Efficient Implementation of a Linear Logic Programming Language

Joshua S. Hodas and Kevin M. Watkins
Department of Computer Science
Harvey Mudd College
Claremont, CA 91711, USA
`hodas@cs.hmc.edu`, `kwatkins@cs.hmc.edu`

Naoyuki Tamura and Kyoung-Sun Kang
Department of Computer and Systems Engineering
Kobe University
Rokkodai, Nada, Kobe 657-0013, Japan
`tamura@kobe-u.ac.jp`, `kang@pascal.seg.kobe-u.ac.jp`

Abstract

A number of logic programming languages based on Linear Logic [3] have been proposed. However, the implementation techniques proposed for these languages have relied heavily on the copying and scanning of large data structures to ensure that linearity conditions have been met, and thus have been most suited to the creation of interpreters written in high-level languages. In this paper we present a new formulation of the logic of the language Lolli that is conducive to the creation of both more efficient interpreters, as well as compilers based on extensions of the standard WAM model. We present such an extension which implements a useful fragment of Lolli. Resource conscious algorithms executed using this system show significant performance advantages relative to their traditional Prolog implementations.

1 Introduction

Implementing logic programming languages based on Linear Logic [3] presents a number of problems, as the constraints on the use of assumptions must be carefully maintained. The issue of *resource management* has been discussed from the earliest proposals [8], and recently several papers have focused exclusively on these issues [2, 6, 4]. However, most of the methods proposed for controlling resource consumption have been specified in a manner that requires the manipulation of large structures. This makes them more suited to implementation by interpreters written in high-level languages, rather than by compilers. Indeed, all implementations of these languages to date have been interpreter-based.

In this paper we first introduce some of the basic concepts of linear logic programming. We discuss some of the basic issues in resource management, and then show how a recent treatment of the logic underlying the language Lolli can be reformulated to provide the foundations of a compiler for a fragment of the language. We present the specification of the compiler for the fragment, which we call LLP, to an extension of the Warren Abstract Machine (WAM). Finally, we show that, for the first time, the linear logic programming solution to a problem can be executed more efficiently that the traditional logic programming solution.

2 Linear Logic Programming

In linear logic the *structural rules* of weakening and contraction are available only for assumptions marked with the modal (or, as it is called in linear logic, *exponential*) "!". This means that, in general, a formula not thus marked can only be used once in a branch of the search for a proof.

2.1 Reasoning With Limited Resources

Limited-use formulas, which we will refer to as *linear resources* (we will refer to assumptions marked with the ! as *intuitionistic resources*) can represent limited resources in some domain. For example, in linear logic, the following formula (which is a tautology in traditional systems) would not be provable:

$$((\mathit{dollar} \supset \mathit{pizza}) \land (\mathit{dollar} \supset \mathit{soda})) \supset (\mathit{dollar} \supset (\mathit{pizza} \land \mathit{soda}))$$

The reason is that the *dollar* assumption provided by the formula $(\mathit{dollar} \supset (\mathit{pizza} \land \mathit{soda}))$ cannot be used twice to drive both the rules on the left. The linear logic programming language Lolli [8] has been developed implements these constraints on the use of assumptions.

In many Prolog programs a list is used to store a set of resources on which the program relies, and this set is managed in a manner corresponding to the constraint on assumptions in linear logic. Consider the solution to the N-Queens problem given in The Art Of Prolog [10], which is shown on the left side of Figure 1. In this program, a list is used to store terms corresponding to the rows of the chess board. For each column of the board, we `select` an unused row and test if it is safe to place the queen there. The list is managed to insure that each row is only used once.

The same basic technique is used in the Lolli program on the right side of Figure 1. The only difference is that resources are used to represent rows, columns, and diagonals. This is done simply because it is easy to do in this setting.

