Extension of WAM for a linear logic programming language

Naoyuki Tamura
Graduate School of Science and Technology, Kobe University
1-1 Rokkodai, Nada, Kobe 657 Japan
E-mail: tamura@kobe-u.ac.jp

and

Yukio Kaneda
Department of Computer and Systems Engineering, Kobe University
1-1 Rokkodai, Nada, Kobe 657 Japan
E-mail: kaneda@seg.kobe-u.ac.jp

ABSTRACT

This paper describes an extension of the WAM (Warren Abstract Machine) for a logic programming language called LLP which is based on intuitionisic linear logic. LLP is a subset of Lolli and includes additive and multiplicative conjunction, linear implication in a goal, exponential (!), and the constant 1. The extension of the WAM is mainly for efficient resource management: especially for resource look-up and deletion. In our design, only one table is maintained to keep resources during the execution. Looking-up of a resource is done through a hash table. Deletion of a resource is done by just "marking" the entry in the table. Our prototype compiler produces 25 times faster code compared with a compiled Prolog program which represents resources by a list structure.

1. Introduction

Linear logic developed by J.-Y. Girard [4] is expected to be applied for various fields in computer science. Linear logic is called "resource-conscious" because consumed hypotheses can not be used again. Therefore, in linear logic, a resource can be represented as a formula rather than represented as a term.

There have been several proposals for logic programming language based on linear logic: LO [2], ACL [7], Lolli [6][3], Lygon [9][5], and Forum [8]. Lolli [a] and Lygon [b] 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. BinProlog 5.00 can compile linear implications of affine logic (a variant of linear logic), but other connectives are not covered [12].

In this paper, we describe an extension of the WAM (Warren Abstract Machine) [13][1] for a logic programming language called LLP which is a subset of Lolli and based on intuitionisic linear logic.

The extension of the WAM is mainly for efficient resource management: especially for resource look-up and deletion. In our design, only one table is maintained to keep resources during the execution. Looking-up of a resource is done through a hash

[a] http://www.cs.hmc.edu/~hodas/research/lolli/
[b] http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html

table. Deletion of a resource is done by just changing the level value stored in the table.

2. Language

We use the following fragment for LLP where the symbol A represents an atomic formula, and $\vec{x}$ represents all free variables in the scope.

$$
\begin{array}{lcl}
P & ::= & C \mid C \otimes P \\
C & ::= & !\forall \vec{x}.A \mid !\forall \vec{x}.(G \multimap A) \\
G & ::= & \mathbf{1} \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid !G \mid R \multimap G \\
R & ::= & S \mid !A \mid R_1 \otimes R_2 \\
S & ::= & A \mid S_1 \,\&\, S_2
\end{array}
$$

P, C, G, R, and S stand for "program", "clause", "goal", "resource", and "selective resource" respectively. A program is a (multiplicative) conjunction of clauses. A clause is either a fact ($!\forall \vec{x}.A$) or a rule ($!\forall \vec{x}.(G \multimap A)$). Exponential symbol (!) in a clause means each clause can be used arbitrary times. A goal is either a logical constant ($\mathbf{1}$ or $\top$), an atomic formula (A), a conjunction of subgoals ($G_1 \otimes G_2$ or $G_1 \,\&\, G_2$), a compound formula with exponential symbol ($!G$), or a compound formula with linear implication ($R \multimap G$) where R is a resource formula. A resource formula, in general, has a form $R_1 \otimes R_2 \otimes \cdots \otimes R_n$ (modulo associativity of $\otimes$) where each R_i is either $A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_m$ (modulo associativity of &) or $!A$. We call resources A and $!A$ *primitive resources*.

We use the following notations to write LLP programs corresponding to the above definition of formulas.

$$
\begin{array}{lcl}
P & ::= & C \mid C\ P \\
C & ::= & A\texttt{.} \mid A\texttt{:-}G\texttt{.} \\
G & ::= & \texttt{1} \mid \texttt{top} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid \texttt{!}G \mid R\texttt{-<>}G \\
R & ::= & S \mid \texttt{!}A \mid R_1\texttt{,}R_2 \\
S & ::= & A \mid S_1\texttt{\&}S_2
\end{array}
$$

The order of the operator precedence is "`:-`", "`-<>`", "`&`", "`,`", "`!`" from wider to narrower.

LLP is based on intuitionisic linear logic, and is a subset of Lolli [6], that is, it has fewer features. The following formulas can be used in Lolli but not in LLP.

- Resources including linear implications ($G \multimap A$).
- Goals and resources including universal quantifiers ($\forall x.G$ and $\forall x.R$).
- Compound formulas (& and $\multimap$) in clause heads.

