A COMPILER SYSTEM OF A LINEAR LOGIC PROGRAMMING LANGUAGE

NAOYUKI TAMURA and YUKIO KANEDA
{tamura,kaneda}@seg.kobe-u.ac.jp
Department of Computer and Systems Engineering,
Faculty of Engineering, Kobe University
1-1 Rokkodai, Nada, Kobe 657 Japan

ABSTRACT

Linear logic developed by J.-Y. Girard can be described as a logic of resources. There have been several proposals for logic programming language based on linear logic: LO, LinLog, ACL, Lolli, Lygon, and Forum. Lolli and Lygon are implemented as interpreter systems (on SML and λProlog for Lolli, on Prolog for Lygon). But, none of them have been implemented as a compiler system.

This paper describes a compiler system of a linear logic programming language called LLP. New features of LLP with various example programs are also shown. LLP is a superset of Prolog and a subset of Lolli. LLP programs are compiled into LLPAM (LLP Abstract Machine) code, which is an extended WAM (Warren Abstract Machine) designed for LLP.

Keywords: Logic programming, Linear logic, Warren Abstract Machine

1 INTRODUCTION

Linear logic[1] developed by J.-Y. Girard [3] can be described as a logic of resources. There have been several proposals for logic programming language based on linear logic: LO [2], LinLog [1], ACL [5], Lolli[2] [4], Lygon[3] [7], and Forum[4] [6]. Lolli and Lygon are implemented as interpreter systems (on SML and λProlog for Lolli, on Prolog for Lygon). But, none of them have been implemented as a compiler system.

We developed a compiler system of a linear logic programming language called LLP [8]. LLP programs are compiled into LLPAM (LLP Abstract Machine) code, which is an extended WAM (Warren Abstract Machine) designed for LLP.

[1] `http://www.csl.sri.com/linear/sri-csl-ll.html`
[2] `http://www.cs.hmc.edu/~hodas/research/lolli/`
[3] `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`
[4] `http://www.cis.upenn.edu/~dale/forum/`

LLP is a superset of Prolog and a subset of another linear logic programming language Lolli. Prolog programs can be executed without any modification (however, some built-in predicates are missing in our current implementation).

In this paper, we describe syntax and programming features of LLP. Various example programs are also included to show that LLP is suitable for describing many kind of constraint satisfaction problems.

2 SYNTAX

A program of LLP is a sequence of clauses (in fact, they are not clauses in the traditional sense). The syntax of clauses can be written in BNF as follows.

$$
\begin{array}{lcl}
C & ::= & A\texttt{.} \mid A\texttt{:-}G\texttt{.} \\
G & ::= & \texttt{true} \mid \texttt{top} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid \\
 & & G_1\texttt{;}G_2 \mid \texttt{!}G \mid R\texttt{-<>}G \\
R & ::= & S \mid \texttt{!}S \mid R_1\texttt{,}R_2 \\
S & ::= & A \mid G\texttt{-<>}S \mid S_1\texttt{\&}S_2
\end{array}
$$

C, G, R, S, and A represent "clause", "goal", "resource", "selective resource", and "atomic formula" respectively.

Since the following equivalence relations are valid in linear logic, LLP compiler translates the left-hand side resource formula by the right-hand side.

$$
\begin{array}{rcl}
\texttt{!}(S_1\texttt{\&}S_2) & \equiv & (\texttt{!}S_1)\texttt{,}(\texttt{!}S_2) \\
G_1\texttt{-<>}(G_2\texttt{-<>}S) & \equiv & (G_1\texttt{,}G_2)\texttt{-<>}S \\
G\texttt{-<>}(S_1\texttt{\&}S_2) & \equiv & (G\texttt{-<>}S_1)\texttt{\&}(G\texttt{-<>}S_2)
\end{array}
$$

Therefore, resource formulas can be rewritten as follows.

$$
\begin{array}{rcl}
R & \equiv & S \mid \texttt{!}A \mid \texttt{!}(G\texttt{-<>}A) \mid R_1\texttt{,}R_2 \\
S & \equiv & A \mid G\texttt{-<>}A \mid S_1\texttt{\&}S_2
\end{array}
$$

The order of the operator precedence is "`:-`", "`;`", "`&`", "`,`", "`-<>`", "`!`" from wider to narrower. Newly introduced symbols to LLP (compared with Prolog) are

"&" (additive conjunction), "-<>" (linear implication), "!" (bang), and "top". LLP program not including these symbols can be executed as a Prolog program.