2.2 The Operators of Linear Logic

In the absence of contraction and weakening, many of the logical operators split into two variants. For example, the conjunction operator splits into *tensor*, "$\otimes$", and *with*, "&". In proving a conjunction formed with $\otimes$, the

```
queen_all(N) :-queens(N, Q),fail.
queen_all(_).

queens(N, Qs) :- range(1, N, Ns),
                 queens(Ns, [], Qs).

queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
    select(Q,UnplacedQs,UnplacedQs1),
    \+ attack(Q,SafeQs),
    queens(UnplacedQs1,[Q|SafeQs],Qs).

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

attack(X, Xs) :- attack(X, 1, Xs).
attack(X, N, [Y|Ys]) :- X is Y + N ;
                        X is Y - N.
attack(X, N, [Y|Ys]) :-N1 is N + 1,
                        attack(X, N1, Ys).

range(M, N, [M|Ns]) :- M < N, M1 is M + 1,
                       range(M1, N, Ns).
range(N, N, [N]).
```

```
queen_all(N) :- queens(N, Q), fail.
queen_all(_).

queens(N, Q) :-n(N) -o result(Q) -o
                      rangeQueens(N).

rangeQueens(1) :-
    n(N),  (c(1),u(2),d(0)) -o
                 solve(N, []).
rangeQueens(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)) -o
        rangeQueens(I1).

solve(0, Q) :-result(Q), erase.
solve(I, Q) :-
   I > 0,c(J),U is I+J,u(U),
```

Figure 1: **Left:** Resource-driven N-Queens program in Prolog from Art Of Prolog, Chap. 14. **Right:**Resource-driven N-Queens program in Lolli

current set of assumptions is split between the two conjuncts: those not used in proving the first conjunct must be used while proving the second. To prove a & conjunction, the set of assumptions is copied to both sides: each conjunct's proof must use all of the assumptions. In Lolli, the $\otimes$ conjunction is represented by the familiar "`,`". This is a natural mapping, as we expect the effect of a succession of goals to be cumulative. Each has available to it the resources not yet used by its predecessors. The & conjunction, which is less used, is written "`&`".

Though there are two forms of disjunction, only one, "$\oplus$" is used in Lolli. There are two forms of truth, $\top$, and $\mathbf{1}$. The latter, which Lolli calls "`true`", can only be proved if all the assumptions have already been used. In contrast, the formula $\top$ is true even if some resources are, as yet, unused. This operator, which Lolli calls "`erase`", can be used to reintroduce a limited form of weakening. If a $\top$ occurs as one of the conjuncts in a $\otimes$ conjunction, then the conjunction may succeed even if the other conjuncts do not use all the linear resources. The $\top$ is seen to consume the leftovers.

A goal marked with the exponential "!" can be proved only if all the assumptions are intuitionistic, unlimited, ones. Rather than use a combination of this operator and a linear implication to load an intuitionistic resource, Lolli instead uses an unadorned resource, paired with the intuitionistic implication operator, $\Rightarrow$, depicted as "`=>`" to load unlimited resources.

It is beyond the scope of this paper to demonstrate the applications of all these operators. Many good examples can be found in the literature, particularly in the papers on Lygon and Lolli [5, 8].

3 The I/O Implementation Model

The management of resources during proof search is a serious problem for the implementor. A naive implementation of the rules for the linear logic operators would lead to a level of backtracking that would render the system useless. Consider, for example, the rule for proving the goal $G_1 \otimes G_2$:

$$\frac{\Delta_1 \longrightarrow G_1 \quad \Delta_2 \longrightarrow G_2}{\Delta_1, \Delta_2 \longrightarrow G_1 \otimes G_2} \otimes_R$$

When the system needs to use this rule during the bottom-up search for a proof, the assumptions have not yet been divided into sets Δ_1 and Δ_2. If the generate-and-test paradigm were used to find an appropriate partition of the assumptions, 2^n possibilities might need to be tested.

The solution, developed by Hodas and Miller in the first paper on Lolli [8], is to treat the management of resources lazily, allowing the conjuncts to determine the partition by their demand for resources. Hodas and Miller described the new execution model by reformulating the rules of linear logic so as to expose the intent of the model. For example, their rule for the $\otimes$ conjunction was:

$$\frac{I\{G_1\}M \quad M\{G_2\}O}{I\{G_1 \otimes G_2\}O} \otimes_R$$

in which the goal is bracketed and the context on the left is the *input context* and the context on the right is the *output context*. This formulation has come to be known as the I/O model for linear logic. Its system of rules is logically equivalent to the original rules: a formula is provable in a given context in the original system if an only if it is provable with that context as input, and with the empty context as output, in this system. However, the rules suggest a certain implementation method.

