WAM used for Prolog compiler.

It is also evident that this program is more complex than the program based on linear logic or the program based on linear and temporal logic.

Language	Time (seconds)	Linear logic	Temporal logic
LTLL	31	yes	-
LTLL	7	yes	yes
LLP	109	yes	-
Prolog	1155	-	-
LTLL	468	-	-

All executions have been made on a computer with processor P4, 2.8 GHz.

The programs written in LTLL and the program written in Prolog are available at [2]. The program written in LLP is available at [3].

5 Conclusion

The language presented in this paper is interesting in several aspects. The first one is a linear logic calculus on which the language is based. As the test of performance has showed linear formulas are very useful for resource-oriented problems. Execution time of the example program with linear formulas is markedly less than time of the equivalent program without linear formulas.

Also the idea to use a temporal logic for the language proposal has proved effective. The temporal logic model in the example program has significantly speeded up execution. In addition, the language LTLL, described in the paper, is evidently the first logical programming language that makes possible to use a temporal logic model.

Another important comparison is between compilers. SWI-Prolog compiler is based on the abstract machine WAM. LLP compiler system is also based on WAM. LTLL compiler is based on a unique machine which has been proposed for this language. The comparison between Prolog program and equivalent LTLL program (the program 4 without linear formulas and temporal logic model) has showed that LTLL program is more than two times faster. The comparison between LLP program and equivalent LTLL program (the program 1 with linear formulas) has showed that LTLL program is approximately three times faster. Apparently the LTLL compiler is effectively proposed.

References:

[1] Girard, Jean-Yves. Linear Logic: Its Syntax and Semantics. Available at http://iml.univ-mrs.fr/~girard/Articles.html.

[2] Linear and Temporal Logic Programming Language. Available at http://www.inf.upol.cz/vecerka/.

[3] LLP: A Linear Logic Programming Language and its Compiler System. Available at http://bach.istc.kobe-u.ac.jp/llp/.

[4] Lolli: A Linear Logic Programming Language. Available at http://www.lix.polytechnique.fr/~dale/lolli/.

[5] Lygon: Logic Programming with Linear Logic. Available at http://www.cs.rmit.edu.au/lygon/.

[6] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. London: Springer-Verlag, 2001.

[7] SWI-Prolog. Avaliable at http://www.swi-prolog.org.

[8] The Forum Specification Language. Available at http://www.lix.polytechnique.fr/~dale/forum/.