Our future goal is to cover them.

3. Resource Programming

The fragment of LLP is small, but a superset of Prolog, and it enables various "resource programming".

- Atomic goal formula A means resource consumption and program invocation. All possibilities are examined by backtracking.
- Goal formula $R \multimap G$ adds resource R and then executes the goal G. R should be consumed up in G.
- Goal formula $G_1 \otimes G_2$ is similar to conjunctive goal G_1, G_2 of Prolog. Resources consumed in G_1 can not be used in G_2.
- Goal formula $G_1 \mathbin{\&} G_2$ is also similar to conjunctive goal G_1, G_2 of Prolog. But, resources are copied before execution, and the same resources should be consumed in G_1 and G_2.
- Goal formula $!G$ is just like G, but only !'ed resources (that is, resources of the form $!A$) can be consumed during the execution of G.
- Goal formula $\top$ means some resources can be remained without consumption.
- Goal formula $\mathbf{1}$ is similar to 'true' of Prolog.
- Resource formula $R_1 \otimes R_2$ means both R_1 and R_2 are resources.
- Resource formula $A_1 \mathbin{\&} \cdots \mathbin{\&} A_n$ means exactly one of A_i's can be selected as a resource.
- Resource formula $!A$ means A is an infinite resource (that is, it can be consumed arbitrary times including 0 times).

The following program (list reverse) uses the resource formula `result(Zs)` as a "slot" 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 next is a program to solve N-queen problem. In addition to the utilization of slots `n(N)` and `result(Q)`, this program uses the benefit of *resource programming*, that is, consumed resource can not be used again. This program maps each column, each right-up diagonal, and each right-down diagonal to `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) (see Figure 1).

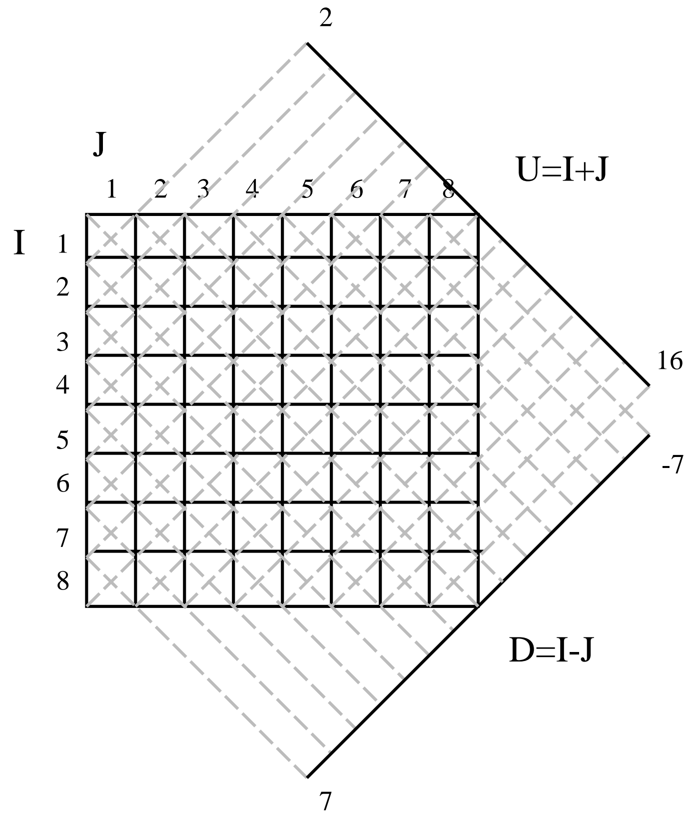


Figure 1: Resources in 8-queen program

```
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.

Of course, it is possible to represent resources by a list and use the following

program to consume a resource.