The I/O model has been refined several times, addressing various sources of non-determinism [7, 6, 2]. Recently, Cervesato, Hodas, and Pfenning have proposed a refinement, which they refer to as $\mathcal{RM}_3$ (for *resource management system 3*) which pays particular attention to the management of resources across a &. In the original I/O system, the rule for & was written:

$$\frac{I\{G_1\}O \quad I\{G_2\}O}{I\{G_1 \,\&\, G_2\}O} \&_R$$

which states that the conjuncts must use the same set of resources. (If the goal is the top-level goal, they must both use the entire context, leaving the output empty.) In a naive implementation the two conjunct are proved separately, and their outputs compared. This leads to unnecessary backtracking.

Cervesato, Hodas, and Pfenning reformulated the rule for & as two rules, differing in whether a $\top$ is seen in the first conjunct.

$$\frac{S|I\{G_1\}_0 O \quad (S, I - O)|[]\{G_2\}_v[]}{S|I\{G_1 \,\&\, G_2\}_0 O}\,\&_{0v} \qquad \frac{S|I\{G_1\}_1 M \quad (S, I - M)|M\{G_2\}_v O}{S|I\{G_1 \,\&\, G_2\}_v O}\,\&_{1v}$$

There are two main changes to the form of I/O rules. The flag appearing as the subscript on the right of the goal indicates whether a $\top$ was seen in the goal. The extra context added on the input side is the *strict* context. Resources in this context *must* be used in the proof of the goal. The other input context is now referred to as the *lax* context.

Thus, when the first conjunct does not encounter a $\top$, the second conjunct must consume all resources that the first had to (its strict context, S), as well as all the lax resources that the first did. All of these appear in the strict context of the second conjunct. The second conjunct has an empty lax context, since it may not consume anything the first did not. In the case where the first conjunct encounters a $\top$, the unconsumed part of the lax context is given as lax context to the second conjunct, since, if it chooses to use any of those formulas, they may be considered to have been also consumed in the first conjunct by the $\top$. The use of strict and *lax* contexts and the $\top$ flag, permeates the rest of $\mathcal{RM}_3$.

4 The $\mathcal{LRM}$ Level-Based Implementation Model

$\mathcal{RM}_3$ provides an efficient model of proof search for Lolli and related languages, at least as regards minimizing needless backtracking. However, the way in which it is formulated makes it best suited to implementation via interpreters written in high-level languages. In particular, the need to move formulas around between strict and lax contexts and perform operations such as intersection on contexts require manipulating large dynamic structures. While the authors remark that efficient implementation using such techniques as bit-vectors representing resource consumption should be possible, they supply no real guidance.

Tamura and Kaneda have suggested using a system of *level indices* to control the consumption of resources [11]. That system handled only a small fragment of linear logic, and was logically incomplete in its treatment of $\top$. In Figure 2 we present an enriched formulation of that system, which we call $\mathcal{LRM}$ for *Level-Based Resource Management System.* This system faithfully reproduces the logic and most of the search behavior of $\mathcal{RM}_3$. However, it is described so as to make the method of efficient implementation apparent.

The sequent system $\mathcal{LRM}$ makes use of two level indices, L and U, to manage consumption of resources. (These indices adorn the turnstile symbol, along with the $\top$-flag carried over from $\mathcal{RM}_3$.) L, a positive integer, is the *current consumption level.* At a given point in the proof tree, only resources labeled with that consumption level may be used. L is also used to set a deadline by which newly added resources must be used. U, a negative integer, is the *current consumption marker*. When a resource is used, its consumption level is changed to this value as an indicator of when it was consumed.

Each formula in the context is adorned with a super and subscript. The formulas can be classified by the values of these two fields: *Linear unconsumed* formulas have the form R_l^d, where l is the value of L at which the resource may be consumed, and d is the *smallest* value of L at which the resource may exist without having been consumed. If any resources exist with $d \geq L$ when it is time to decrement the proof level to $L-1$, the solver will either backtrack (in the strict case) or consume those resources immediately (in the lax case). *Linear consumed* formulas have the form R_u^0, where u is the (negative) value of U at the time the resource was consumed. *Intuitionistic* formulas always take the form R_0^0.

The **pickr** rule handles the selection of clauses for backchaining. The relation $\texttt{pickR}_{L,U}(A, G, I, M)$ used in that rule is defined as a relation between an input list and atom, I and A, and an output list and G-formula, M and G. The relation selects an available (i.e. linear unconsumed, or intuitionistic) clause from the list I matching the atom A. The output goal G is the body of the clause (or $\mathbf{1}$ if the clause is atomic). The output list M is the same as I, but with the selected clause marked as linear consumed (with $u = U$) if it was linear unconsumed.