3 RESOURCE HANDLING

In this section, we give an intuitive explanation of resource programming features of LLP. We assume readers have a basic knowledge on Prolog.

The biggest difference of LLP is its resource consciousness. LLP system maintains a resource table, to which resources can be dynamically added or deleted (consumed) during the execution.

3.1 Resource Addition

Resource is added by the execution of a goal formula R-<>G. For example, the following query adds a resource `r(1)` to the resource table, then executes a goal `r(X)` which consumes `r(1)` by letting `X` = 1.

```
?- r(1) -<> r(X).
```

In the execution of the goal R-<>G, all resources in R should be consumed up during the execution of G. For example, the following query fails since `r(1)` is not consumed.

```
?- r(1) -<> true.
```

Resource formula G-<>A is used to represent a rule-type resource. The goal G is executed on resource consumption. The following query displays `1`.

```
?- (write(X) -<> r(X)) -<> r(1).
```

Resource formula R_1,R_2 is used to add multiple resources. The following query adds resources `r(1)` and `r(2)`, then consumes both of them by letting `X` = 1 and `Y` = 2, or `X` = 2 and `Y` = 1.

```
?- (r(1), r(2)) -<> (r(X), r(Y)).
```

Resource formulas !A and !(G-<>A) mean infinite resources, that is, they can be consumed arbitrary times (including zero times). The following query succeeds by letting `X` = 1 or `X` = 2.

```
?- (!r(1), !r(2)) -<> (r(X), r(X)).
```

Resource formula S_1&S_2 is used to represent a selective resource. For example, when `r(1)&r(2)` is added as a resource, either `r(1)` or `r(2)` can be consumed, but not both of them. The following query succeeds by letting `X` = 1 or `X` = 2.

```
?- (r(1) & r(2)) -<> r(X).
```

3.2 Resource Consumption

Atomic goal formula A means resource consumption and program invocation. All possibilities are examined by backtracking. For example, the following program displays `1` and `2`.

```
r(2).
?- !r(1) -<> r(X), write(X), nl, fail.
```

In goal formula G_1&G_2, resources are copied before execution, and the same resources should be consumed in G_1 and G_2. The following query succeeds by letting `X` = `Y` = 1 and `Z` = 2, or `X` = `Y` = 2 and `Z` = 1, because `r(X)` and `r(Y)` should consume the same resources.

```
?- (r(1), r(2)) -<> ((r(X) & r(Y)), r(Z)).
```

Goal formula !G is just like G, but only infinite resources can be consumed during the execution of G. The following query succeeds by letting `X` = 1 and `Y` = 2.

```
?- (!r(1), r(2)) -<> (!r(X), r(Y)).
```

Goal formula `top` means the erasure of remaining resources. In the following queries, the first one succeeds, but the second one fails because there is a remaining resource.

```
?- (r(1), r(2)) -<> (r(X), top).
?- (r(1), r(2)) -<> r(X).
```

4 EXAMPLES

In this section, some example programs of LLP are described. Through these small programs, we would like to show programming techniques using resources, and usage of resources for constraint satisfaction problems.

Some other useful applications, such as a propositional theorem prover, a database query, and a natural language parser, are described in Hodas and Miller's paper [4]. In addition, current LLP distribution includes example programs, such as BIBD (Balanced Incomplete Design Block), pentomino puzzle solver, four color problem, etc.

4.1 List Reverse

Resources can be used as "slots" to pass parameter values and results.

The following program (list reverse) uses the resource formula `result(Zs)` to return the result from the deepest recursive call of `rev`. For example, by the goal `reverse([1,2,3],Zs)`, the slot `result(Zs)` is added as a resource and the subgoal `rev([1,2,3],[])` is called. At the third recursive call `rev([],[3,2,1])`, the resource `result(Zs)` is consumed and the `Zs` is unified with `[3,2,1]`.

```
reverse(Xs, Zs) :- result(Zs) -<> rev(Xs, []).
rev([], Zs) :- result(Zs).
rev([X|Xs], Zs) :- rev(Xs, [X|Zs]).
```

The same technique can be used to describe "accumulators". For example, a program calculating the summation of a given list can be written as follows.

```
sum(List, Sum) :- result(Sum) -<> s(List, 0).
s([], S) :- result(S).
s([X|Xs], S0) :- S is X+S0, s(Xs, S).
```

4.2 Knight Tour

Resources can be used to represent constrains.