```
select([X|Xs],X,Xs).
select([Y|Xs],X,Zs) :- select(Xs,X,Zs).
```

However, list is a sequential data structure, and we need to scan the list to find a consumable resource. Also reconstruction is needed when we delete some resource from the list. This is what Lolli interpreter is doing.

4. Resource Management Model

In the execution of the goal $G_1 \otimes G_2$, if we split the resource into two disjoint parts as defined in the original linear logic system, there occurs a lot of non-determinism.

Hodas and Miller in [6] solved this problem by using IO-model in which each goal has its input resource and output resource. Output resource of G_1 is forwarded to G_2 as its input resource.

Before specifying the IO-model of LLP, we define some relations on R-formulas (in this section, we treat $\mathbf{1}$ as a R-formula in addition to the definition given in the previous section).

- **subcontext**(O, I) holds iff
  - $I = \mathbf{1}$ and $O = I$, or
  - $I = !A$ and $O = I$, or
  - $I = A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_n$ (modulo associativity of &) and $O = I$, or
  - $I = A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_n$ and $O = \mathbf{1}$, or
  - $I = I_1 \otimes I_2$, $O = O_1 \otimes O_2$, **subcontext**(O_1, I_1), and **subcontext**(O_2, I_2).
- **pickR**(I, O, A) holds iff
  - $I = !A$ and $O = I$, or
  - $I = A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_n$, $A = A_i$ for some i, and $O = \mathbf{1}$, or
  - $I = I_1 \otimes I_2$, **pickR**(I_1, O_1, A), and $O = O_1 \otimes I_2$, or
  - $I = I_1 \otimes I_2$, **pickR**(I_2, O_2, A), and $O = I_1 \otimes O_2$.
- **thinable**(O) holds iff
  - $O = \mathbf{1}$, or
  - $O = !A$, or
  - $O = O_1 \otimes O_2$, **thinable**(O_1), and **thinable**(O_2).

Figure 2 provides a specification of the IO-model for the propositional fragment of LLP. Note that only the propositional fragment is described, it is easy to extend it to the first order.

$$\frac{}{P \longrightarrow I\{\mathbf{1}\}I} \qquad \frac{\mathbf{subcontext}(O, I)}{P \longrightarrow I\{\top\}O}$$

$$\frac{!A \in P}{P \longrightarrow I\{A\}I} \qquad \frac{!(G \multimap A) \in P \quad P \longrightarrow I\{G\}O}{P \longrightarrow I\{A\}O}$$

$$\frac{\mathbf{pickR}(I, O, A)}{P \longrightarrow I\{A\}O}$$

$$\frac{P \longrightarrow I\{G_1\}M \quad P \longrightarrow M\{G_2\}O}{P \longrightarrow I\{G_1 \otimes G_2\}O} \qquad \frac{P \longrightarrow I\{G_1\}O \quad P \longrightarrow I\{G_2\}O}{P \longrightarrow I\{G_1 \,\&\, G_2\}O}$$

$$\frac{P \longrightarrow I\{G\}I}{P \longrightarrow I\{!G\}I} \qquad \frac{P \longrightarrow R \otimes I\{G\}O' \otimes O \quad \mathbf{thinable}(O')}{P \longrightarrow I\{R \multimap G\}O}$$

Figure 2: IO-model for the propositional LLP

In the sequent $P \longrightarrow I\{G\}O$, P is a set of clauses (C-formulas), I and O are resources (R-formulas), and G is a goal (G-formula). We also assume $\mathbf{subcontext}(O, I)$ holds when we use the sequent.

The sequent $P \longrightarrow I\{G\}O$ means the execution of G under program P where I is given as input resource and O is produced as output resource. A computation of a goal G under program P is considered as a bottom-up and left-to-right proof search of $P \longrightarrow \mathbf{1}\{G\}\mathbf{1}$. Here, we do not consider programs as resources because our abstract machine treats them differently, while Miller and Hodas's original IO-model treats them uniformly.

IO-model is very useful as a basic computation model, but there are still some points which can be improved when its efficient implementation is considered.

- Output resource is reconstructed from input resource by replacing the consumed place with $\mathbf{1}$. For example, when $p \otimes q \otimes r$ is the input for the goal q, the output $p \otimes \mathbf{1} \otimes r$ is reconstructed by the **pickR** predicate. This slows down the execution speed.

- Multiple resources are kept in memory. For example, in the execution of $G_1 \,\&\, G_2$, the input resource I should be kept until the execution of G_2. This wastes memory.

To improve these points, we propose the following ideas.

- Use of consumption-level which is assigned to each primitive resource in the resource. When a primitive resource is consumed, its consumption-level is changed

rather than it is replaced with **1**.

- Use of single resource table. Only one table is maintained during the execution.

Before revising the IO-model based on this idea, we introduce some definitions. We assign a positive integer (called a *consumption-level*) for each primitive resource in R-formula. We use $P^{(\ell)}$ to denote a primitive resource with consumption-level ℓ. Informally speaking, under the specified level L, a primitive resource is consumable if it is an atomic resource whose level is equal to L, or it is a !'ed resource.

In fact, the consumption-level values for !'ed primitive resources are not necessary, and also those for atomic primitive resources in the same selective resource are always the same (that is, $\ell_1 = \cdots = \ell_n$ for any $A_1^{(\ell_1)} \,\&\, \cdots \,\&\, A_n^{(\ell_n)}$), but we assign the level value for each primitive resource to describe our abstract machine architecture precisely.