The two $\mathcal{RM}_3$ rules for & are split into four rules in $\mathcal{LRM}$, due to the need for different bookkeeping depending on whether $\top$ is seen in the right conjunct. Three relations define the "side conditions" used in the rules for &. Each can be implemented destructively in one pass through the context in an actual implementation:

- Let L be a positive integer. Then $\texttt{consumed}_L$ is a predicate on a list of R-formulas that is true if none of the R-formulas in the list have L as their deadline.
- Let L and L' be integers. Then $\underset{L \to L'}{\texttt{change}}$ is a relation between an input and output list of R-formulas that modifies the input list so that any R-formula in the list with its level, l, set to L has the level changed to L' in the output.
- Let L, L', D, and D' be integers. Then $\underset{(L,D) \to (L',D')}{\texttt{changepair}}$ is a relation between an input and output list of R-formulas that modifies the input list so that any R-formula in the list with its level, l, set to L and its deadline, d, set to D, has the level changed to L' and its deadline changed to D' in the output.

We will not present a detailed discussion of the rules here. The system has, however, been proved to be logically equivalent to $\mathcal{RM}_3$ [9]. The behavior of the rules can be understood by a careful reading of the description of their implementation in Section 6, below.

$$\frac{}{\vdash^{0}_{L,U} I\{\mathbf{1}\}I}\ \mathbf{1} \qquad \frac{}{\vdash^{1}_{L,U} I\{\top\}I}\ \top \qquad \frac{\vdash^{v}_{L+1,U-1} I\{G\}O}{\vdash^{0}_{L,U} I\{!G\}O}\ !$$

$$\frac{\vdash^{v_1}_{L,U} I\{G_1\}M \quad \vdash^{v_2}_{L,U} M\{G_2\}O}{\vdash^{v_1 or v_2}_{L,U} I\{G_1 \otimes G_2\}O}\ \otimes \qquad \frac{\mathtt{pickR}_{L,U}(A,G,I,M) \quad \vdash^{v}_{L,U} M\{G\}O}{\vdash^{v}_{L,U} I\{A\}O}\ \mathbf{pickR}$$

$$\frac{\begin{array}{c} \vdash^{0}_{L,U-1} I\{G_1\}M \qquad \underset{(U-1,0)\to(L+1,L+1)}{\mathtt{changepair}}\ (M,M') \\ \vdash^{0}_{L+1,U} M'\{G_2\}O \quad \mathtt{consumed}_{L+1}(O) \end{array}}{\vdash^{0}_{L,U} I\{G_1 \,\&\, G_2\}O}\ \&_{\mathbf{00}}$$

$$\frac{\begin{array}{c} \vdash^{0}_{L,U-1} I\{G_1\}M \qquad \underset{(U-1,0)\to(L+1,L+1)}{\mathtt{changepair}}\ (M,M') \\ \vdash^{1}_{L+1,U} M'\{G_2\}O' \qquad \underset{(L+1,L+1)\to(U,0)}{\mathtt{changepair}}\ (O',O) \end{array}}{\vdash^{0}_{L,U} I\{G_1 \,\&\, G_2\}O}\ \&_{\mathbf{01}}$$

$$\frac{\begin{array}{c} \vdash^{1}_{L,U-1} I\{G_1\}M \qquad \underset{(U-1,0)\to(L+1,L+1)}{\mathtt{changepair}}\ (M,M') \qquad \underset{L\to L+1}{\mathtt{change}}\ (M',M'') \\ \vdash^{0}_{L+1,U} M''\{G_2\}O' \quad \mathtt{consumed}_{L+1}(O') \end{array}}{\vdash^{0}_{L,U} I\{G_1 \,\&\, G_2\}O}\ \&_{\mathbf{10}}$$

$$\frac{\begin{array}{c} \vdash^{1}_{L,U-1} I\{G_1\}M \qquad \underset{(U-1,0)\to(L+1,L+1)}{\mathtt{changepair}}\ (M,M') \qquad \underset{L\to L+1}{\mathtt{change}}\ (M',M'') \\ \vdash^{1}_{L+1,U} M''\{G_2\}O'' \qquad \underset{(L+1,L+1)\to(U,0)}{\mathtt{changepair}}\ (O'',O') \qquad \underset{L+1\to L}{\mathtt{change}}\ (O',O) \end{array}}{\vdash^{1}_{L,U} I\{G_1 \,\&\, G_2\}O}\ \&_{\mathbf{11}}$$

