Design and implementation of linear logic programming languages

番原, 睦則

(Degree)
博士（工学）

(Date of Degree)
2002-09-20

(Date of Publication)
2007-08-09

(Resource Type)
doctoral thesis

(Report Number)
乙2637

(URL)
https://hdl.handle.net/20.500.14094/D2002637

Doctoral Dissertation

Design and Implementation of Linear Logic Programming Languages

Mutsunori Banbara

September 2002

The Graduate School of Science and Technology
Kobe University, Japan

Supervisor: Prof. Yuzuru Kakuda
Co-supervisor: Prof. Yukio Kaneda
Co-supervisor: Prof. Toshiyasu Arai
Co-supervisor: Assoc. Prof. Naoyuki Tamura
Department of Computer and Systems Engineering
Faculty of Engineering
Kobe University
1-1 Rokkodai, Nada, Kobe, Hyogo 657-8501 Japan

to my wife, Megumi

Acknowledgments

I would like to thank Prof. Naoyuki Tamura for his advice, help, patience, and understanding. I would also like to thank Prof. Yuzuru Kakuda who have encourage me to make a great effort since I was an undergraduate student at Kobe University. I would like to thank Prof. Yukio Kaneda and Prof. Toshiyasu Arai for their advice.

I would like to thank Prof. Kenichi Aragane, Masaru Teranishi, and Naoshi Kanazawa at Nara National College of Technology for their kindness.

I would like to thank Eiji Sugiyama and Kyoung-Sun Kang who had spent pleasant days together at the Graduate School of Science and Technology of Kobe University.

I would like to thank Kenichiro Shii who attracted me by his remarkable programming sense.

I would like to thank Makoto Kikuchi for his great deal of advice, Christopher Barney for his English support, and CS32 laboratory.

I thank my parents Yoshiyuki and Yuko, my younger sister Yoshiko, and my wife Megumi. I could not finish this dissertation without their emotional support. In particular, I thank my grandfather Masayoshi in heaven, who said me you must give up trying to get a Ph.D.

Finally, I would also like to thank my friends for sharing interesting times.

Mutsunori Banbara
Kobe, Japan
September, 2002

Abstract

Linear logic was introduced by J.-Y.Girard in 1987 as a resource-conscious refinement of classical logic. Linear logic has found many applications in computer science because it allows elegant solutions to many problems that are difficult to represent in traditional logics. The expressive power of linear logic is shown by some natural encoding of computational models, such as Petri nets, counter machines, and π-calculus. One key topic of linear logic was the development of new programming languages. A number of logic programming languages based on linear logic have been proposed: LO, LinLog, Lolli, ACL, Lygon, Forum, and Linear LF. These languages suggest a direction to extend logic programming to be more expressive and more efficient.

In spite of a great deal of theoretical works, there have been very few practical tools for developing resource-conscious applications based on linear logic. It is therefore very important to develop efficient compiler system for linear logic programming languages. However, the implementation of such languages meets many difficulties not arise in traditional logic programming languages. In particular efficient *resource management* is a serious problem for the implementors.

In this dissertation, we propose new compilation methods to develop efficient implementation for linear logic programming languages. Main contributions are summarized as follows:

1. A compiler system for a linear logic programming language:
   Compiling resources is an important issue on implementation for linear logic programming languages. We present a method for compiling resources and provide an extension of the WAM for a linear logic programming language. In performance, our compiler provides 40% speedup for a theorem proving application of classical logic, relative to its Prolog implementation.

2. A translator system from a linear logic programming language into Java:
   There has been no 100% pure Java implementation for linear logic programming languages so far. We present a method for translating a linear logic programming language into Java. In performance, our translator is 1.7 times faster for a set of classical Prolog benchmarks, than an existing Prolog-to-Java translator jProlog.

3. A compiler system for a temporal linear logic programming language:
   We present theory, language design, an abstract machine and its instruction set for compiler system of a temporal linear logic programming language. In addition to resource sensitive features, it is possible to express time-dependent properties of resources, in particular, the precise order of the moments when some resources are consumed.

Our compiler has already applied to a theorem proving application of first-order classical logic, in which linear logic operators were elegantly used for specifying the problems. Furthermore it gives significantly nice performance relative to its famous Prolog implementation. Our results should be equally well applied to other resource-conscious applications based on linear logic.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this dissertation, we propose new compilation methods to develop efficient implementation for linear logic programming languages. Main contributions are summarized as follows:

1. A compiler system for a linear logic programming language:
   We present an abstract machine and its instruction set for compiler system of a linear logic programming language.

