Compiling Resources in a Linear Logic Programming Language

Mutsunori Banbara
Department of Mathematics,
Nara National College of Technology,
22 Yata, Yamatokoriyama,
639-1080, Japan
banbara@libe.nara-k.ac.jp

Naoyuki Tamura
Department of Computer and Systems Engineering, Kobe University
1-1 Rokkodai, Nada, Kobe,
657-8501, Japan
tamura@kobe-u.ac.jp

Abstract

There have been several proposals for logic programming language based on linear logic: Lolli [6], Lygon [5], LO [3], LinLog [2], Forum [8]. In addition, BinProlog allows the use of linear implications of affine logic (a variant of linear logic) [12, 13]. In these languages, it is possible to create and consume resources dynamically as logical formulas. The efficient handling of resource formulas is therefore an important issue in the implementation of these languages.

In [10], N. Tamura and Y. Kaneda proposed an abstract machine called LLPAM which is an extension of the standard WAM [14, 1] for a linear logic programming language called LLP. LLP is a superset of Prolog and a subset of Lolli. However, in the original LLPAM design, a resource formula was stored as a term in a heap memory and was not compiled into LLPAM code.

In this paper, we describe an extension of LLPAM for compiling resource formulas. In our extension, resources are compiled into *closures* which consist of compiled code and a variable binding environment.

1 Introduction

Linear logic, a new logic proposed by J.-Y. Girard [4], is drawing attention for applications in various fields of computer science. There have been several proposals for a logic programming language based on linear logic: Lolli [6], Lygon [5], LO [3], LinLog [2], Forum [8]. In addition, BinProlog allows the use of linear implications of affine logic (a variant of linear logic) [12, 13]. In these languages, since it is possible to create and consume resources dynamically as logical formulas, the efficient handling of resource formulas is an important issue.

In [10], N. Tamura and Y. Kaneda proposed an abstract machine called LLPAM which is an extension of the standard WAM [14, 1] for a linear logic programming language called LLP.[1] LLP is a superset of Prolog and a subset of Lolli.[2] The LLPAM resource management method and its extension for $\top$ goals are formally described in the paper [7] by J. Hodas et al.

The creation of resources is, in some sense, similar to the `assert` of clauses (but it is canceled by backtracking). For example, in the goal $(q \multimap p) \Rightarrow G$, creation of the resource $q \multimap p$ is just like

[1] `http://bach.seg.kobe-u.ac.jp/llp/`
[2] `http://www.cs.hmc.edu/~hodas/research/lolli/`

`assert`ing a usual Prolog clause p`:-`q and it can be called during the execution of G. Therefore, in this case, resources can be compiled in the same way as the compilation of clauses.

However, in the previous LLPAM design, a resource formula was stored as a term in a heap memory and was not compiled into LLPAM code. The reason why resources were not compiled is the existence of free variables in the resources. For example, in the goal $(q(X) \multimap p(X)) \Rightarrow G$, X is a free variable which might be instantiated after the creation of the resource. Therefore, it is not possible to compile the resource as an independent clause. Resource formulas with free variables require a variable binding environment.

In this paper, we discuss an extension of LLPAM for compiling resource formulas. To compile resources with free variables, we introduce a new data structure which consists of compiled code and a variable binding environment. This is well known as a *closure* and is widely used in functional programming languages.

2 The LLP Language

The LLP language discussed in this paper is based on the following fragment of linear logic, where A is an atomic formula, $\vec{x}$ represents all free variables in the scope, and $\Rightarrow$ means intuitionistic implication (that is, $A \Rightarrow B$ is equivalent to $!A \multimap B$):

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

The letters C, G and R stand for "clause", "goal" and "resource" respectively. Compared with the fragment used in the previous papers concerning LLPAM [10, 7], newly added operators are implications and universal quantifiers in resources (that is, $G \multimap R$, $G \Rightarrow R$, and $\forall x.R$).