$$\frac{\vdash^{v}_{L,U} [R^L_L|I]\{G\}[R^0_U|O]}{\vdash^{v}_{L,U} I\{R \multimap G\}O}\ \multimap \qquad \frac{\vdash^{1}_{L,U} [R^L_L|I]\{G\}[R^L_L|O]}{\vdash^{1}_{L,U} I\{R \multimap G\}O}\ \multimap_{\mathbf{1}} \qquad \frac{\vdash^{v}_{L,U} [R^0_0|I]\{G\}[R^0_0|O]}{\vdash^{v}_{L,U} I\{R \Rightarrow G\}O}\ \Rightarrow$$

Figure 2: $\mathcal{LRM}$: A resource management system with deadlines

5 LLP: A Compiled Linear Logic Programming Language

While $\mathcal{LRM}$ can serve as the foundation of a Lolli interpreter, the design of the system was undertaken with a greater goal in mind: the creation of a compiler for a significant fragment of Lolli. That goal has been met in LLP.

5.1 The Definition of LLP

LLP is based on the following fragment of linear logic (where A is an atomic formula):

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

The sets of allowed clauses and goals are subsets of the allowed formulas of Lolli. The principal limitations relative to the full language of Lolli are:

- The clauses of the initial program, denoted by the set C, are all !'ed. That is, the program may not contain any linear clauses.
- The resources added dynamically with the $\multimap$ and $\Rightarrow$ operators are either atomic formulas, or products (formed with &) of atomic formulas. Lolli allows arbitrary clause formulas to be added.
- Lolli allows nested quantification and the use of L_λ higher-order quantification and unification of λ-terms.

In spite of these limitations, the fragment covered by LLP is large enough to cover a broad and important class of Lolli programs. In particular, programs like the N-Queens solution presented in Section 2, in which the linear context is used simply to store data values being manipulated by the rest of the program, are included in the fragment covered by LLP.

Note that, as an informal extension to this language, we will allow a $\otimes$-product of multiple R-formulas to appear on the left of either implication operator. Such a formula is treated as though it were written with successive uses of the implication operator loading each individual R-formula.

6 The LLPAM Abstract Machine

LLP is implemented as a compiler to the LLP Abstract Machine (LLPAM), an extension of the Warren Abstract Machine (WAM) [12, 1]. All the data and control structures of the WAM are retained. Further, when executing LLP programs that do not make use of the resource management features of the language, at worst a small constant factor overhead is incurred by the new structures. This because the only added work in such a case is to record four new registers in each choicepoint, and to confirm on predicate dispatch that there are no relevant resources in the (always empty) resource table.

6.1 New Registers

The LLPAM has four new registers: `R`, `L`, `U`, and `T`:

- `R` is a non-negative integer index indicating the current top of the resource table, a new data area which is described below. The value of `R` increases as resources are added to the table by implicational goals, and decreases on backtracking. Its initial value is 0.
- `L` holds the current value of L as used in $\mathcal{LRM}$. It indicates the current consumption level in the proof tree. Its initial value is 1.
- `U` holds the current value of U as used in $\mathcal{LRM}$. This is the value assigned to resources as they are consumed. Its initial value is -1.
- `T` is a boolean flag indicating whether $\top$ has been seen as a subgoal at the current level. Its initial value is false.

6.2 New Data Areas

The LLPAM has three new data areas: `RES`, `HASH`, and `SYMBOL`:

6.2.1 The Resource Table

`RES` is used to store the formulas added to the program dynamically with implicational goals. It grows when resources are added by $\multimap$, or $\Rightarrow$, and shrinks on backtracking. An entry in `RES` can represent either a linear resource, or an unlimited one, depending on which implication operator was used to load it. Each entry corresponds to a single R-formula: either an atom, or a &-product of atoms. Note that by representing all the subformulas in a compound resource formula by a single resource table entry, the exclusivity of their consumption is maintained. If one of the atoms is used, the entry is so marked and the other atoms &'ed to it become unavailable.

