Electronic Notes in Theoretical Computer Science 30 No. 3 (2003)
URL: `http://www.elsevier.nl/locate/entcs/volume30.html` 26 pages

Translating a Linear Logic Programming Language into Java

Mutsunori Banbara

Department of Mathematics
Nara National College of Technology
Nara, Japan

Naoyuki Tamura

Department of Computer and Systems Engineering
Kobe University
Kobe, Japan

Abstract

There have been several proposals for logic programming language based on linear logic: Lolli [8], Lygon [7], LO [3], LinLog [2], Forum [11], HACL [10]. 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. Lolli, Lygon, and Forum are implemented as interpreter systems; Lolli is on SML and λProlog, Lygon is on Prolog, Forum is on SML, λProlog and Prolog. However, none of them have been implemented in Java.

In this paper, we describe the Prolog Café [1] system which translates a linear logic programming language called LLP to Java via the LLPAM [12][5], an extension of the standard WAM [16][1] for LLP. LLP is a superset of Prolog and a subset of Lolli. The main difference from the first implementation [4] is resource compilation. That is to say, resource formulas are compiled into *closures* which consist of a reference of compiled code and a set of bindings for free variables. Calling these resources is integrated with the ordinary predicate invocation.

Prolog Café is portable to any platform supporting Java and easily expandable with increasing Java's class libraries. In performance, on average, Prolog Café generate 2.2 times faster code for a set of classical Prolog benchmarks compared with jProlog.

[1] `http://pascal.seg.kobe-u.ac.jp/~banbara/PrologCafe/`

1 Introduction

The implementation design of efficient Prolog systems in Java is now an active topic. A number of Prolog systems have been developed recently: BirdLand's Prolog in Java, CKI Prolog, DGKS Prolog, JavaLog, Jinni, JP, jProlog, LL, MINERVA, and W-Prolog. jProlog [2], developed by B. Demoen and P. Tarau, is the first Prolog-to-Java translator system based on continuation passing style compilation, called binarization transformation [15]. MINERVA, developed by IF Computer, compiles Prolog into its own virtual machine which is then executed in Java. The others are implemented as interpreter systems.

Linear Logic, a new logic proposed by J.-Y. Girard [6], is drawing attention for applications in various fields of computer science. Linear Logic is called "resource-conscious" logic because the structural rules of weakening and contraction are available only for assumptions marked with the modal "!". In other words, assumptions not marked can only be used once. Limited resources can be therefore represented by formulas rather than by terms.

There have been several proposals for logic programming language based on linear logic: Lolli [8], Lygon [7], LO [3], LinLog [2], Forum [11], HACL [10]. 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. Lolli, Lygon, and Forum are implemented as interpreter systems; Lolli is on SML and λProlog, Lygon is on Prolog, Forum is on SML, λProlog and Prolog. In [12], N. Tamura and Y. Kaneda proposed an abstract machine LLPAM which is an extension of the standard WAM [16][1] for a linear logic programming language called LLP. An extension of the LLPAM for compiling resources was proposed in the paper [5], and the complete treatment of $\top$ was proposed in the paper [9]. However, none of them have been implemented in Java.

In this paper, we describe the Prolog Café system which translates LLP to Java via an extended LLPAM [5] for compiling resources. LLP is a superset of Prolog and a subset of Lolli. The first implementation [4] of Prolog Café is based on the original LLPAM [12]. The main difference from the first implementation is resource compilation. In our extension, resource formulas are compiled into *closures* which consist of a reference of compiled code and a set of bindings for free variables. The calling of resources is integrated with the ordinary predicate invocation.

It is true that Java is not a fast language even with the help of JIT (Just-In-Time Compiler), but Prolog Café and other Prolog systems implemented in Java have the following advantages and possibilities:

extensibility: the Prolog Café system can be easily expandable with increasing Java's class libraries: Java3D, JavaSpaces, JDBC, and so on. We will present an implementation of `assert` and `retract` by using Java's hash

[2] `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`

table in section 4.5.

possibilities of network programming: It is possible for Prolog Café to download HTML (or XML) source files through the internet and parse them by DCG since URL connections can be achieved easily in Java. By integrating JavaSpaces into Prolog Café, it is also possible to exchange data and migrate programs through the internet since all data structures are Java objects in Prolog Café. These are not simple tasks in other languages like C.

portability: the Prolog Café system can run on any platform supporting Java.

This paper is organized as follows. In section 2, we present the definition and resource programming features of LLP. Section 3 shows how the first implementation translates LLP into Java. In section 4, we present the differences between Prolog Café and the first implementation. Section 5 shows performance evaluations.

2 The LLP Language

In this section, we present the definition and resource programming features of LLP.

2.1 The Definition of LLP

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 $!A \multimap B$):

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

The letters C, G and R stand for “clause”, “goal” and “resource” respectively. Compared with the fragment used in the first implementation [4], the newly added operator is the universal quantifier in resources (that is, $\forall x.R$). The definition of R can be replaced with the following:

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

This is because the following logical equivalences hold in linear logic (where z is not free in G):

$$\begin{aligned}
G_1 \multimap (G_2 \multimap R) &\equiv (G_1 \otimes G_2) \multimap R\\
G \multimap (R_1 \,\&\, R_2) &\equiv (G \multimap R_1) \,\&\, (G \multimap R_2)\\
G \multimap (\forall x.R) &\equiv \forall z.(G \multimap R[z/x])\\
\forall x.(R_1 \,\&\, R_2) &\equiv (\forall x.R_1) \,\&\, (\forall x.R_2)
\end{aligned}$$

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

$C ::= A.$ | A`:-`$G.$
$G ::=$ `true` | `top` | A | G_1`,`G_2 | G_1`&`G_2 | G_1`;`G_2 | `!`G | R`-<>`G | R`=>`G
$R ::= A$ | R_1`&`R_2 | G`-<>`R | G`=>`R | `forall` x`\` R

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

2.2 Resource Programming

We will give an intuitive explanation of the resource programming features of LLP. Compared with Prolog, the biggest difference of LLP is its resource consciousness. Our system maintains a single resource table to which resources can be dynamically added or consumed during the execution.