The definition of R can be replaced with the following:

$$\begin{array}{rcl} R & ::= & R' \mid R_1 \,\&\, R_2 \\ R' & ::= & A \mid G \multimap A \mid G \Rightarrow A \mid \forall x.R' \end{array}$$

This is because the following logical equivalences hold in linear logic:

$$\begin{array}{rcll} G_1 \multimap (G_2 \multimap R) & \equiv & (G_1 \otimes G_2) \multimap R & \\ G \multimap (R_1 \,\&\, R_2) & \equiv & (G \multimap R_1) \,\&\, (G \multimap R_2) & \\ G \multimap (\forall x.R) & \equiv & \forall x.(G \multimap R) & \text{(where } x \text{ is not free in } G\text{)} \\ \forall x.(R_1 \,\&\, R_2) & \equiv & (\forall x.R_1) \,\&\, (\forall x.R_2) & \end{array}$$

We also allow $\otimes$-product of resource formulas because $(R_1 \otimes R_2) \multimap G$ is equivalent to $R_1 \multimap (R_2 \multimap G)$.

We will refer to the resource formulas defined by R' as *primitive resources*, and the formula A in a primitive resource as its *head* part. We will also refer to the resource formulas occurring in R of $R \multimap G$ and $R \Rightarrow G$ as *linear resources* and *exponential resources* respectively.

We use the following notation to write LLP programs corresponding to the above definition.

$$\begin{array}{rcl} C & ::= & A. \mid A\texttt{:-}G. \\ G & ::= & \texttt{true} \mid \texttt{top} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid G_1\texttt{;}G_2 \mid \texttt{!}G \mid R\texttt{-<>}G \mid R\texttt{=>}G \\ R & ::= & A \mid R_1\texttt{\&}R_2 \mid G\texttt{-<>}R \mid G\texttt{=>}R \mid \texttt{forall}\ x\texttt{\textbackslash}\ R \end{array}$$

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

LLP covers a significant fragment of Lolli. The principal limitations are: limited-use clauses in the initial program are not allowed; universal quantifiers in goal formulas are not allowed; higher-order quantification and unification of λ-terms are not allowed.

3 Programming in LLP

In this section, we give an intuitive explanation of the resource programming features of LLP. Compared with Prolog, the biggest difference of LLP is its resource consciousness. The LLP system maintains a resource table to which resources can be dynamically added or deleted (consumed) during the execution.

3.1 Resource Addition

Resources are added by the execution of a goal formula R`-<>`G. For example, the following query adds a resource `r(1)` to the resource table and 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 during the execution of G. For example, the following query fails since `r(1)` is not consumed.

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

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

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

The resource formula R_1`,`R_2 is used to add multiple resources. The following query adds resources `r(1)` and `r(2)` and 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)).
```

Note that the following does the same thing.

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

Resource formulas on the left-side of $\Rightarrow$ mean infinite resources; that is, they can be consumed arbitrarily many times (including zero many times). The following query succeeds by letting `X` = `1` or `X` = `2`.

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

The resource formula R_1`&`R_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).
```

Executing a goal with universally quantified resource on the left-hand side of $\Rightarrow$ is similar to `assert`ing a clause. For example, in the following query, the effect during the execution of `r` is as if a clause `p(X) :- q(X)` is `assert`ed.

```
?- (forall X\ (q(X) -<> p(X))) => r.
```

3.2 Resource Consumption

An 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 the 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 $\texttt{X} = \texttt{Y} = 1$ and $\texttt{Z} = 2$, or $\texttt{X} = \texttt{Y} = 2$ and $\texttt{Z} = 1$, because `r(X)` and `r(Y)` should consume the same resources.

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

The goal formula `!`G is just like G, except that only infinite resources may be consumed during the execution of G. The following query succeeds by letting $\texttt{X} = 1$ and $\texttt{Y} = 2$.