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

```
record
  head: term;
  level: integer;
  deadline: integer;
  out_of_scope: Boolean;
end;
```

The fields of the record are assigned as follows:

- The atoms in the resource formula to which this entry in `RES` corresponds are stored as a list of terms on the `HEAP`. The field `head` contains a pointer to that structure.
- The fields `level` and `deadline` correspond to the l and d values attached to resources in $\mathcal{LRM}$. For linear resources the initial values of both fields are taken from the value of the `L` register. For exponential (unlimited use) resources, the two fields are set to 0.
- The `out_of_scope` bit is initially false. It is set to true when the goal underneath the implication that loaded this resource is completed, and the resource, therefore, goes out of scope. This flag is used because the resource table shrinks only on backtracking.

6.2.2 The Hash and Symbol Tables

Two additional structures are maintained to speed access to the resources in `RES`. `HASH` is a hash table of the resources added to `RES`. The entries are hashed on the predicate name/arity, and the first argument.

We cannot always rely on the hash table for access to the resources. When the goal atom has a logic variable in the first argument position, we must access all entries for the given predicate symbol, regardless of first argument. Similarly, those resources in which the first argument is a logic variable must be examined for every call on that predicate symbol. Therefore the entry for a predicate symbol in the symbol table, `SYMBOL`, contains a list of pointers to all resources with that predicate name/arity, and a list of pointers to all resources with that predicate name/arity and an unbound logic variable as its first argument.

Note that, in the case of an entry in `RES` corresponding to a &-product of atoms, multiple entries in `HASH` and `SYMBOL` will point to that entry in `RES`.

6.3 LLPAM Code Generation

The code generated for each operator in a goal represents an imperative implementation of the declarative $\mathcal{LRM}$ rule (or rules) for the operator. In each case the relationship between code and rule should be clear. The code for head unification remains the same as in the WAM, though some steps are added between the point that a `call` is issued, and the point that the block for the predicate is entered. These changes will be discussed below.

6.3.1 Code for $G_1 \otimes G_2$

The tensor product, $\otimes$, takes the place of "ordinary" conjunction in this setting. In accordance with the $\mathcal{LRM}$ rule for $\otimes$, the effect of the two conjuncts on the resource table and the $\top$-flag is merely the accumulation of their individual effects. Therefore, the code generated for $G_1 \otimes G_2$ is simply:

Code for G_1
Code for G_2

6.3.2 Code for $R \multimap G$ or $R \Rightarrow G$

An implicational goal, $R \multimap G$ or $R \Rightarrow G$, requires adding the resource R to the resource table and then executing the goal G in the augmented context. Further, if the $\multimap$ implication is used, then the resource R must be used during the proof of G.

The following new instructions are used in the LLPAM in the code generated for an implication of the form $R \multimap G$ or $R \Rightarrow G$:

- `begin_imp Yi`: Stores the current value of register `R` in a new permanent variable $\texttt{Y}_i$.
- `add_res Ai`: Used when the implication operator is $\multimap$. Adds a record for a primitive resource of the form A or $A_1 \& \ldots \& A_n$ as a new entry at the top of the resource table, `RES`. `Ai` is a pointer to a structure previously built on the heap holding a (possibly singleton) list of the individual atoms in the product. The value of register `L` is stored in the `level` field and the `deadline` field, the `out_of_scope` flag is set to false, and the register `R` is incremented.

- `add_exp_res Ai`: Used when the implication operator is $\Rightarrow$. Behaves the same as `add_res`, except that the `level` and `deadline` fields are set to zero.
- `mid_imp Yj,Yk`: Used between the code that loads the resource, R, and the code for the subgoal, G. Stores the current values of register `R` (the top of the resource table) and `T` (the current value of the top flag) to the permanent variables `Yj` and `Yk`, respectively.
- `end_imp Yi,Yj,Yk`: Used after the code for the subgoal, G, when the implication operator is $\multimap$. If there are any resources in positions from `Yi` to `Yj`-1 that have not been consumed, fail. Otherwise, set the `out_of_scope` flags of all records from `Yi` to `Yj`-1 to true (trailing the changes), and set the register `T` to `Yk` $\vee$ `T`.
  In order to account for the use of $\top$ at the top level of the subgoal, G, the check for unconsumed resources is made as follows:
  - If `T` is false, the `level` and `deadline` of each resource should be `U` and 0 respectively. Otherwise, the resource is unused.
  - If `T` is true, the `level` and `deadline` of each resource should be either `U` and 0, or `L` and `L` respectively. Otherwise, the resource is unconsumed.
- `end_exp_imp Yi,Yj,Yk`: Used after the code for the subgoal, G, when the implication operator is $\Rightarrow$. The added resource entries need not be examined. Just set the `out_of_scope` flags of all records from `Yi` to `Yj`-1 to true (trailing the changes), and set register `T` to `Yk` $\vee$ `T`.