Resources are added by the execution of goal formula R`-<>`G. All resources in R should be consumed during the execution of G.

- The resource formula A represents a *fact-type resource* which can be consumed exactly once.
- The resource formula G`-<>`A represents a *rule-type resource*, in which the goal G is executed on resource consumption of A.
- The resource formula R_1`,`R_2 represents multiple resources.
- The resource formula on the left-side of `=>` represents infinite resources. That is, they can be consumed arbitrarily many times (including zero many times).
- The resource formula R_1`&`R_2 is used to represent a *selective resource*. When R_1`&`R_2 is added to the resource table, either R_1 or R_2 can be consumed, but not both of them.
- Adding an universally quantified resource on the left-hand side of `=>` is similar to `assert`ing a clause. However, this addition can be cancelled by backtracking.

An atomic goal formula A represents resource consumption and predicate invocation. All possibilities are examined by backtracking.

- The goal formula G_1`,`G_2 is similar to the goal of Prolog, except that resources consumed in G_1 can not be consumed in G_2.
- The goal formula G_1`&`G_2 is similar to the conjunctive goal of Prolog, except that all resources are copied before execution, and the same resources must

be consumed in both G_1 and G_2.

- The goal formula `!`G is just like G, except that only exponential resources can be consumed during the execution of G.
- The goal formula `top` represents consumption of all consumable resources.

In current implementation, the goal formula `top` does not correspond to $\top$ in linear logic completely. The correct way to treat $\top$ is to consume some consumable resources, but not necessarily all. The complete treatment of $\top$ for the LLPAM was proposed in the paper [9].

2.3 LLP Example

Let us consider a problem for tiling board with dominoes. Each domino consists of two equal squares and has exactly two possible shapes. The goal is to place the dominoes so that they fit into the board of given dimension. Let the board size be (m, n). The following conditions can be easily expressed by using fact-type and rule-type resources (Fig. 1):

- All $m \times n$ units of the board must be used exactly once. This condition can be represented by mapping each unit to a fact-type resource `b(_,_,_)`.
- All $\frac{m \times n}{2}$ dominoes must be used and placed 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 automatically by consuming `b(`i`,`j`,_)` and `b(`i`,`$j+1$`,_)` (or consuming `b(`i`,`j`,_)` and `b(`$i+1$`,`j`,_)`) in body parts.

Other LLP examples are described in [13]. 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 [8]. In addition, BinProlog programs that using linear implication are described in BinProlog user guide [14].

3 The first implementation of Prolog Café

In this section, we will detail the first implementation of Prolog Café.

This prototype system is based on jProlog which was the first Prolog to Java translator developed by B. Demoen and P. Tarau. jProlog is based on continuation passing style compilation called binarization [15]. In the jProlog approach, each predicate is translated a set of classes; there is one class for entry point, and other classes exist for clauses. Continuations are represented as terms and executed through hash lookup to transform the terms to their corresponding predicate objects.

In our prototype system, each term is translated into a Java object of classes: `VariableTerm`, `IntegerTerm`, `SymbolTerm`, `ListTerm`, and `StructureTerm`. The `Term` class, with an abstract method `unify`, is a common superclass of these classes. Each clause is first translated into a binary clause and then translated into one method. Before binarization, the goals R`-<>`G, R`=>`G, G_1`&`G_2, and `!`G are replaced by built-in predicates which correspond to the original LLPAM instructions [12] not incorporating resource compilation. Each predicate is translated into only one class, and each has methods corresponding to its clauses. Translated predicates are executed by invoking their `exec` method, and returning from it means failure. The `Predicate` class with an abstract method `exec` is a common superclass of that class. The `Predicate` class has `cont` and `trail` fields for continuation and trail stack respectively.

Fig. 2 shows a part of generated code for the following example which contains the goals: R`-<>`G, G_1`&`G_2, and `!`G. The third argument of `add_res/3` is used to represent the positions of &-producted resources. We omit the code for R`=>`G since it is almost same as R`-<>`G.

```
%%% source clauses
p(X, Y) :- q(X) -<> r(Y).
p(X, Y) :- q(X) & r(Y).
p(X, Y) :- !((q(X), r(Y))).

%%% intermediate clauses
p(X, Y) :-
        begin_imp(A),
        add_res(q(X), true, []),
        mid_imp(B), r(Y),
        end_imp(A, B).
p(X, Y) :-
        begin_with,
        q(X),
        mid_with,
        r(Y),
        end_with.
p(X, Y) :-
        begin_bang,
        q(X),
        r(Y),
        end_bang.
```

In this approach, we do not need to maintain choice point stack, and trail stack is maintained in each predicate locally. The cut is easily implemented by Java's exception handling: `try` and `catch`. In performance, our prototype translator generated slight faster code for some Prolog benchmarks compared with jProlog. However, the execution speed of the first implementation for LLP benchmarks is about 50 times slower than that of the LLPAM code. Other drawbacks of this approach are as follows:

- Resource formulas are not compiled and stored as terms in the resource table. For example, the resource consumption of `domino(_)` in LLP example

(section 2.3) is done by construct-general unification, and its body part is executed by the interpreter. This slows down the execution speed of resources.
- A continuation is compiled as a predicate and executed by invoking its `exec` method. However, this invocation will call another nested `exec` before returning and will not return until the system reaches the first solution of program. This leads to a Java memory overflow for large programs.