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

The goal formula `top` means the consumption of some consumable resources. Of 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).
```

3.3 LLP Example Programs

We show some example programs of LLP here. Other example programs of LLP are described in [11].

The following program finds a path through a directed graph. Since the arcs are added as rule-type linear resources, the conditions can be elegantly expressed.

- Each arc can be used at most once.
- A path is the transitive closure of the arc connected relation.

```
% Path Finding Program
path :-
    (a -<> b) -<>    % arc a -> b
    (b -<> c) -<>    % arc b -> c
    (c -<> a) -<>    % arc c -> a
    (c -<> d) -<>    % arc c -> d
    (d -<> b) -<>    % arc d -> b
    a -<> (d, top).  % find path from a to d
```

The following program temporarily defines a predicate `test`/1 by adding a resource. The predicate is used in the `filter` program to test whether the element satisfies the condition or not.

```
% choose(Xs, Y, Zs)
% Zs is a list of elements greater than Y in Xs.
choose(Xs, Y, Zs) :-
    (forall X \ ( X>Y -<> test(X) )) =>   % define test/1
```

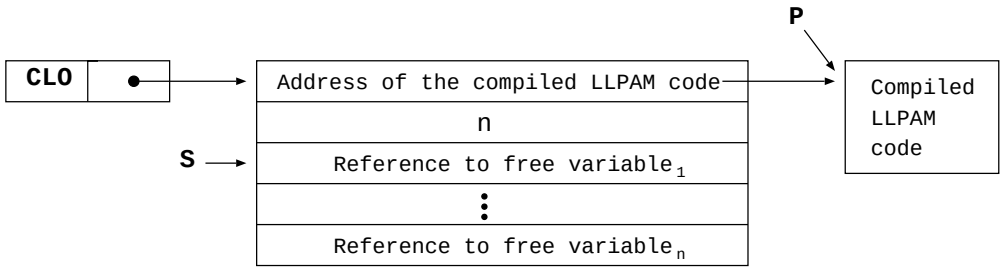

Figure 1: Closure Structure

```
    filter(Xs, Zs).                          % then call filter/2

% filter(Xs, Zs)
% Zs is a list of elements satisfying test/1 in Xs.
filter([], []).
filter([X|Xs], [X|Zs]) :- test(X), !, filter(Xs, Zs).
filter([_|Xs], Zs) :- filter(Xs, Zs).
```

Some useful applications of Lolli, such as a propositional theorem prover, a database query, and a natural language parser, are described in Hodas and Miller's paper [6]. In addition, some programs of BinProlog (including a path finding program) with using linear implication are described in BinProlog user guide [12] and the paper [13].

4 Compiling Resource Formulas

Since any resource formula can be translated into primitive resources, we will only discuss the compilation of primitive resources.

The compilation of resources without free variables is straightforward. They can be compiled just as usual clauses are because they don't require a variable binding environment. For example, the resource formula $\forall X.\forall Y.(q(X,Y) \multimap p(f(X),Y))$ can be compiled just like the clause $p(f(X),Y)$:-$q(X,Y)$.

```
get_structure f/1, A1
unify_variable A1
execute q/2
```

We now discuss compiling resources which contain free variables. For example, X is a free variable in the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$. This resource can't be compiled like a usual clause because we should know the value of X at run-time for the consumption of the resources.

To solve this problem, we introduce a new data structure called *closure*. The closure structure consists of an address of compiled code followed by a variable binding environment, that is, a number of free variables and their references (Figure 1). Pointers to closure structures are tagged by a `CLO` tag value.

When the closure is called, the WAM register `S` is set to point to the third cell of the closure structure, and the instruction pointer `P` is set to the address of the compiled code. Therefore, the

compiled coded will begin with `unify_variable` instructions to retrieve the values of free variables. The following is an example code for the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$:

```
unify_variable A3          % set the value of X to A3
get_structure f/1, A1
unify_value A3
put_value A3, A1
execute q/2
```

5 LLP Abstract Machine

The LLP Abstract Machine (LLPAM) is an extension of the Warren Abstract Machine (WAM) [14, 1]. The extension is mainly for efficient resource management. In this section, we will focus on the compilation of resources and only describe the differences with the previous LLPAM design [10, 7].

5.1 The Resource Table

The resource table `RES` is an array of records with the following structure.