Let us consider a problem of finding a Hamilton path of a given graph. In Hamilton path, all vertices are visited exactly once. This constraint can be represented easily by using resources for vertices. Visited vertices are removed during the execution, and the program succeeds only when all vertices are visited exactly once.

The following program finds a tour of Knight (a chess piece) on a 5×5 board.

```
knight5(Tour) :-
  (k(1,1), k(1,2), k(1,3), k(1,4), k(1,5),
   k(2,1), k(2,2), k(2,3), k(2,4), k(2,5),
   k(3,1), k(3,2), k(3,3), k(3,4), k(3,5),
   k(4,1), k(4,2), k(4,3), k(4,4), k(4,5),
   k(5,1), k(5,2), k(5,3), k(5,4), k(5,5)) -<>
   tour(1, 1, Tour).

tour(I, J, [(I,J)|Tour]) :-
  k(I, J), next(I, J, I1, J1), tour(I1, J1, Tour).
tour(I, J, [(I,J)]) :-
  k(I, J).

next(I, J, I1, J1) :- I1 is I-2, J1 is J-1.
next(I, J, I1, J1) :- I1 is I-2, J1 is J+1.
next(I, J, I1, J1) :- I1 is I-1, J1 is J-2.
next(I, J, I1, J1) :- I1 is I-1, J1 is J+2.
next(I, J, I1, J1) :- I1 is I+1, J1 is J-2.
next(I, J, I1, J1) :- I1 is I+1, J1 is J+2.
next(I, J, I1, J1) :- I1 is I+2, J1 is J-1.
next(I, J, I1, J1) :- I1 is I+2, J1 is J+1.
```

4.3 Kirkman's School Girl Problem

In 1850, Kirkman posed the following problem, which relates to a BIBD (Balanced Incomplete Block Design) problem in mathematics.

> How 15 school girls can walk in 5 rows of 3 each for 7 days so that no girl walks with any other girl in the same triplet more than once.

The following program finds the arrangement.

```
kirkman(Groups) :-
  (arrange(35, Groups) -<> cont) -<> gen_res(15).

gen_res(0) :- cont.
gen_res(N) :- N > 0,
  (g(N),g(N),g(N),g(N),g(N),g(N),g(N)) -<>
  gen_res(1, N).

gen_res(N, N) :-
  N1 is N-1, gen_res(N1).
gen_res(I, N) :-
  I < N, I1 is I+1,
  meet(I, N) -<> gen_res(I1, N).

arrange(0, []).
arrange(I, [[G1,G2,G3]|Groups]) :-
  I > 0,
  % pick up 3 girls
  g(G1), g(G2), g(G3),
  % haven't met each other
  meet(G1, G2), meet(G1, G3), meet(G2, G3),
  I1 is I-1, arrange(I1, Groups).
```

This program works as follows.

(1) `gen_res` creates resources of seven $\mathtt{g}(i)$'s for each $i = 1..15$ and $\mathtt{meet}(i, j)$ for each $i = 1..14$, $j = (i+1)..15$. Seven $\mathtt{g}(i)$'s correspond to seven attendances of i-th girl. Resource $\mathtt{meet}(i, j)$ corresponds to each pair of girls.

(2) Then, `gen_res` calls a goal `cont`, which calls the goal `arrange(35, Groups)` because `cont` is a rule-type resource.

(3) `arrange` finds 35 groups, so that each group consists of three girls, each girl appears in seven groups, and any pair of girls is included in exactly one group.

4.4 Cryptarithmetic Puzzles

By using `top`, we can represent a condition "at most once".