Now, we need to revise the predicates **subcontext**, **pickR**, and **thinable** for R-formulas with level values. The predicates **subcontext** and **pickR** have two additional arguments L and U where L indicates the level value for consumable primitive resources (its initial value is 1), and U is the level value to be set after the consumption (its initial value is **maxint**, the maximum integer value). The predicate **thinable** has one additional argument L.

We also need two new relations **change** and **put** to represent changing levels and putting levels respectively.

- $\mathbf{subcontext}_{L,U}(O, I)$ holds iff
  - $I = !A^{(\ell)}$ and $O = I$, or
  - $I = A_1^{(\ell)} \,\&\, \cdots \,\&\, A_n^{(\ell)}$ and $O = I$, or
  - $I = A_1^{(L)} \,\&\, \cdots \,\&\, A_n^{(L)}$ and $O = A_1^{(U)} \,\&\, \cdots \,\&\, A_n^{(U)}$, or
  - $I = I_1 \otimes I_2$, $O = O_1 \otimes O_2$, $\mathbf{subcontext}_{L,U}(O_1, I_1)$, and $\mathbf{subcontext}_{L,U}(O_2, I_2)$.
- $\mathbf{pickR}_{L,U}(I, O, A)$ holds iff
  - $I = !A^{(\ell)}$ and $O = I$, or
  - $I = A_1^{(L)} \,\&\, \cdots \,\&\, A_n^{(L)}$, $A = A_i$ for some i, and $O = A_1^{(U)} \,\&\, \cdots \,\&\, A_n^{(U)}$, or
  - $I = I_1 \otimes I_2$, $\mathbf{pickR}_{L,U}(I_1, O_1, A)$, and $O = O_1 \otimes I_2$, or
  - $I = I_1 \otimes I_2$, $\mathbf{pickR}_{L,U}(I_2, O_2, A)$, and $O = I_1 \otimes O_2$.
- $\mathbf{thinable}_L(O)$ holds iff
  - $O = !A^{(\ell)}$, or
  - $O = A_1^{(\ell)} \,\&\, \cdots \,\&\, A_n^{(\ell)}$ and $L \neq \ell$, or
  - $O = O_1 \otimes O_2$, $\mathbf{thinable}_L(O_1)$, and $\mathbf{thinable}_L(O_2)$.
- $\mathbf{change}_{L_1,L_2}(I, O)$ holds iff
  - $I = !A^{(\ell)}$ and $O = I$, or

$$\frac{}{P \longrightarrow_{L,U} I\{\mathbf{1}\}I} \qquad \frac{\mathbf{subcontext}_{L,U}(O, I)}{P \longrightarrow_{L,U} I\{\top\}O}$$

$$\frac{!A \in P}{P \longrightarrow_{L,U} I\{A\}I} \qquad \frac{!(G \multimap A) \in P \quad P \longrightarrow_{L,U} I\{G\}O}{P \longrightarrow_{L,U} I\{A\}O}$$

$$\frac{\mathbf{pickR}_{L,U}(I, O, A)}{P \longrightarrow_{L,U} I\{A\}O}$$

$$\frac{P \longrightarrow_{L,U} I\{G_1\}M \quad P \longrightarrow_{L,U} M\{G_2\}O}{P \longrightarrow_{L,U} I\{G_1 \otimes G_2\}O}$$

$$\frac{P \longrightarrow_{L,U-1} I\{G_1\}M \quad \mathbf{change}_{U-1,L+1}(M, N) \quad P \longrightarrow_{L+1,U} N\{G_2\}O \quad \mathbf{thinable}_{L+1}(O)}{P \longrightarrow_{L,U} I\{G_1 \,\&\, G_2\}O}$$

$$\frac{P \longrightarrow_{L+1,U} I\{G\}O}{P \longrightarrow_{L,U} I\{!G\}O} \qquad \frac{\mathbf{put}_L(R, I') \quad P \longrightarrow_{L,U} I' \otimes I\{G\}O' \otimes O \quad \mathbf{thinable}_L(O')}{P \longrightarrow_{L,U} I\{R \multimap G\}O}$$

Figure 3: IO-model with consumption-level for the propositional LLP

  - $I = A_1^{(L_1)} \,\&\, \cdots \,\&\, A_n^{(L_1)}$ and $O = A_1^{(L_2)} \,\&\, \cdots \,\&\, A_n^{(L_2)}$, or
  - $I = I_1 \otimes I_2$, $O = O_1 \otimes O_2$, $\mathbf{change}_{L,U}(I_1, O_1)$, and $\mathbf{change}_{L,U}(I_2, O_2)$.

- $\mathbf{put}_L(R, O)$ holds iff
  - $R = !A$ and $O = !A^{(L)}$, or
  - $R = A_1 \,\&\, \cdots \,\&\, A_n$ and $O = A_1^{(L)} \,\&\, \cdots \,\&\, A_n^{(L)}$, or
  - $R = R_1 \otimes R_2$, $O = O_1 \otimes O_2$, $\mathbf{put}_L(R_1, O_1)$, and $\mathbf{put}_L(R_2, O_2)$.

Figure 3 provides a specification of the IO-model with consumption-level values. In the sequent $P \longrightarrow_{L,U} I\{G\}O$, P is a set of clauses, I and O are R-formulas with consumption-level, G is a goal, and L and U are positive integers. When we use the sequent, we also assume U is sufficiently larger than L (that is, $U - L$ is greater than the number of nesting calls of $G_1 \,\&\, G_2$ and $!G$), all level values in I and O are in the ranges $1..L$ or $U..\mathbf{maxint}$, and $\mathbf{subcontext}_{L,U}(O, I)$ holds.

A computation of a goal G under program P is considered as a bottom-up and left-to-right proof search of $P \longrightarrow_{1,\mathbf{maxint}} \mathbf{1}^{(1)}\{G\}\mathbf{1}^{(1)}$. In the next section, we describe an extended WAM based on this computation model.

5. LLP Abstract Machine

In this section, we describe LLPAM (LLP Abstract Machine) which is an extended

WAM for LLP language.

5.1. Registers

LLPAM has three new registers `R`, `L`, and `U` in addition to `P` (program pointer), `CP` (continuation program pointer), `S` (structure pointer), `H` (top of heap), `HB` (heap backtrack pointer), `E` (last environment), `B` (last choice point), `B0` (cut pointer), and `TR` (top of trail).

- `R`: top of resource table
  `R` is an index indicating the top of resource table. It increases when resource is added, and decreases by backtracking. Its initial value is 0.
- `L`: consumable level
  `L` is an integer indicating the consumable level of resources. Its initial value is 1.
- `U`: level of consumed resource
  `U` is an integer for the level of consumed resource. Its initial value is **maxint**.

5.2. Resource Table

LLPAM has one new data area called `RES` (resource table) in addition to `CODE`, `HEAP`, `STACK`, `TRAIL`, and `PDL`.

`RES` is an array of the following record structure (we use non-negative integers for indices).