```
record
  level: integer;          {consumption level}
  deadline: integer;       {deadline level}
  out_of_scope: Boolean;   {out_of_scope flag}
  pred: symbol;            {predicate symbol of the head part}
  code: closure;           {code of the resource}
  rellist: term            {related resources}
end;
```

`RES` grows when resources are added by $\multimap$ or $\Rightarrow$, and shrinks on backtracking. Each entry in `RES` corresponds to a single primitive resource.

The `level`, `dead_line`, and `out_of_scope` fields are for the management of resource consumption and their usage is explained in [7], so we omit the explanation in this paper. The field `rellist` is used to find the positions of &-producted resources, and it is explained in [10], so we also omit the explanation here.

The `pred` and `code` fields are used to store the resource information. The field `pred` contains the predicate symbol of the resource, and the field `code` contains a pointer to the closure structure for the resource. In the previous design, a single field `head` is used to contain a term structure of the resource instead of these two fields.

5.2 Code for Resource Addition

A resource R is added by a goal $R \multimap G$ or $R \Rightarrow G$.

The following instructions are used to add primitive resources.

- `put_closure` L, n, `Ai` :
  This creates a closure structure with n free variables on `HEAP`, and sets `Ai` to point to it. The mode is set to `write` mode, and the `unify_value` instructions following this instruction are used to store the references to free variables.

  ```
  Ai := <CLO, H>;
  HEAP[H] := L;
  H := H + 1;
  HEAP[H] := <INT, n>;
  H := H + 1;
  mode := write;
  ```

- `add_prim_res Ai, Aj` :
  This adds a primitive linear resource to `RES`. `Ai` is the head part term of the resource, and `Aj` is a closure. The predicate symbol of `Ai` is stored in the `pred` field, and `Aj` is stored in the `code` field. The index register `R` pointing to the bottom of `RES` is incremented.

  ```
  RES[R].pred :=  predicate symbol of Ai;
  RES[R].code := Aj;
   Set level, dead_line, and out_of_scope fields;
   Record this entry in the hash table;
  R := R + 1;
  ```

- `add_prim_exp_res Ai, Aj` :
  This behaves the same as `add_prim_res`, except the setting of the `level` and `dead_line` fields are set for primitive exponential resources.

In `add_prim_res` and `add_prim_exp_res`, the whole structure of the head part of the resource is used instead of only the predicate symbol of the head part. This is because we want the argument values of the head part to calculate the hash value of the added resource. In the current implementation, we only use the predicate symbol name (with its arity) and the first argument value for hash, but to improve the implementation in the future, we decided to pass all argument values.

The following code is for adding the primitive linear resource $q(X, Y) \multimap p(X)$ where `A1` and `A2` store the variable X and Y respectively.

```
put_structure p/1, A3
unify_value A1
put_closure L, 2, A4
unify_value A1
unify_value A2
add_prim_res A3, A4
```