The code generated for the goal $(p(1) \otimes (p(2) \,\&\, p(3))) \multimap G$ is:

```
begin_imp Y1
put_str p/1, A2       % put [p(1)] to A1
set_int 1
put_list A1
set_val A2
set_con []
add_res A1            % add linear [p(1)]
put_str p/1, A2       % Begin: put [p(2), p(3)] to A1
set_int 2
put_str p/1, A4
set_int 3
put_list A3
set_val A4
set_con []
put_list A1
set_val A2
set_val A3            % End: put [p(2), p(3)] to A1
add_res A1            % add linear [p(2), p(3)]
mid_imp Y2, Y3
Code for G
end_imp Y1, Y2, Y3
```

If the goal used $\Rightarrow$ rather than $\multimap$, then the code would have uses of `add_res` and `end_imp` replaced by `add_exp_res` and `end_exp_imp`, respectively.

6.3.3 Code for $G_1 \,\&\, G_2$

The code for a goal of the form $G_1 \& G_2$ is, structurally, quite simple, however the semantics of each instruction is relatively complex, in order to match with the behavior defined in the corresponding $\mathcal{LRM}$ rules.

```
begin_with Yi
Code for G1
mid_with Yj
Code for G2
end_with Yi,Yj
```

- `begin_with Yi`: Decrement `U` so that we can tell which resources are consumed in G_1 Store the current value of `T` in a new variable `Yi` and set `T` to false.
- `mid_with Yj`: Perform $\underset{(\mathtt{U},0)\rightarrow(\mathtt{L}+1,\mathtt{L}+1)}{\mathtt{changepair}}$. This marks all of the resources that were used in the left conjunct so that they can, and must, be used in the right conjunct. If the value of `T` is true, then perform $\underset{\mathtt{L}\rightarrow\mathtt{L}+1}{\mathtt{change}}$ (making resources that were available but not used, but which $\top$ could have used, available in the second conjunct). Increment `L` and `U`. Store the current value of `T` in a new variable `Yj`. Set `T` to false.
- `end_with Yi, Yj`: Decrement `L`. If the value of `T` is true, then perform $\underset{(\mathtt{L}+1,\mathtt{L}+1)\rightarrow(\mathtt{U},0)}{\mathtt{changepair}}$ ($\top$ was seen in this conjunct, so we can set all the resources that should have been consumed, but weren't, as though they were). Otherwise, perform $\mathtt{consumed}_{\mathtt{L}+1}$ to check whether all the resources that should have been consumed, were. If this fails, fail. Otherwise, If `Yj` is true, then perform $\underset{\mathtt{L}+1\rightarrow\mathtt{L}}{\mathtt{change}}$ (Those resources that were made available to the second conjunct because the first conjunct included a $\top$, but weren't used in the second conjunct either, are put back to their original level). Set `T` to $\mathtt{Yi} \vee (\mathtt{Yj} \wedge \mathtt{T})$.

The code generated for the goal $p \,\&\, (q \otimes (r \,\&\, s))$ is:

```
begin_with Y1
call p/0
mid_with Y2
call q/0
begin_with Y3
call r/0
mid_with Y4
call s/0
```

```
end_with Y3, Y4
end_with Y1, Y2
```

6.3.4 Code for $!G$

In the execution of $!G$, only exponential resources can be used. The following instructions are used for $!G$.

- `begin_bang Yi`: Increment `L`. (It is now higher than the `level` field of all resources, so they are all unavailable.) Store the value of `T` in a new variable `Yi`.
- `end_bang Yi`: Decrement `L`. Reset the value of the register `T` from the variable `Yi`.

The code generated for the goal $!(p \otimes q)$ is:

```
begin_bang Y1
call p/0
call q/0
end_bang Y1
```

6.3.5 Code for $\top$

The use of $\top$ as a goal (which is written `erase`) is compiled to the instruction:

- `top`: Set the register `T` to true.