2. A translator system from a linear logic programming language into Java:
   We present a method for translating a linear logic programming language into Java.

3. A compiler system for a temporal linear logic programming language:
   We present theory, language design, an abstract machine and its instruction set for compiler system of a temporal linear logic programming language.

Linear logic [19] was introduced by J.-Y.Girard in 1987 as a resource-conscious refinement of classical logic. Linear logic has found many applications in computer science because it allows elegant solutions to many problems that are difficult to represent in traditional logics. The expressive power of linear logic is shown by some natural encoding of computational models, such as Petri nets, counter machines, and π-calculus. In recent years, linear logic have been applied to operational semantics of security protocols [12] and specifying read-time finite-state systems [37].

One key topic of linear logic was the development of new programming languages. A number of logic programming languages based on linear logic have been proposed: LO [3], LinLog [2], Lolli [25, 26], ACL [32, 33], Lygon [20, 21], Forum [27, 41], and Linear LF [14]. These languages suggest a direction to extend logic programming to be more expressive and more efficient. For example, Lolli was used to develop a filler-gap parser [24] in natural language processing. Forum was used to specify the operational semantics of a pipelined RISC processor [15]. More about linear logic programming has been well-summarized in Miller's papers [39, 40].

Most early works on logic programming had been based on Horn clauses (a simple logic underlying pure Prolog) and SLD-resolution (Prolog's execution model). However, it is very hard to extend this traditional approach for new logic programming languages based on richer logics, rather than Horn clauses. Many researcher recently have used proof-theoretical approaches, *goal-directed proof search* in Gentzen-style sequent calculus. One design principle, called *uniform proofs* [38] proposed by Miller *et al.*, is a simple and powerful notion for designing logic programming languages. A logical system is an *abstract logic programming language* if restricting it to uniform proofs retains completeness. The logics of pure Prolog, λProlog [44], Lolli, and Forum are examples of abstract logic programming language.

In spite of a great deal of theoretical works, there have been very few practical tools for developing linear-logic-based applications. It is therefore very important to develop efficient compiler systems for linear logic

programming languages. However, the implementation of such languages meets many difficulties not arise in traditional logic programming languages. In linear logic programming languages, it is possible to add and delete resources (limited-use clauses) dynamically as logical formulas. The efficient *resource management* is therefore an important issue for the implementors. This issue has been discussed from the earliest proposals [26], and several papers have focused on these issues [13] [30, 49] [22] [29].

Recently, only one compiling effort has been made. N. Tamura and Y. Kaneda [49] have developed an extension of the Warren Abstract Machine (WAM) [1, 56] and a compiler system for a useful fragment of Lolli. J. Hodas *et al.* proposed a refinement [29] with the complete treatment of $\top$. However, the compiler supported only limited forms of resources. Furthermore, the resources were stored as terms in a heap memory and were not compiled. This inefficiency is clearly unacceptable for resource-conscious applications based on linear logic.

This dissertation is the latest step in a course of research begun by N. Tamura and Y. Kaneda towards efficient implementation for linear logic programming languages. We discuss several theoretical and practical issues on implementation in the case of Lolli proposed by J. Hodas and D. Miller [26]. After that we present the following systems:

1. A compiler system for a linear logic programming language:
   Compiling resources is an important issue on implementation for linear logic programming languages. To solve this problem, we introduce the idea of *closure* and present a method for compiling resources. We also present an extension of the WAM for a linear logic programming language LLP. In performance, our compiler provides 40% speedup for a theorem proving application of classical logic, relative to its Prolog implementation.

   This part of dissertation is based on joint work with N. Tamura *et al.* in the paper [6, 9, 47].

2. A translator system from a linear logic programming language into Java:
   A number of Java implementations for logic programming languages have been developed. However, there has been no 100% pure Java implementation for linear logic programming languages. We present a LLP-to-Java source-to-source translator system. Our translation method is based on continuation passing style compilation [54]. In performance, our translator is 1.7 times faster for a set of classical Prolog benchmarks, than an existing Prolog-to-Java translator jProlog.

   This part of dissertation is based on joint work with N. Tamura *et al.* in the paper [5, 8, 10].

3. A compiler system for a temporal linear logic programming language:
   We present theory and design of a logic programming language based on intuitionistic temporal linear logic, called TLLP. In addition to resource sensitive features of LLP, TLLP can express time-dependent properties of resources, in particular, the precise order of the moments when some resources are consumed. We also present an abstract machine and its instruction set for TLLP compiler system, and a method for translating TLLP into LLP. In performance, our compiler is 1.7 times faster for a simple example of Timed Petri Net, than translating TLLP into LLP.