In addition, the prototype system did not incorporate indexing, specialization of unification, or bootstrapping.

4 Prolog Café

Prolog Café is a LLP system which translates LLP to Java via an extended LLPAM [5] for compiling resources. Let us summarize the differences between Prolog Café and its first implementation:

- To execute resources efficiently, resource formulas are compiled into *closures* which consist of a reference of compiled code and a set of bindings for free variables. The calling of resources is integrated with the ordinary predicate invocation.
- Each predicate is translated into a set of classes including classes for clauses like jProlog. To avoid memory overflows, compiled continuations are executed by a supervisor function like KL1.
- The `JavaObjectTerm` class enables us to use Java objects as terms which can be created by a built-in predicate `java_constructor/2`. We can make use of methods and access fields of those objects in LLP programs by built-in predicates: `java_method/3`, `java_set_field/3`, and `java_get_field/3`.
- Prolog Café incorporates *switch_on_term*(a form of indexing), specialization of head unification, and bootstrapping.

In this section, we focus on Java implementation for compiling resources and describe the differences from the first implementation.

4.1 Compiling Resource Formulas

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

The compilation of resources that have no free variables is straightforward. They can be translated just as usual clauses are because they don't require a variable binding environment. For example, the resource formula $\forall X.\forall Y.(q(X,Y) \multimap p(X,Y))$ can be translated just like $p(X,Y)$`:-`$q(X,Y)$.

We now discuss compiling resources which contain free variables. For example, X is a free variable in the resource $\forall Y.(q(X,Y) \multimap p(X,Y))$. 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 introduced in the paper [5] a data structure called *closure*. The closure structure consists of a reference of compiled code and a set of bindings for free variables. When the closure is called, certain argument registers are set to point to the free variables, and the compiled code is executed. Therefore, the translated code for resources must contain the instructions to retrieve the values of free variables.

For compiling resources, we implement the idea of closure in Java. Fig. 3 shows generated code for the first clause of our previous example, which contains only R-<>G. The second argument `[q/1,0,[X]]` of `add_prim_res/2` corresponds to the closure structure for resource `q(X)`, where `0` means a number assigned to this resource. We omit the code for G_1&G_2 and `!`G since they are almost same as those of the first implementation.

```
%%% source clauses
p(X, Y) :- q(X) -<> r(Y).

%%% intermediate clauses
p(X, Y) :-
        begin_imp(A),
        add_prim_res(q(X), [q/1,0,[X]]),
        mid_imp(B),
        r(Y),
        end_imp(A, B).
```

In fig. 3, the `engine` field means the current LLP engine which is activated. The `engine.aregs` and `engine.cont` fields mean the argument registers and the continuation register respectively. The closure structures can be implemented easily by using the `ClosureTerm` class. In later section, we will show an improvement in performance by compiling resources compared with the first implementation.

4.2 The Resource Table

The resource table (`RES`) is implemented as an array of the `PrimRes` class.

```
final class PrimRes{
    public int level;             // consumption level
    public boolean isOutOfScope;  // out_of_scope flag
    public SymbolTerm pred;       // predicate symbol of head part
    public ClosureTerm closure;   // closure for resource
    public Term rellist           // related resources
    public PrimRes(){}
}
```

`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` and `isOutOfScope` fields are for the management of resource consumption and their usage is explained in [9] so we omit the explanation in this paper. The field `rellist` is used to find the positions of &-producted resources, and as it is explained in [12][4], we also omit its explanation here. The `pred` and `closure` fields are used to store the resource information. The field `pred` contains the predicate symbol of head part of resource, and the field `closure` contains a reference to the closure structure for the resource.

4.3 Code Generation for Resource Addition

A resource R is added by a goal $R \multimap G$ or $R \Rightarrow G$. The following new built-in classes are used to add primitive resources. Each class corresponds to an extended LLPAM instruction in [5].

- `PRED_add_prim_res_2(Term arg1, Term arg2, Predicate cont)`
  This adds a primitive linear resource to `RES`. The `arg1` is the head part term of the resource, and the `arg2` is its closure. The predicate symbol of head term and closure are stored respectively in the `pred` and `closure` field of the entry in `RES`.
- `PRED_add_prim_exp_res_2(Term arg1, Term arg2, Predicate cont)`
  This behaves the same as `PRED_add_prim_res_2`, except that the `level` field is set to the value for primitive exponential resources.

A hash table is used to speed access to the resources in `RES`. In the current (also first) implementation, we only use the predicate symbol and the first argument value of their head parts for hash key. However, we can not always rely on the hash table for access to the resources. When the goal has an unbound variable as the first argument, we must access all entries for the given predicate symbol, regardless of the first argument. Similarly, those resources in which the first argument is an unbound variable must be examined for every call on that predicate symbol. Therefore, the entry for a predicate symbol in the symbol table contains two lists. One is a list of indices of all resources with that predicate name/arity. Another is a list of indices of all resources with that predicate name/arity and an unbound variable as its first argument.