6.4 Code for Atomic Goals

The execution of a `call` to an atomic goal A proceeds as follows:

1. Extract a list of pointers to possibly unifiable resources in the resource table, `RES`, by reference to `HASH` and `SYMBOL`.
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) For each term in the list pointed to by the `head` field R, Attempt to unify A with the term. If successful, proceed. If not, backtrack.
   (c) Mark the entry R as consumed (set `level` to the current value of `U`, and `dealine` to 0).
3. After the failure of all trials, `call` the ordinary code for predicate A.

6.5 Backtracking

In order to be able to recover the correct state on backtracking:

- The values of registers `R`, `L`, `U`, and `T` are stored in choice-point frames.
- Changes to the `HASH` table should be trailed.
- Moving entries in `RES` out of scope, and changing their `level` or `deadline` should be trailed.

N	# Runs Avged.	SICStus	LLPAM 0.42	% Δ	LLPAM Prolog
8	10	158 ms	140 ms	11%	327 ms
9	10	740 ms	608 ms	18%	1527 ms
10	5	3638 ms	2663 ms	27%	7443 ms
11	5	21264 ms	12939 ms	39%	41927 ms
12	1	109520 ms	68450 ms	38%	224283 ms
13	1	675990 ms	383466 ms	43%	1353983 ms
14	1	4220390 ms	2478850 ms	41%	8668933 ms
15	1	ms	ms	%	

Figure 3: Comparison of results for LLPAM 0.42 vs SICStus 2.1

7 Performance and Conclusion

For several years the advocates of logic programming languages based on richer logics than Horn clauses have argued that the naturalness of programming in these more expressive logics would balance any loss of efficiency due to the added expense of proof search. Some took a bolder stance, however: that with appropriate implementation techniques, these logics could yield solutions to some problems whose performance would be better than the traditional solutions. The LLPAM is the first system to fulfill that promise.

The standard solution of the N-Queens problem as given in Figure 1 makes natural use of resources. Because these resources are managed in a list structure, there is little the compiler can do to optimize resource management. The rendition of this problem in Lolli as given in Figure 1 makes similar conceptual use of resources. However, because the resources are managed at the clause level, rather than at the term level, the compiler may bring various techniques to bare, in particular hashed access to the resource list.

The result of this shift is that the Lolli solution, executed under LLP 0.42[1] has superior performance compared to the traditional solution compiled under SICStus 2.1. Figure 3 compares the performance of the two programs finding all solutions for various values of N. All times were collected on a Sparcstation 20 running SunOS 4.1.4. As the reliance on resource management increases due to the increased number of queens being managed, the performance of LLP versus Prolog becomes more pronounced. Preliminary tests of other (admittedly simple) benchmarks such as knight's tour have confirmed these results.

In order to show the true nature of the speedup, the final column shows the time to execute the Prolog version of the program on the LLPAM emulator. While this result seems to bely our claim that the new structures

[1]LLP 0.42 is actually based on the earlier version of the LLPAM which handled a smaller fragment of Lolli than the LLPAM described here. (That version of the LLPAM is described in [11], and is available from `http://bach.seg.kobe-u.ac.jp/llp/`.) We do not, however, expect any slowdown from implementing the version described here.

incur only a small overhead, the disparity is more likely due to the difference between a modern optimizing Prolog compiler generating native code, and our naive compiler generating code then run on the LLPAM emulator.

References

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

[2] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In editors, *Proceedings of the Fifth Workshop on Extensions of Logic Programming*, 1996.

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

[4] James Harland and David Pym. Resource distribution via boolean constraints. In editor, *Proceedings of the Fourteenth International Conference on Automated Deduction*, 1997.

[5] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In *Algebraic Methodology and Software Technology*, 1996.

[6] James Harland and Michael Winikoff. Implementing the linear logic programming language Lygon. In *Proceedings of the 1995 International Logic Programming Symposium*, 1995.

[7] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[8] Joshua S. Hodas and Dale 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, 1991.

[9] Joshua S. Hodas, Kevin Watkins, Naoyuki Tamura, and Kyoung-Sun Kang. Efficient implementation of a linear logic programming language. An extended, journal version of this paper. In preparation.

[10] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge MA, 1986.

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

[12] David H. D. Warren. An abstract Prolog instruction set. Technical Report Technical Report 309, SRI International, October 1983.