   This part of dissertation is based on joint work with N. Tamura *et al.* in the paper [4, 7]

Our compiler has already applied to a theorem proving application [28] of first-order classical logic, in which linear logic operators were elegantly used for specifying the problems. Furthermore it gives significantly nice performance relative to its famous Prolog implementation. Our results should be equally well applied to other resource-conscious applications based on linear logic.

Our translator has already been applied to a Prolog-to-C# translator [17] and a tool that generates wrappers for command line programs [58].

Finally, we give the outline of this dissertation. Chapter 2 gives a brief introduction to linear logic. After that we present language design of a linear logic programming language Lolli proposed by J. Hodas and D. Miller. We also discusses an issue on resource management in proof search of Lolli, a serious problem for the implementors.

Chapter 3 shows several example programs in Lolli so that the reader easily understand a sense of *resource programming*.

Chapter 4 discusses some basic issues on implementation of the Lolli language. We present a method for compiling resources into *closure*, a reference of compiled code and a set of bindings for free variables, widely used in implementations of functional programming languages.

Chapter 5 presents the detail of LLP abstract machine: registers, data areas, instructions, code generation, and optimization. Appendix A summarizes the instruction set including auxiliary procedures and functions.

Chapter 6 discusses three approaches for translating Prolog into Java. After that we present a method for translating LLP into Java.

Chapter 7 presents theory, language design, an abstract machine and its instruction set for compiler system of a temporal linear logic programming language.

Chapter 8 concludes this dissertation and presents our future works.

Chapter 2

The Lolli Language and its Resource Management Models

Most early works on logic programming had been based on Horn clauses, a simple logic underlying pure Prolog. In recent years, many researcher have used proof-theoretical approaches to design and implement new logic programming languages based on richer logics, rather than Horn clauses.

One design principle, called *uniform proofs* [38] proposed by Miller *et al*., is a simple and powerful notion for designing logic programming languages. A logical system is an *abstract logic programming language* if restricting it to uniform proofs retains completeness. The logics of pure Prolog, λProlog, and Lolli are examples of abstract logic programming language.

In this chapter, we will give a brief introduction to Linear Logic [19] and present the logic of a linear logic programming language Lolli [25, 26]. We will also discuss the issue of *resource management*, one of the most serious problem for the implementor.

2.1 A Brief Introduction to Linear Logic

Linear logic was introduced by J.-Y.Girard in 1987 [19] as a resource-conscious refinement of classical logic. Linear logic has found many applications in computer science because it allows elegant solutions to many problems that are difficult to represent in traditional logics. The expressive power of linear logic is shown by some natural encoding of computational models, such as Petri nets, counter machines, π-calculus, and others.

In traditional logic, the *structural rules* of weakening and contraction allow formulas to be discarded and duplicated respectively. In linear logic, these rules are removed, but the modalities are added instead. Thus we have not lost anything: both classical and intuitionistic logic can be faithfully embedded into linear logic. In the absence of weakening and contraction, many of the logical operators split into two variants.

- There are two conjunctions, "$\otimes$" (*tensor*) and "$\&$" (*with*), two disjunctions, "$\wp$" (*par*) and "$\oplus$" (*o-plus*). The operators $\otimes$ and $\wp$ are dual to each other, also $\&$ and $\oplus$ are dual.
- There are two truth, "$\top$" (*top*) and "1" (*one*), two falsehoods "0" (*zero*) and "$\bot$" (*bottom*). Each of these constants is a unit: $\top$ is the unit of $\&$, 1 is the unit of $\otimes$, $\bot$ is the unit of $\wp$, and 0 is the unit of $\oplus$. The constants $\top$ and 0 are dual to each other, also 1 and $\bot$ are dual.
- The implication operator is written "$\multimap$" and called *linear implication*.
- The negation of A is written "$A^{\bot}$" and called *linear negation*.
- There are two modalities, "!" (*of course*) and "?" (*why not*), that are dual to each other. These modalities are very similar to the modal operators $\Box$ and $\Diamond$ in the usual modal logics. The role of these

$$\frac{}{B \longrightarrow B}\ (\text{Identity}) \qquad \frac{\Delta_1 \longrightarrow B \quad \Delta_2, B \longrightarrow C}{\Delta_1, \Delta_2 \longrightarrow C}\ (\text{Cut})$$