Let us consider a program to solve a famous cryptarithmetic puzzle: "SEND+MORE=MONEY". In this puzzle, digits can be used at most once. In the following program, remaining digits are erased by `top` predicate.

```
crypt([S,E,N,D]+[M,O,R,E]=[M,O,N,E,Y]) :-
  (d(0), d(1), d(2), d(3), d(4),
   d(5), d(6), d(7), d(8), d(9))
  -<>
  (add( 0, D, E, Y, C1),
   add(C1, N, R, E, C2),
   add(C2, E, O, N, C3),
   add(C3, S, M, O, C4),
   add(C4, 0, 0, M,  0),
   S \== 0, M \== 0,
   top).

add(C0, X, Y, Z, C1) :-
  digit(X), digit(Y), digit(Z),
  Sum is C0+X+Y, Z is Sum mod 10, C1 is Sum//10.
```

```
digit(X) :- var(X), d(X).
digit(X) :- nonvar(X).
```

4.5 N-Queens

The last example is the N-queens problem. The following program maps each column, each right-up diagonal, and each right-down diagonal to resources `c(_)`, `u(_)`, and `d(_)` respectively. Attack check is done automatically by consuming `c(`j`)`, `u(`$i+j$`)`, and `d(`$i-j$`)` when placing a queen at (i, j).

```
queen(N, Q) :- (n(N), result(Q)) -<> place(N).

place(1) :-
  (c(1),u(2),d(0)) -<> (n(N), solve(N, [])).
place(I) :-
  I > 1, I1 is I-1,
  U1 is 2*I, U2 is 2*I-1, D1 is I-1, D2 is 1-I,
  (c(I),u(U1),u(U2),d(D1),d(D2)) -<> place(I1).

solve(0, Q) :-
  result(Q), top.
solve(I, Q) :-
  I > 0, c(J), U is I+J, u(U), D is I-J, d(D),
  I1 is I-1, solve(I1, [J|Q]).
```

For example, at the execution of the goal `queen(8,Q)`, the `place` predicate adds the resources `c(1)`, ..., `c(8)`, `u(2)`, ..., `u(16)`, `d(-7)`, ..., `d(7)`, then `solve(8,[])` is called. The `solve` predicate finds a solution by consuming `c(`j`)`, `u(`$i+j$`)`, and `d(`$i-j$`)` for each row $i = 1..8$. After placing 8 queens, the result is returned through the slot `result(Q)`, and remaining resources are erased by the `top` predicate.

5 LLP COMPILER SYSTEM

LLP is a first compiler system for linear logic programming languages. LLP compiles LLP programs into an abstract machine code, which is interpreted by the abstract machine emulator. The abstract machine, called LLPAM, is an extension of the WAM (Warren Abstract Machine [9]). Two new data areas, three new registers, and thirteen new instructions are added to the WAM. Please refer to our previous paper [8] for the details.

The compiler is written in Prolog (we are using SICStus Prolog version 2.1 and BinProlog 5.00). The LLPAM emulator is written in ANSI C (we are using GNU C compiler version 2.7.0).

The newest package (version 0.41) including all source code can be obtained from the following Web page.

`http://bach.seg.kobe-u.ac.jp/llp/`

The system is still under development, and has some limitations. All of them are planned to be solved in a future release.

- No debugging aids. A small interpreter system with tracing facility is available.
- Few built-in predicates (no file I/O, no assert/retract).
- No garbage collectors.
- No optimizations (register allocation, environment trimming, last-call-optimization). The speed of the compiled code for Prolog programs is about 2 or 3 times slower than SICStus Prolog 2.1 WAM code.
- No floating point numbers, integer is 28 bits signed.

6 CONCLUSION

We described new programming features and various example programs of a compiler system of a linear logic programming language called LLP. Its resource handling features enables to describe many kind of constraint satisfaction problems.

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[3] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[4] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstraction in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[5] N. Kobayashi and A. Yonezawa. ACL — a concurrent linear logic programming paradigm. In D. Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 279–294, Vancouver, Canada, October 1993. MIT Press.

[6] D. Miller. A multiple-conclusion meta-logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[7] D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[8] Naoyuki Tamura and Yukio Kaneda. Extension of WAM for a linear logic programming language. In *Proceedings of The Second Fuji International Workshop on Functional and Logic Programming*, Nov. 1996.

[9] David H. D. Warren. An abstract Prolog instruction set. Technical Report Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.