Fig. 4 shows the code for the goal $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y)) \multimap G$, where `varX` stores the variable X. This goal adds a rule-type resource in which the resource $q(X)$ will be added at resource consumption of $p(\_,\ \_)$.

4.4 Code Generation for Atomic Goals

An atomic goal means resource consumption and an ordinary predicate invocation in LLP. The outline of the execution of an atomic goal A with predicate symbol p/n is as follows:

(i) Extract the list of indices of the possibly consumable primitive resources in the resource table, `RES`, by referring to the hash and symbol tables. In the current implementation, the 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.

(ii) For each `RES` entry R with predicate symbol p/n in the extracted lists `R1` and `R2`, execute 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 of closure for R.

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

We use eight methods to call compiled resources efficiently: `lookUpHash`, `restoreResource`, `tryResource`, `retryResource`, `trustResource`, `consume`, `executeClosure`, and `pickupResource`. Except for `lookUpHash`, each method corresponds to an extended LLPAM instruction. Fig. 5 shows code for executing an atomic goal A' with predicate symbol $p/2$. We will explain that code in detail.

In the `exec` method of predicate $p/2$, the `lookUpHash` method sets the registers `R1` and `R2` to the indices of possibly consumable primitive resources in `RES` by referring to the hash and symbol tables. `R1` is set to contain the indices of the resources which have the same predicate symbol `pred` and the same first argument in the head part of resource. `R2` is set to contain 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 these resources in `R2` because their heads are also possibly unifiable with the goal A'. The `pickupResource` method is used to check whether there are any consumable resources or not. That is, it finds an index value of a consumable resource with predicate symbol `pred` from `R1` (or `R2` if `R1` is `nil`) and sets that index value to the third argument register (`aregs[3]`). `R1` and `R2` are then updated to have the remaining resources. If there are no consumable resources, the control is passed to the ordinary program of A' immediately. The `tryResource` method behaves the same as WAM instruction *"try L"*, but `R1`, `R2`, and `engine.cont` (the register for continuation) are also saved in the created choice point. It sets the next clause field (`BP`) to point to `L0` and returns `L1`.

In `L1`, the `consume` method first marks the resource, which is pointed by the index value in `aregs[3]`, as consumed. It then returns the closure for the resource to `clo`. Continuously, the `executeClosure` method retrieves the free variables and the compiled code from the closure `clo`. After setting variables to certain argument registers, it returns the compiled code.

After the failure of one trail of resource consumption, control is passed to `L0` since `BP` was set to point to `L0` in `tryResource`. In `L0`, the `pickupResource` method is invoked again after the `restoreResource` method restores the register values from the current choice point. If there are no more consumable resources, the control is passed to `L2`. The `retryResource` method replaces the `R1` and `R2` values in the current choice point by their current values, and then returns `L1`.

In `L2`, the `trustResource` method discards the current choice point, and the control is passed to the ordinary program of A'.

An optimization design of LLPAM code for atomic goals was proposed in [5]. The resource consumption must be examined for every execution of an atomic goal, regardless of whether there exists an ordinary program or not. However, an atomic goal means only resource consumption when there is no ordinary predicate invocation. That optimization design is limited to this case, and its essence 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 before consuming the last consumable resource.

The above optimization idea can be implemented quite easily by inserting the following new method just after the `pickupResource` methods. Fig. 6 shows an optimized code corresponding to the fig. 5, where we omit code which is the same as fig. 5.

- `public final boolean hasMoreResource()`
  This method scans whether there are consumable resources in `R1` and `R2`. If there are no consumable resources, it fails.

Finally, we introduce a new declaration called *resource declaration* for separate compilation. Resource declaration is similar to dynamic declaration in Prolog. A resource declaration "`:- resource` p/n" means any predicate, even those which are not defined in same file, can add or consume resources with predicate symbol p/n. This idea is applicable not only to the Prolog Café implementation but also to the LLPAM.

4.5 An Implementation of `assert` *and* `retract`

Prolog Café is easily expandable with increasing Java's class libraries since all data structures are Java objects in our system. In this section, we present an implementation of `assert` and `retract` by using Java's hash table. In our design, each entry in the hash table contains a list of clauses. The entries are hashed on the predicate name/arity of the head part of clauses. The following built-in predicates for handling a hash table are used to implement `assert` and `retract` easily.

- `put_term(+Hash, +Key, ?Term)`
  This maps the key to the value of `Term` in the hash table. `Key` is a ground term for the hash key. We note that any unbound variables in the `Term` are not replaced by new private variables.
- `get_term(+Hash, +Key, ?Term)`
  This retrieves the value to which the key is mapped in the hash table and unifies it with `Term`. `Term` is unified with empty list if the key is not mapped to any value in the hash table.

The first argument `Hash` is implemented as `JavaObjectTerm` objects which contain a hash table. Let us note that multiple hash tables can be maintained in Prolog Café since hash tables are created as terms by the built-in predicate `java_constructor/2`.

Fig. 7 shows source code for `assert` and `retract`, where `user` represents the standard hash table in the Prolog Café system. The behavior of `assert/2` is straightforward. First, it takes the hash key of clause and extracts the list of clauses by referring to the hash table (`get_term/3`). Then, it creates a new list by inserting the target clause to the extracted list and registers a copy of the created list in the hash table (`put_term/3`). We note that we need to make a copy because unbound variables in clauses might be instantiated after the `assert`ion, or variable bindings might be canceled on backtracking. The behavior of `retract/2` is also straightforward so we omit the explanation here.