```
record
  level: integer;        { consumption-level }
  eflag: Boolean;        { exponential flag }
  head: term;            { resource structure }
  rellist: term          { related-resources }
end;
```

The register `R` points the top of `RES`. It grows when resource is added by $\multimap$, and shrinks by backtracking.

Each entry of `RES` corresponds to a primitive resource A or $!A$. The field `head` contains a pointer to the resource structure A on `HEAP` ("!" operator is removed for non-atomic resource $!A$). The field `eflag` is set to be true if the resource is $!A$. The field `level` indicates the consumption-level of the primitive resource.

The field `rellist` is a list of the relative positions of the related resources. *Related resources* are primitive resources combined with & operators. For example, in $A_1 \& A_2 \& A_3$, `rellist` of each A_i is $[1, 2]$, $[-1, 1]$, and $[-2, -1]$ respectively. The

	level	eflag	head	rellist
RES[0]	1	1	$p(X)$	[]
RES[1]	1	0	$q(Y)$	[1]
RES[2]	1	0	$p(Z)$	[−1]

Figure 4: Resource table: $!p(X) \otimes (q(Y) \& p(Z))$

related resource list is a constant term and can be determined at compilation time because relative positions are used.

Figure 4 shows the contents of `RES` after adding $!p(X) \otimes (q(Y) \& p(Z))$.

5.3. Code for $R \multimap G$

Resource R is added by a goal $R \multimap G$. To speed up the resource look-up, R is decomposed into primitive resources at compilation time.

The outline of the execution of $R \multimap G$ will be as follows:

(1) Remember the current value of the register `R` in a permanent variable $\mathtt{Y}_i$.
(2) Add all primitive resource in R (suppose there are n primitive resources) to the resource table (`R` is increased by n) and add their entries to the hash table (predicate symbol and the first argument are used as the key).
(3) Execute G.
(4) Check there are no remaining resources in R (resources of positions from $\mathtt{Y}_i$ to $\mathtt{Y}_i + n - 1$ are checked).
(5) Change the consumption-level of each primitive resource of R to a negative value so that it should not be used later.

The following instructions are used for $R \multimap G$.

- `begin_imp Yi`
  Remembers `R` value to a permanent variable `Yi`.
- `add_res` f/n`,Ai,Aj`
  Adds an atomic resource `Ai` of f/n to the resource table and the hash table. `Aj` is the value for `rellist`. The value of register `L` is stored in the `level` field.
- `add_exp_res` f/n`,Ai`
  Adds an exponential resource `Ai` of f/n to the resource table and the hash table. The `rellist` is set to [] (empty list). The value of register `L` is stored in the `level` field.
- `end_imp Yi,`n
  Fails if there is a remaining resource in positions from `Yi` to `Yi`$+n-1$. Otherwise,

negates their consumption-levels (the negation will be canceled by backtracking).

The following is a code for $!p(1) \otimes (p(2) \,\&\, p(3)) \multimap G$.