```
L : unify_variable A3    % retreive X
    unify_variable A2    % retreive Y
    get_value A3, A1     % unify the 1st argument with X
    execute q/2
```

5.3 Code for Calling Resources

An atomic goal means resource consumption or an ordinary program invocation in LLP. In the previous LLPAM design, the execution of a `call` to an atomic goal A proceeded as follows:

(1) Extract a list of pointers to the possibly consumable resources in the resource table, `RES`, by referring to the hash table.

(2) For each `RES` entry R in the extracted list, attempt the following:

  (a) If R is out of scope, or is linear and has been consumed, fail.

  (b) Attempt to unify A with the `head` field of R.

  (c) Mark the entry R as consumed.

(3) After the failure of all trials, `call` the ordinary code for predicate A.

The above procedure was executed by a special built-in predicate, and it was invoked from the `call` instruction as described in [10]. This approach was taken only to make the implementation easy, but it makes the execution of LLPAM difficult to understand.

In this section, we describe a new code generation method for atomic goals. The outline of the execution of an atomic goal A with predicate symbol p/n is as follows:

(1) Extract the list of indices of the possibly consumable primitive resources in the resource table, `RES`, by referring to the hash table. In the current implementation, predicate symbol p/n and the first argument of A are used for the hash key. The two registers `R1` and `R2` are used to store the extracted lists of indices.

(2) For each `RES` entry R with predicate symbol p/n in the extracted lists `R1` and `R2`, attempt the following:

  (a) If R is out of scope, or is linear and has been consumed, fail.

  (b) Mark the entry R as consumed.

  (c) Execute the compiled code closure of R.

(3) After the failure of all trials, `call` the ordinary code for predicate A.

Step (1) is done by the `call` (or `execute`) instruction. The register `R1` is set to include the indices of the resources which have the same predicate symbol and the same first argument in its head part. The register `R2` is set to include the indices of the resources which have the same predicate symbol, but the first argument was an unbound variable when the resource was added. We need the resources in `R2` because their heads are also possibly unifiable with the goal A.

The following new instructions are used for steps (2) and (3).

- `try_resource` L :
  This behaves the same as "`try` L", but `R1` and `R2` are also saved in the created choice point frame.

- `restore_resource` :
  This restores the register values from the current choice point frame (indicated by the current value of `B` register).

- `retry_resource_else` L :
  This replaces the `R1` and `R2` values in the current choice point frame by their current values. The next clause field `BP` in the choice point frame is also replaced by L.

- `trust_resource` L :
  This sets the current value of the `B` register to its predecessor and then jumps to an address L.

- `pickup_resource` p/n`, Ai,` L :
  This finds the index value of the consumable resource with predicate symbol p/n from `R1` (or `R2` if `R1` is `nil`) and sets that index value to `Ai`. `R1` and `R2` are updated to have the remaining resources. If there are no consumable resources, it jumps to L.

```
found := false;
while R1 ≠ nil and not found do begin
    i := car(R1);
    R1 := cdr(R1);
    if RES[i].pred = p/n then
        found := true
end;
while R2 ≠ nil and not found do begin
    i := car(R2);
    R2 := cdr(R2);
    if RES[i].pred = p/n then
        found := true
end;
if not found then
    P := L;
```

- `consume Ai, Aj` :
  This marks the resource pointed by the index value `Ai` as consumed and sets `code` field value to `Aj`.

```
 Mark RES[Ai]  as consumed;
Aj := RES[Ai].code;
```

- `execute_closure Ai` :
  This executes the closure `Ai` (suppose it is $\langle \texttt{CLO}, c \rangle$) and then sets register `S` to $c + 2$ and set the mode to `read`. Then, it jumps to the address pointed to by `HEAP[`c`]`.

```
<CLO, c> := Ai;
S := c + 2;
mode := read;
P := HEAP[c];
```

```
p/2   :  pickup_resource p/2, A3, L'      %  added
         try_resource L1
L0    :  restore_resource
         pickup_resource p/2, A3, L2
         retry_resource_else L0
L1    :  consume A3, A4
         execute_closure A4
L2    :  trust_resource L'
```

Figure 2: Code for $p/2$

For example, the following is the simplest code for an atomic goal with $p/2$ (where L' is the address of ordinary program code):