5 Performance Evaluation

In this section, we present the performances of the Prolog Café system. Prolog Café consists of the LLP to Java translator (written in SICStus Prolog, 2500 lines) and the LLP run-time system (written in Java).

5.1 Benchmark programs

Table 1 and 2 shows the performances of Prolog Café for Prolog and LLP benchmarks respectively. Except for JIT, the benchmarks were executed under Linux (MMX Pentium 266MHz, 128MB Memory) with JDK 1.1.7 with the `-O` option. Times using JIT were measured under Windows 98 (the same machine) with Symantec JIT compiler for JDK 1.1. We checked six items for each benchmark program: the number of lines of the LLP/Prolog source program, translation time (seconds) from LLP/Prolog to Java source code, the size (KBytes) of generated Java source code and its bytecode, and the execution time (seconds) with no-JIT and JIT. Two well known benchmarks `nand` and `simple_analyzer` could not be executed by the lack of built-in predicates: `recorda/3`, `recorded/3`, and `keysort/2`.

5.2 Comparison with the first implementation of Prolog Café

We present an improvement in performance by compiling resources compared with the first implementation, in which resource are represented as terms. The generated code for `domino` (presented in section 2.3) of Prolog Café is about 3 times faster than that of the first implementation. This speedup is due to the compilation of rule-type resources which have compound goal formulas as their body parts. However, the speedup of other benchmarks (in table 2) which contain only fact-type resources is about 10% on average.

Prolog programs	Lines	Translation time	Java code size	Bytecode size	Exec. time	Exec. time with JIT
`boyer`	395	182	430	755	86.445	38.950
`browse`	108	88	67	92	54.694	16.553
`chat_parser`	1180	923	789	1508	1.451	0.888
`crypt`	92	58	40	65	0.100	0.056
`fast_mu`	108	57	36	43	0.058	*sufficiently short*
`meta_qsort`	110	63	67	137	0.603	0.303
`mu`	49	21	31	37	0.043	*sufficiently short*
`nrev` (300 *elem.*)	31	15	12	15	0.957	0.358
`poly_10`	109	66	78	119	2.712	1.714
`prover`	104	55	62	84	0.085	*sufficiently short*
`qsort`	35	25	17	19	0.036	*sufficiently short*
`queens_8` (*all*)	95	27	29	39	2.502	1.044
`queens_10` (*all*)	95	27	29	39	57.766	21.244
`queens_16` ($first$)	95	27	29	39	36.177	14.094
`query`	87	35	49	185	0.509	0.110
`reducer`	384	234	263	344	2.515	1.428
`tak`	31	17	5	5	8.338	5.933
`unify`	157	329	133	167	0.176	0.088
`zebra`	53	87	28	26	1.404	0.418

Table 1
Prolog Café performances for Prolog benchmarks

LLP programs	Lines	Translation time	Java code size	Bytecode size	Exec. time	Exec. time with JIT
`color`	188	192	210	1583	120.466	30.814
`crypt`	30	89	29	24	0.497	0.189
`domino` ($2 \times 6, all$)	156	106	66	89	58.955	13.458
`fast_knight` ($6, first$)	207	202	84	103	94.303	29.286
`fast_knight` ($8, first$)	207	202	84	103	1484.629	438.527
`kirkman`	34	44	30	36	30.576	6.908
`knight5`	40	183	50	55	24.860	6.898
`peg`	94	93	39	56	175.072	39.734
`prop`	70	88	81	273	4.303	2.032
`queens_8` (*all*)	83	76	46	64	2.462	0.770
`queens_10` (*all*)	83	76	46	64	46.474	13.153
`queens_16` ($first$)	83	76	46	64	18.192	5.096
`triangle` ($4, all$)	94	50	40	54	6.930	2.241

Table 2
Prolog Café performances for LLP benchmarks

5.3 Comparison with Prolog systems implemented in Java

We compare Prolog Café with other Prolog systems implemented in Java.

Recently, a number of Prolog systems have been developed: BirdLand's Prolog in Java, CKI Prolog, DGKS Prolog, JavaLog, Jinni, JP, jProlog, LL, MINERVA, and W-Prolog. Among these systems, we choose two compiler systems:

- jProlog 0.1 (academic, B. Demoen and P. Tarau) [3]
  As mentioned in this paper, jProlog is the first Prolog-to-Java translator system based on binarization [15] and provided a basis for developing Prolog Café. jProlog supports intuitionistic assumption, backtrackable destructive assignment, and delayed execution. However, jProlog does not incorporate indexing and specialization of head unification. We note that some benchmark programs are slightly adapted to make jProlog deal with if-then-else, DCG, and so on.
- MINERVA 2.0 (commercial, IF Computer) [4]
  MINERVA is an efficient Prolog system implemented in Java. This implementation compiles Prolog into its own virtual machine code, and then executes it in Java. We use the evaluation version of MINERVA available on WWW.

Table 3 shows the execution speed of three systems and the average speedup (marked by $\uparrow$) or slowdown (marked by $\downarrow$) of Prolog Café. In jProlog, there are blank entries for `crypt`, `meta_qsort`, and `unify` because we had trouble in executing translated code for those programs.

Prolog Café generates code 2.2 times faster than jProlog on average. This speedup is almost entirely due to *switch_on_term*, and the specialization of head unification inherent in Prolog Café. Compared with MINERVA, Prolog Café is 2.2 times slower on average. Some of this slowdown is due to the overhead needed to support the LLP language. For instance, the values of not only the Prolog registers, but also three new registers must be stored every time a choice point frame is created. We measured the cost of this overhead, and it was 12% for Prolog benchmarks (in Table 1) on average.