$$\frac{}{\Delta, 0 \longrightarrow C}\ (\text{L0}) \qquad \frac{}{\Delta \longrightarrow \top}\ (\text{R}\top)$$

$$\frac{\Delta \longrightarrow C}{\Delta, 1 \longrightarrow C}\ (\text{L1}) \qquad \frac{}{\longrightarrow 1}\ (\text{R1})$$

$$\frac{\Delta, B_i \longrightarrow C}{\Delta, B_1 \,\&\, B_2 \longrightarrow C}\ (\text{L}\,\&_i) \qquad \frac{\Delta \longrightarrow C_1 \quad \Delta \longrightarrow C_2}{\Delta \longrightarrow C_1 \,\&\, C_2}\ (\text{R}\,\&)$$

$$\frac{\Delta, B_1, B_2 \longrightarrow C}{\Delta, B_1 \otimes B_2 \longrightarrow C}\ (\text{L}\otimes) \qquad \frac{\Delta_1 \longrightarrow C_1 \quad \Delta_2 \longrightarrow C_2}{\Delta_1, \Delta_2 \longrightarrow C_1 \otimes C_2}\ (\text{R}\otimes)$$

$$\frac{\Delta, B_1 \longrightarrow C \quad \Delta, B_2 \longrightarrow C}{\Delta, B_1 \oplus B_2 \longrightarrow C}\ (\text{L}\oplus) \qquad \frac{\Delta \longrightarrow C_i}{\Delta \longrightarrow C_1 \oplus C_2}\ (\text{R}\oplus_i)$$

$$\frac{\Delta_1 \longrightarrow C_1 \quad \Delta_2, B \longrightarrow C_2}{\Delta_1, \Delta_2, C_1 \multimap B \longrightarrow C_2}\ (\text{L}\multimap) \qquad \frac{\Delta, B \longrightarrow C}{\Delta \longrightarrow B \multimap C}\ (\text{R}\multimap)$$

$$\frac{\Delta, B \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ (\text{L}\,!) \qquad \frac{!\,\Delta \longrightarrow C}{!\,\Delta \longrightarrow !\,C}\ (\text{R}\,!)$$

$$\frac{\Delta \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ (\text{W}\,!) \qquad \frac{\Delta, !\,B, !\,B \longrightarrow C}{\Delta, !\,B \longrightarrow C}\ (\text{C}\,!)$$

$$\frac{\Delta, B[t/x] \longrightarrow C}{\Delta, \forall x.B \longrightarrow C}\ (\text{L}\forall) \qquad \frac{\Delta \longrightarrow C[t/x]}{\Delta \longrightarrow \exists x.C}\ (\text{R}\exists)$$

$$\frac{\Delta, B[y/x] \longrightarrow C}{\Delta, \exists x.B \longrightarrow C}\ (\text{L}\exists) \qquad \frac{\Delta \longrightarrow C[y/x]}{\Delta \longrightarrow \forall x.C}\ (\text{R}\forall)$$

provided, in each case, y does not appear free in the conclusion.

Figure 2.1: The Proof System *ILL* for Intuitionistic Linear Logic

modalities is to reintroduce weakening and contraction.

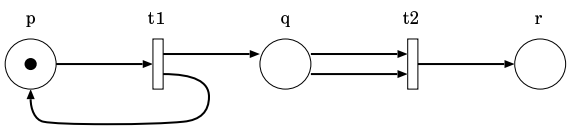
- The first-order quantifiers "$\forall$" and "$\exists$" are the same as those in traditional logic.

On the other hand, these operators can be classified into four groups: *multiplicative* operators ($\otimes$, $\wp$, $\multimap$, 1, $\bot$), *additive* operators ($\&$, $\oplus$, $\top$, 0), *exponentials* (!, ?), and quantifiers ($\forall$, $\exists$). It is noted that there are two major notations for linear logic operators: Girard's notation [19] that we use and Troelstra's notation [55].

As traditional logical systems, linear logic has been studied both from classical (multiple conclusion) and intuitionistic (single conclusion) points of view. We will focus on an intuitionistic variant in Figure 2.1. The *ILL* system does not include the operators $\wp$, ?, and $\bot$. This is because we are interested only in cut-free proofs of Gentzen-style sequent calculus in intuitionistic linear logic. The sequent of *ILL* is an expression of the form $\Delta \longrightarrow C$ where Δ is a multiset of formulas.