```
p/2   :  try_resource L1
L0    :  restore_resource
L1    :  pickup_resource p/2, A3, L2
         retry_resource_else L0
         consume A3, A4
         execute_closure A4
L2    :  trust_resource L'
```

The above code contains an obvious inefficiency. Even if there is no consumable resource, it allocates a new choice point frame by the `try_resource` instruction. The improved code (Figure 2) will begin with the `pickup_resource` instruction to check whether there are any consumable resources or not. If there are no consumable resources, it executes the ordinary program code immediately.

We now discuss the optimization of the code for atomic goals. For every execution of an atomic goal, the resource consumption must be examined, regardless of whether there exists an ordinary program invocation or not.

On the other hand, when there is no ordinary program invocation, an atomic goal means only resource consumption. Our design of optimization is limited to this case, and the essence of it is as follows:

- If there is only one consumable resource, all we have to do is consume it immediately.
- It is safe to discard the current choice point frame before consuming the last consumable resource.

The following new instruction is used to generate the optimized code for atomic goals.

- `if_no_resource` L :
  This scans whether there are consumable resources in `R1` and `R2`. If there are no consumable resources, jump to L.

```
p/2  :  pickup_resource p/2, A3, fail
        if_no_resource L1                    %  added
        try_resource L1
L0   :  restore_resource
        pickup_resource p/2, A3, L2
        if_no_resource L3                    %  added
        retry_resource_else L0
L1   :  consume A3, A4
        execute_closure A4
L2   :  trust_resource fail
L3   :  trust_resource L1                    %  added
```

Figure 3: Optimized code for $p/2$ when there are no programs

The above idea can be achieved quite easily by inserting the `if_no_resource` instruction just after the `pickup_resource` instructions. Figure 3 shows an optimized code corresponding to the Figure 2.

6 Performance Evaluation

In this section, we would like to show an improvement in performance by compiling resources instead of representing resources as heap terms. Currently we are developing the LLP compiler system based on the LLPAM presented in this paper, and a prototype system has been developed.

We use a simple puzzle "tiling board with Dominoes" (all solution) as a benchmark. Dominoes are puzzle pieces. Each piece consists of two equal squares. There are exactly two possible shapes. The goal of the puzzle is to place the dominoes so that they fit into the board of given dimension.

A LLP program (Figure 4) is used for the comparison. In the program, let the board size be (m, n), the following conditions can be expressed elegantly:

- All $m \times n$ units of the board can be used exactly once.
  This can be represented easily by mapping each unit to a resource `b(_,_,_)`.
- All $\frac{m \times n}{2}$ dominoes can be used and placed anywhere on the board.
  This condition can be expressed by mapping each domino to a &-product of rule-type resources which have `domino(_)` as their head parts. Placing a domino at (i, j) is done by consuming `b(`i`,`j`,_)` and `b(`i`,`$j+1$`,_)` (or `b(`i`,`j`,_)` and `b(`$i+1$`,`j`,_)`) in body parts.

Attack check is done automatically by consuming `domino(_)`.

The LLPAM code for Domino puzzle program is about 28% faster than code of previous version of the LLPAM. This means that compiling resources is about 28% faster than representing resources as terms in a heap memory. This speedup is due to the compilation of rule-type resources which have compound goal formulas as their body parts. Sadly, the speedup for other benchmarks which don't contain compound resources is smaller than that for Domino puzzle. Table 1 shows CPU times on our prototype system of Domino puzzle program finding all solutions for various board sizes. All times were collected on a SUN SPARCstation-20 running SunOS 4.1.4.

Board size (row, column)	# Runs Averaged	Previous version of the LLPAM	The LLPAM described here	% Δ
(2, 5)	10	533 ms	370 ms	30%
(2, 6)	10	5,863 ms	4,216 ms	28%
(3, 4)	10	6,016 ms	4,283 ms	29%
(2, 7)	10	71,750 ms	53,800 ms	25%
(4, 4)	5	1,247,700 ms	905,354 ms	27%

Table 1: Execution time of Domino puzzle (all solution)

The LLP compiler system [11] based on the previous LLPAM design has been developed since 1996. It consists of the LLP to LLPAM compiler (written in Prolog) and the LLPAM emulator (written in C). Prolog programs can be executed without any modification. The speed of the compiled code for Prolog programs is about 2 or 3 times slower than SICStus Prolog 2.1 WAM code. The latest package (version 0.43) including all source code can be obtained from the following site:

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

7 Conclusion and Future Work

In this paper we have described an extension of LLPAM for compiling resource formulas. To compile resources with free variables, we introduce a new data structure which consists of some compiled code and a set of bindings for free variables. Such a structure corresponds to the idea of a *closure* that is widely used in implementations of functional programming languages.

With regard to performance, the LLPAM code for a benchmark is about 28% faster than code of previous version of the LLPAM. This means that compiling resources is about 28% faster than representing resources as terms in a heap memory.

We note the similarity between our scheme and that used for the higher-order logic programming language λProlog. In [9], G. Nadathur et al. propose implementation method for allowing $D \supset G$ as a goal in λProlog. The implication goal $D \supset G$ corresponds to the goal $D \Rightarrow G$ of LLP. Indeed, the idea of closure is similar to that discussed in Section 5 (in [9]). Our scheme differs in details from that in [9] because of the existence of linear implication.

The following points are still remaining:

- Compiling resource formulas dynamically at execution time.
  Our scheme doesn't cover use of metavariables as resources. Dynamic resource compilation will give an answer to this problem.

- Allowing $\forall x.G$ as a goal formula.
  The paper [9] describe implementation method for allowing $\forall x.G$ as a goal in λProlog.

Currently, we are improving existing LLP compiler system to incorporate resource compilation mechanism presented in this paper.