```
begin_imp Y1
put_str p/1, A1          % put p(1) to A1
set_int 1
add_exp_res p/1, A1      % add !p(1)
put_str p/1, A1          % put p(2) to A1
set_int 2
put_list A2              % put [1] to A2
set_int 1                % add p(2)
set_con []
add_res p/1, A1, A2
put_str p/1, A1          % put p(3) to A1
set_int 3
put_list A2              % put [−1] to A2
set_int -1
set_con []
add_res p/1, A1, A2      % add p(3)
Code for G
end_imp Y1, 3
```

5.4. Code for $G_1 \,\&\, G_2$

In IO-model, G_1 and G_2 of $G_1 \,\&\, G_2$ take the same input resource and should produce the same output resource. In other words, G_2 can consume only what is consumed in G_1 and should consume all of them. The similar idea can be found in [3].

To implement this idea, we assign an integer called *consumption-level* for each primitive resource.

We also introduce two new registers `L` and `U`. The initial values of `L` and `U` are 1 and **maxint** respectively. `L` indicates the consumable level. `U` indicates the level to be assigned after consumption.

The outline of the execution of $G_1 \,\&\, G_2$ will be as follows:

(1) Decrement `U` so that we can know which are consumed in G_1.
(2) Execute G_1.
(3) Change the level of consumed resource in G_1 to `L+1`, that is, all resources of level `U` is changed to level `L+1`.
(4) Increment `L` and `U`.
(5) Execute G_2.

goal	$p\otimes(($	q	$\&$	r	$)\&($	$s\otimes($	t	$\&$	u	$)))$
L	1	1		2		2	2		3	
	$\downarrow$	$\downarrow$	$\nearrow$	$\downarrow$	$\nearrow$	$\downarrow$	$\downarrow$	$\nearrow$	$\downarrow$	
U	m	$m-2$		$m-1$		m	$m-1$		m	
consumption check				2					$3,2$	

Figure 5: Level Transition in $p \otimes ((q \,\&\, r)\&(s \otimes (t \,\&\, u)))$

(6) Check there are no more remaining resource whose consumption-level is `L`.
(7) Decrement `L`.

Figure 5 shows the transitions of level values in the goal $p\otimes((q \,\&\, r) \,\&(s\otimes(t \,\&\, u)))$. Where m means the **maxint**, down arrow means the consumption, up-right arrow means the rewriting of the consumption-level in (3), and values in consumption check field show the consumption-levels to be checked in (6).

The following instructions are used for $G_1 \,\&\, G_2$.

- `begin_with`
  Decrements `U`.
- `mid_with`
  Changes the level of `U` to `L+1` and increments `L` and `U`.
- `end_with`
  Fails if there is a remaining resource of the level `L`. Otherwise, decrements `L`.

The following is a code for $p \,\&(q \otimes (r \,\&\, s))$.