However, in exchange for the overhead, Prolog Café provides several resource programming features based on an richer logic than Horn clause. Let us note that Prolog Café generates code for N-Queen (LLP program, $N > 9$, all solutions) faster than the `queens` benchmark (Prolog program in Table 3) which is compiled by MINERVA. Prolog Café is 1.2 times faster for 10-Queens and 1.5 times faster for 12-Queens, and the speedup of Prolog Café becomes larger as N increases. In Prolog Café, as resources are represented as formulas rather than terms, these resources are consumed efficiently through hash lookup. However, as resources are managed in a list structure in Prolog, we

[3] `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`

[4] `http://www.ifcomputer.com/MINERVA/`

Prolog programs	Prolog Café 0.43 JDK 1.1, no-JIT	MINERVA 2.0 JDK 1.1, no-JIT	jProlog 0.1 JDK 1.1, no-JIT
`boyer`	86.445	28.253	261.699
`browse`	54.694	19.554	228.146
`chat_parser`	1.451	0.340	2.077
`crypt`	0.100	0.151	——
`fast_mu`	0.058	0.074	0.139
`meta_qsort`	0.603	0.172	——
`mu`	0.043	0.017	0.052
`nrev` (300 *elem.*)	0.957	0.540	4.791
`poly_10`	2.712	1.058	3.924
`prover`	0.085	0.023	0.075
`qsort`	0.036	0.019	0.045
`queens_8` (*all*)	2.502	2.276	6.332
`queens_10` (*all*)	57.766	54.437	150.102
`queens_16` (*first*)	36.177	35.800	95.454
`query`	0.509	0.091	0.405
`reducer`	2.515	0.892	8.204
`tak`	8.338	64.287	13.867
`unify`	0.176	0.146	——
`zebra`	1.404	0.710	2.176
average speedup/slowdown of Prolog Café		$\downarrow$ 2.2	$\uparrow$ 2.2

Table 3
Prolog Café versus MINERVA and jProlog

not only scan the list to find a consumable resource, but also reconstruct the list when resources are consumed.

5.4 Comparison with an LLP compiler system

We compare Prolog Café with an LLP compiler system based on the LLPAM.

There have been several proposals for linear logic programming languages: Lolli [8], Lygon [7], LO [3], LinLog [2], Forum [11], and HACL [10]. Lolli, Lygon, and Forum are implemented as interpreter systems; Lolli is on SML and λProlog, Lygon is on Prolog, Forum is on SML, λProlog and Prolog. However, none of them are implemented as compiler systems. We therefore choose an LLPAM-based compiler system called LLP, which was the first compiler system for linear logic programming languages.

- LLP 0.43 (academic, N. Tamura et al.) [5]
  LLP is one of the most efficient systems for linear logic programming languages. LLP programs are compiled into LLPAM code and then executed by an emulator written in ANSI C. LLP version 0.43 has not incorporated

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

well-known optimizations (register allocation, last-call-optimization, and so on) and resource compilation yet.

LLP programs	Prolog Café 0.43	LLP 0.43
	JDK1.1, JIT	LLPAM code
`color`	30.814	2.640
`crypt`	0.189	0.020
`domino` $(2 \times 6, all)$	13.458	2.090
`fast_knight` $(6, first)$	29.286	3.108
`fast_knight` $(8, first)$	438.527	49.362
`kirkman`	6.908	0.910
`knight5`	6.898	0.660
`peg`	39.734	4.590
`prop`	2.032	0.070
`queens_8` (all)	0.770	0.054
`queens_10` (all)	13.153	1.072
`queens_16` $(first)$	5.096	0.400
`triangle` $(4, all)$	2.241	0.270
average slowdown of Prolog Café with JIT		↓ 10.9

Table 4
Prolog Café versus LLP

Prolog programs	Prolog Café 0.43	SICStus Prolog 3.7.1	SWI-Prolog 3.2.6
	JDK 1.1, JIT	WAM code	WAM code
`boyer`	38.950	0.610	2.720
`browse`	16.553	0.773	2.272
`poly_10`	1.714	0.047	0.153
`queens_8` (all)	1.044	0.057	0.320
`queens_10` (all)	21.244	1.270	7.650
`queens_16` $(first)$	14.094	0.780	5.040
`reducer`	1.428	0.037	0.096
`tak`	5.933	0.163	50.692
`zebra`	0.418	0.042	0.070
average slowdown of Prolog Café with JIT		↓ 28.9	↓ 6.9

Table 5
Prolog Café versus SICStus Prolog and SWI-Prolog

Table 4 shows the LLP benchmark results of the two systems. Prolog Café with JIT is 10.9 times slower than LLP on average. Some of the slowdown is due to the expense of Java's "new" operator which results from binarization. That is to say, in our translation method, all goals in the body part of the clause are translated into one goal with continuations due to binarization. Therefore, when a clause is called, all goals in its body part must be created at execution time even if the first goal fails. This slows down the execution speed for LLP/Prolog programs.

To improve this point, we are currently investigating a new translation method based on Prolog's box control flow model. The main difference from our method in this paper is the treatment of goals. Our new approach does not use binarization transformation, and each goal is created and executed after its previous goal succeeds. Through this process, we can keep down excess expense due to Java's "new" operator. Let us note that the execution speed of experimental translated code for N-Queen, which follows our new translation method, is 1.7 times faster than Prolog Café.