```
%%%  Solver
solve_domino(M, N) :-
        D is M*N/2,
        row(M) =>
        column(N) =>
        num_of_dominoes(D) =>
        %((place_domino(D) & write_board(1,1)) -<> cont)
        (place_domino(D) -<> cont)
        -<>
        gen_res(M, N).

place_domino(0).
place_domino(N) :-
        N > 0,
        domino(N),
        N1 is N-1,
        place_domino(N1).

%%%  Create Resources
gen_res(0) :- cont.
gen_res(N) :-
        N > 0,
        ((b(I, J, N), J1 is J+1, b(I, J1, N)) -<> domino(N)
        &
        (b(I, J, N), I1 is I+1, b(I1, J, N) ) -<> domino(N))
        -<>
        (N1 is N-1, gen_res(N1)).

gen_res(I, J) :-
        I < 1,
        !,
        num_of_dominoes(D),
        gen_res(D).
gen_res(I, J) :-
        J < 1,
        !,
        I1 is I-1,
        column(N),
        gen_res(I1, N).
gen_res(I, J) :-
        J1 is J-1,
        b(I, J, _) -<> gen_res(I, J1).
```

Figure 4: Domino puzzle program used for the comparison

Acknowledgements

I would like to thank referees for giving us useful comments and suggestions. I'm grateful to Joel David Hamkins for his useful advice.

References

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

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

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

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

[5] J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 304–318, San Diego, California, Oct. 1991.

[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 abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[7] J. S. Hodas, K. Watkins, N. Tamura, and K.-S. Kang. Efficient implementation of a linear logic programming language. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, June 1998. (to appear).

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

[9] G. Nadathur, B. Jayaraman, and K. Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, Nov. 1995.

[10] N. Tamura and Y. Kaneda. Extension of WAM for a linear logic programming language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 33–50. World Scientific, Nov. 1996.

[11] N. Tamura and Y. Kaneda. A compiler system of a linear logic programming language. In *Proceedings of the IASTED International Conference Artificial Intelligence and Soft Computing*, pages 180–183, July 1997.

[12] P. Tarau. BinProlog 5.40 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, Apr. 1997. Available from `http://clement.info.umoncton.ca/BinProlog`.

[13] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. In *ILPS'95 Workshop on Visions for the Future of Logic Programming*, Nov. 1995.

[14] D. H. D. Warren. *An abstract Prolog instruction set*. Technical Note 309, SRI International, October 1983.