```
begin_with
call p/0
mid_with
call q/0
begin_with
call r/0
mid_with
call s/0
end_with
end_with
```

5.5. *Code for* $!G$

In the execution of $!G$, only exponential resources can be consumed. Again, the consumption-level can be used to implement it.

The outline of the execution of $!G$ will be as follows:

(1) Increment `L` so that only exponential resources can be consumed during G.
(2) Execute G.
(3) Decrement `L`.

The following instructions are used for $!G$.

- `begin_bang`
  Increments `L`.
- `end_bang`
  Decrements `L`.

The following is a code for $!(p \otimes q)$.

```
begin_bang
call p/0
call q/0
end_bang
```

5.6. Code for $\top$

The execution of $\top$ arises non-determinism because **subcontext** has a lot of possibilities. Therefore, we define the operational semantics of $\top$ as follows, although it is not complete.

- `top`
  Consumes all consumable resources.

Consuming *some* consumable resources is the correct way. The treatment in [3] or [9, 5] should be considered for the complete handling of $\top$.

5.7. Code for Atomic Goals

Goal A means resource consumption and predicate invocation (all possibilities are examined by backtracking). The outline of the execution of A will be as follows:

(1) Extract a list of possibly unifiable primitive resources from the hash table (predicate symbol and the first argument are used as the key).
(2) For each primitive resource P in the list, try the following. After the failure of all trials, invoke the predicate A.
(3) Backtrack if P is not consumable.
(4) Unify A with P. Backtrack if the unification fails.
(5) Update the consumption level of P and its related resources to `U` if P is an atomic resource.

The consumability of a primitive resource P of level M can be checked as follows:

- Atomic resource P is consumable iff $M = \texttt{L}$.
- Non-atomic resource P (that is, $P = !A$) is consumable iff $1 \leq M \leq \texttt{L}$.

We use a special built-in predicate `res/2` for the above procedure to make the implementation easy.

The `res/2` might be invoked from the `call` p/n instruction (also `execute` p/n) when there is a resource for p/n. The first argument `A1` is set to the structure A, and the second argument `A2` is set to the list of indices of resources for p/n.

The `res/2` scans the list `A2` to find a consumable primitive resource which is unifiable with `A1`. If there are no such resource, it calls `A1`.

The following is the program of `res/2`.

```
res/2:  try_me_else L3
L1:     get_consumable X3,A2
        update_cpf A2
        update_cpf_BP L2
        unify_resource X3,A1
        proceed
L2:     retry L1
L3:     trust_me
        exec_program_indirect A1
```

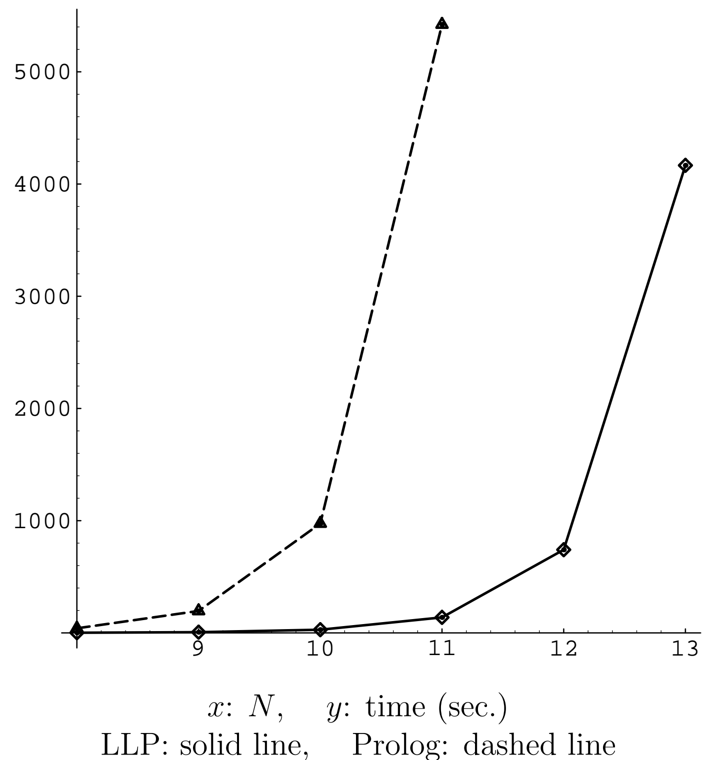
The instructions used in `res/2` are as follows.

- `get_consumable Ai,Aj`
  Finds an index value of consumable resource from the list `Aj`. Fails if there are no consumable resources. `Ai` is set to the index value. The value of `Aj` is also updated.
- `unify_resource Ai,Aj`
  Unifies the resource `RES[Ai]` with `Aj`. If it succeeds, consume the resource `RES[Ai]`.
- `update_cpf Ai`
  Updates the value of `Ai` in the current choice point frame.
- `update_cpf_BP` L
  Updates the value of `BP` in the current choice point frame.

5.8. Backtracking

To recover the old state on backtracking, we need to do the following.

- Register values `R`, `L`, and `U` are stored on each choice point frame.



x: N, y: time (sec.)
LLP: solid line, Prolog: dashed line

Figure 6: Execution time of N-queen (all solutions)

- Resource addition should be trailed.
- Each level change should be trailed.

6. Performance Evaluation

Currently we are developing a compiler system. A prototype compiler system has been developed, in which the compiler is written in SICStus Prolog, and the LLPAM emulator is written in C.

We use N-queen program (all solutions) as a benchmark. A Prolog program representing resources by a list is used for comparison (see Figure 7). This is almost equivalent with a partially evaluated program of Lolli interpreter described in [6]. Only the `subcontext` predicate is modified to improve the speed.

The prototype compiler generates 25 times faster code for 8-queen compared with the Prolog program (compiled to the WAM compact code by SICStus Prolog version 2.1) which spends about 90% of the time for resource management (in `bc/4` and `pickR/3`).

Figure 6 and Table 1 show the CPU times on LLP compiler system and Prolog compiler system.