We supplement the explanation of LLP performances here. LLP is at least 50 times faster than the Lolli interpreter on SML. LLP generates code for N-Queen (LLP program, all solutions) faster than the Prolog program (in "The art of Prolog") which is compiled into WAM compact code by SICStus Prolog version 3.7.1. Let us note that LLP is 1.2 times faster for 10-Queens and 1.5 times faster for 12-Queens, and the speedup of LLP becomes larger as N increases. However, the execution speed of the compiled code for Prolog programs is about 2 or 3 times slower than the SICStus Prolog WAM code.

Finally, we compare Prolog Café with high performance Prolog compilers: SICStus Prolog and SWI-Prolog. Table 5 shows Prolog benchmark results of three systems. We omit benchmark results where the execution time of compiled code by SICStus Prolog was sufficiently short in 5 trials. Prolog Café with JIT is on average 28.9 and 6.9 times slower than SICStus Prolog and SWI-Prolog respectively. Some of this slowdown is, again, due to the overhead necessary to support the LLP language and Java's "new" operator as mentioned above.

6 Conclusion and Future Plans

In this paper, we described a translation method from a linear logic programming language called LLP into Java. Particularly, we focused on Java implementation for compiling resources. We also described the performances of the resulting system called Prolog Café in detail.

As a result, the generated code for `domino` under Prolog Café is about 3 times faster than that of the first implementation. This means that compiling resources is about 3 times faster for `domino` than representing resources as terms, executed under an interpreter.

For a set of classical Prolog benchmarks, Prolog Café is 2.2 times faster than jProlog, the first Prolog to Java translator. Although Prolog Café is 2.2 times slower than MINERVA on average, it provides several resource programming features based on an richer logic than Horn clause.

The following points are still remaining:

- Generated code size is large

- Slow compilation speed
- $\forall x.G$ is allowed as a goal formula
- Complete treatment of $\top$
- No floating point numbers
- Implementing box control flow model in Java

To improve performance, we are currently developing a new version of Prolog Café based on Prolog's box control flow model.

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] M. Banbara and N. Tamura. Java implementation of a linear logic programming language. In *Proceedings of the 10th Exhibition and Symposium on Industrial Applications of Prolog*, pages 56–63, Oct. 1997.

[5] M. Banbara and N. Tamura. Compiling resources in a linear logic programming language. In *Proceedings of Post-JICSLP'98 Workshop on Parallelism and Implementation Technology for Logic Programming Languages*, June 1998.

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

[7] 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.

[8] 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.

[9] 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.

[10] N. Kobayashi and A. Yonezawa. Typed higher-order concurrent linear logic programming. Technical Report 94-12, University of Tokyo, 1994.

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

[12] 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.

[13] 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.

[14] 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`.

[15] P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.

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

```
%%%  Solver
solve_domino(M, N) :-
        D is M*N//2,
        row(M) => column(N) => num_of_dominoes(D) =>
        (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).
```

Fig. 1. LLP example: tiling board with dominoes

```
public class PRED_p_2 extends Predicate{
    static SymbolTerm Nil  = symbol [];
    static SymbolTerm symT = symbol true;
    static SymbolTerm symQ = symbol q;
    Term arg1, arg2;
    .....
    public PRED_p_2(Term a1,Term a2,Predicate cont){
        arg1 = a1;
        arg2 = a2;
        this.cont = cont;
    }
    private boolean clause1() { Code for 1st clause }
    private boolean clause2() { Code for 2nd clause }
    private boolean clause3() { Code for 3rd clause }
    public void exec(){
        if(clause1()) return;
        if(clause2()) return;
        if(clause3()) return;
    }
    .....
}

private boolean clause1(){
  .....
    try{
        if(arg1.unify(a1,trail) && arg2.unify(a2,trail)){
            .....
            p1 = new PRED_end_imp_2(a3, a4, cont);
            p2 = new PRED_r_1(a2, p1);
            p3 = new PRED_mid_imp_1(a4, p2);
            Term[] a5 = { a1 };
            a6 = new StructureTerm(symQ, a5);
            p4 = new PRED_add_res_3(a6, symT, Nil, p3);
            p5 = new PRED_begin_imp_1(a3, p4);
            p5.exec();
        }
    } catch (CutException e) {
        if(e.id != this)
            throw e;
        return true;
    } finally { trail.undoAll(); }
    return false;
}
```

Fig. 2. Code for predicate $p/2$ in 1st. implementation of Prolog Café

```
public class PRED_p_2 extends Predicate{
    static final Predicate resQ = new RES_q_1();
    static final SymbolTerm symQ = symbol q/1;
    .....
    public final Predicate exec(){
        a1 = arg1.dereference();
        a2 = arg2.dereference();
        .....
        Term[] h2 = { a1 };
        a4 = new StructureTerm(symQ, h2); // create head q(X)
        Term[] h3 = { a1 };
        a5 = new ClosureTerm(resQ, h3); // create closure
        p1 = new PRED_end_imp_2(a3, a6, cont);
        p2 = new PRED_r_1(a2, p1);
        p3 = new PRED_mid_imp_1(a6, p2);
        p4 = new PRED_add_prim_res_2(a4, a5, p3); // add q(X)
        return new PRED_begin_imp_1(a3, p4);
    }
}

public final class RES_q_1 extends Predicate{
    public RES_q_1(){}
    .....
    public final Predicate exec(){
        Term a1, a2;
        a1 = engine.aregs[1].dereference();
        a2 = engine.aregs[2].dereference(); // retrieve X
        // unify the 1st argument with  X
        if ( !a1.unify(a2, engine.trail) )
            return engine.fail(); // backtracking
        return engine.cont;
    }
}
```