```
queen(N,Q,R0,R) :-
  place(N,[n(N),result(Q)|R0],[1,1|R]).

place(1,R0,R) :-
  proveA([c(1),u(2),d(0)|R0],R1,n(N)),
  solve(N,[],R1,[1,1,1|R]).
place(I,R0,R) :-
  I > 1, I1 is I-1,
  U1 is 2*I, U2 is 2*I-1,
  D1 is I-1, D2 is 1-I,
  place(I1,[c(I),u(U1),u(U2),d(D1),d(D2)|R0],[1,1,1,1,1|R]).

solve(0,Q,R0,R) :-
  proveA(R0,R1,result(Q)), subcontext(R,R1).
solve(I,Q,R0,R) :-
  I > 0, proveA(R0,R1,c(J)),
  U is I+J, proveA(R1,R2,u(U)),
  D is I-J, proveA(R2,R3,d(D)),
  I1 is I-1, solve(I1,[J|Q],R3,R).

proveA(I,O,A) :- pickR(I,M,R), bc(M,O,A,R).

bc(I,I,A,A).
bc(I,O,A,(R1&R2)) :- bc(I,O,A,R1); bc(I,O,A,R2).

pickR([!R|I],[!R|I],R).
pickR([R|I],[1|I],R) :- \+ R=(!T).
pickR([S|I],[S|O],R) :- pickR(I,O,R).

subcontext([1|O],[R|I]) :- !, subcontext(O,I).
subcontext([S|O],[S|I]) :- !, subcontext(O,I).
subcontext([],[]).
```

Figure 7: Prolog program used for the comparison

N	LLP seconds	LLP (ratio)	Prolog seconds	Prolog (ratio)
8	1.5	(1.0)	40.9	(26.6)
9	6.4	(1.0)	195.6	(30.4)
10	28.4	(1.0)	974.7	(34.3)
11	138.0	(1.0)	5421.8	(39.3)
12	741.2	(1.0)		
13	4166.4	(1.0)		

Table 1: Execution time of N-queen (all solutions)

7. Conclusion and Future Works

The prototype compiler generates 25 times faster code compared with a Prolog program which represents resources as a list. This means the resource management method of LLPAM is 25 times faster compared with the method representing resource as a compound term (e.g. a list).

We are planning the following enhancements.

- Handling $\top$ by using $\top$-flags described in [3].
- Allowing $G \multimap A$ as a resource formula by adding `body` field in `RES` table. The field value will be a pointer to a compiled code (a kind of *closure*).
- Putting `R`, `L`, and `U` in each environment frame so that we can do the last-call-optimization for $R_1 \multimap G_1$, $G_1 \,\&\, G_2$, and $!G$.

Acknowledgments

We would like to thank all students included in this project. Eiji Sugiyama, Mitsunori Banbara, and Kyoungsun Kang helped the design of the language and the abstract machine. Fumiko Anno, Yuichi Ikeda, Tomoaki Kume, and Tadanori Wakamatsu helped the implementation.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine*. The MIT Press, 1991.

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

[3] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the 1996 International Workshop on Extensions of Logic Programming*, pages 67–81. Springer-Verlag LNAI 1050, March 1996.

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

[5] J. Harland and M. Winikoff. Deterministic resource management for the linear logic programming language Lygon. Technical Report TR 94/23, Melbourne University, Department of Computer Science, 1994.

[6] 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.

[7] 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.

[8] D. Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*. (to appear).

[9] 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.

[10] Naoyuki Tamura and Yuichi Ikeda. Resource management method for a compiler system for a linear logic programming language. In *IPSJ 96-PRO-7*, pages 25–30, May 1996. (in Japanese).

[11] Naoyuki Tamura and Yukio Kaneda. Resource management method for a compiler system of a linear logic programming language. In *GMD-Studien Nr. 296, Proceedings of the Poster Session at JICSLP'96*, pages 87–98. German National Research Center for Information Technology, Sep. 1996.

[12] Paul Tarau. *BinProlog 5.00 User Guide*. Department d'Informatique, Université de Moncton, Moncton, Canada, Apr. 1996.

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