Fig. 3. Code for `p(X,Y) :- q(X) -<> r(Y)` in Prolog Café

```
static Predicate resP = new RES_p_2();
static SymbolTerm symP = symbol p/2;
public final Predicate exec(){
    ...
    Term[] args = {varX, new VariableTerm()};
    a3 = new StructureTerm(symP, args); // create head p(X,Y)
    Term[] vars = {varX};
    a4 = new ClosureTerm(resP, vars); // create closure
    p1 = new PRED_end_imp_2(a2, a5, cont);
    p2 =  Code for the goal G;
    p3 = new PRED_mid_imp_1(a5, p2);
    p4 = new PRED_add_prim_res_2(a3, a4, p3);// add ∀ Y((q(X) ⊸  r(Y)) ⊸  p(X,Y))
    return new PRED_begin_imp_1(a2, p4);
}

public final class RES_q_1 extends Predicate{
    ...
    public final Predicate exec(){
        ...
        a1 = engine.aregs[1].dereference();
        a2 = engine.aregs[2].dereference(); // retrieve X
        this.cont = engine.cont;
        if ( !a1.unify(a2, engine.trail) ) // unify the 1st argument with X
            return engine.fail();
        return cont;
    }
}

public final class RES_p_2 extends Predicate{
    static Predicate resQ = new RES_q_1();
    static SymbolTerm symQ = symbol q/1;
    ...
    public final Predicate exec(){
        ...
        a1 = engine.aregs[1].dereference();
        a2 = engine.aregs[2].dereference();
        a3 = engine.aregs[3].dereference(); // retrieve X
        this.cont = engine.cont;
        if ( !a1.unify(a3, engine.trail) ) // unify the 1st argument with X
            return engine.fail();
        ...
        Term[] args = {a1};
        a5 = new StructureTerm(symQ, args); // create head q(X)
        Term[] vars = {a1};
        a6 = new ClosureTerm(resQ, vars); // create closure
        p1 = new PRED_end_imp_2(a4, a7, cont);
        p2 = new PRED_r_1(a2, p1);
        p3 = new PRED_mid_imp_1(a7, p2);
        p4 = new PRED_add_prim_res_2(a5, a6, p3); // add q(X)
        return new PRED_begin_imp_1(a4, p4);
    }
}
```

Fig. 4. Code for the goal $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y)) \multimap G$

```
public class PRED_p_2 extends Predicate{
    static SymbolTerm pred = predicate symbol p/2;
    static Predicate L  = ordinary program code for p/2;
    static Predicate L0 = new L0();
    static Predicate L1 = new L1();
    static Predicate L2 = new L2();
    public Term arg1, arg2;

    public PRED_p_2(Term a1, Term a2, Predicate cont){
        arg1 = a1; arg2 = a2; this.cont = cont;
    }
    public Predicate exec(){
        engine.setB0(); // set cut point
        engine.aregs[1] = arg1;
        engine.aregs[2] = arg2;
        engine.cont = cont;
        engine.lookUpHash(pred);
        if (!engine.pickupResource(pred, 3))
            return L;
        return engine.tryResource(L1, L0);
    }
}

final class L0 extends PRED_p_2{
    public final Predicate exec(){
        engine.restoreResource();
        if (!engine.pickupResource(pred, 3))
            return L2;
        return engine.retryResource(L1, L0);
    }
}

final class L1 extends PRED_p_2{
    public final Predicate exec(){
        ClosureTerm clo = engine.consume(3);
        return engine.executeClosure(clo);
    }
}

final class L2 extends PRED_p_2{
    public final Predicate exec(){
        return engine.trustResource(L);
    }
}
```

Fig. 5. Code for $p/2$

```
public class PRED_p_2 extends Predicate{
    static SymbolTerm pred = predicate symbol p/2;
    static Predicate L0 = new L0();
    static Predicate L1 = new L1();
    static Predicate L2 = new L2();
    static Predicate L3 = new L3();     // added
    ...
    public Predicate exec(){
        ...
        if (!engine.pickupResource(pred,3))
            return engine.fail();       // changed
        if (!engine.hasMoreResource())  // added
            return L1;
        return engine.tryResource(L1, L0);
    }
}

final class L0 extends PRED_p_2{
    public final Predicate exec(){
        ...
        if (!engine.pickupResource(pred, 3))
            return L2;
        if (!engine.hasMoreResource())  // added
            return L3;
        return engine.retryResource(L1, L0);
    }
}

final class L3 extends PRED_p_2{     // added
    public final Predicate exec(){
        return engine.trustResource(L1);
    }
}
```

Fig. 6. Optimized code for $p/2$ when there is no ordinary predicate invocation

```
assert(Clause) :- assertz(user, Clause).
retract(Clause) :- retract(user, Clause).

assertz(Hash, Clause) :-
        canonical_clause(Clause, Key, Cl),
        get_term(Hash, Key, Cls0),
        copy_term([Cl|Cls0], Cls),
        put_term(Hash, Key, Cls).

retract(Hash, Clause) :-
        canonical_clause(Clause, Key, Cl),
        get_term(Hash, Key, Cls0),
        copy_term(Cls0, Cls1),
        select_in_reverse(C, Cls1, Cls),
        C = Cl,
        put_term(Hash, Key, Cls).
```

Fig. 7. An implementation of `assert` and `retract`