Linear logic treats logical assumptions as consumable resources. In *ILL*, the weakening and contraction rules are available only for assumptions marked with the modality !. This means that, in general, a assumption not thus marked can only be used once in a branch of the search for a proof. Limited-use formulas can represent limited resources in some domain.

To illustrate this *formulas-as-resources* notion, we show an example for encoding Petri net reachability in Figure 2.2. In linear logic the formula $!(p \multimap (p \otimes q))$ can be used to encode the transition $t1$ taking one token from place p and adding tokens to place p and q. Similarly, the formula $!((q \otimes q) \multimap r)$ represents the transition $t2$ taking two tokens from q and adding one token to place r. Petri net transitions can be encoded as (reusable) !-marked linear implication. Tokens are represented as (limited-use) atomic formulas.



$places$: p, q, r (atomic formulas)
$t1$: $!\,(p \multimap (p \otimes q))$
$t2$: $!\,((q \otimes q) \multimap r)$

Figure 2.2: A Linear Logic Encoding of a Petri Net

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q \longrightarrow q}\quad\dfrac{}{q \longrightarrow q}\,(\mathrm{R}\otimes)}{q,\ q \longrightarrow q \otimes q}\quad\dfrac{}{p \longrightarrow p}\quad\dfrac{}{r \longrightarrow r}\,(\mathrm{R}\otimes)}{p, r \longrightarrow p \otimes r}\,(\mathrm{L}\multimap)}{p,\ q,\ q,\ (q \otimes q) \multimap r \longrightarrow p \otimes r}\,(\mathrm{L}!)}{!((q \otimes q) \multimap r),\ p,\ q,\ q \longrightarrow p \otimes r}\,(\mathrm{L}\otimes)}{\dfrac{}{p \longrightarrow p}\quad !((q \otimes q) \multimap r),\ p \otimes q,\ q \longrightarrow p \otimes r}\,(\mathrm{L}\multimap)}{!((q \otimes q) \multimap r),\ p,\ p \multimap (p \otimes q),\ q \longrightarrow p \otimes r}\,(\mathrm{L}!)}{!(p \multimap (p \otimes q)),\ !((q \otimes q) \multimap r),\ p,\ q \longrightarrow p \otimes r}\,(\mathrm{L}\otimes)}{\dfrac{}{p \longrightarrow p}\quad !(p \multimap (p \otimes q)),\ !((q \otimes q) \multimap r),\ p \otimes q \longrightarrow p \otimes r}\,(\mathrm{L}\multimap)}{!(p \multimap (p \otimes q)),\ !((q \otimes q) \multimap r),\ p,\ p \multimap (p \otimes q) \longrightarrow p \otimes r}\,(\mathrm{L}!)}{!(p \multimap (p \otimes q)),\ !(p \multimap (p \otimes q)),\ !((q \otimes q) \multimap r),\ p \longrightarrow p \otimes r}\,(\mathrm{C}!)$$
$$!(p \multimap (p \otimes q)),\ !((q \otimes q) \multimap r),\ p \longrightarrow p \otimes r$$

Figure 2.3: Proof Search for the Reachability of a Petri Net in Figure 2.2

A reachability problem from the initial marking (one token in p) to the final marking (one token in both p and r) can be represented as a sequent: $!(p \multimap (p \otimes q)), !((q \otimes q) \multimap r), p \longrightarrow p \otimes r$. Figure 2.3 presents a *ILL* proof that corresponds to a sequence of Petri net from initial marking $\{p\}$ to the final marking $\{p, r\}$.

Linear logic has been applied to several other areas in computer science. One key application of formulas-as-resources aspect was the development of new programming languages. Recently, a number of logic programming languages based on linear logic have been proposed: LO [3], LinLog [2], Lolli [25, 26], ACL [32, 33], Lygon [20, 21], Forum [27, 41], and Linear LF [14]. These languages suggest a direction to extend logic programming to be more expressive and more efficient. The treatment of formulas-as-resources gives us not only powerful expressiveness, but also efficient access to a large set of data. More about linear logic programming has been well-summarized in Miller's papers [39, 40].

2.2 Uniform Proofs in Intuitionistic Linear Logic

The idea of *uniform proofs* [38], proposed by Miller *et al.*, is a simple and powerful notion for designing logic programming languages. Uniform proof search is a cut-free, *goal-directed proof search* in which a sequent $\Gamma \longrightarrow G$ denotes the state of the computation trying to solve the goal G from the program Γ. Uniform proof is characterized operationally by the bottom-up construction of proofs in which right-introduction rules are applied first and left-introduction rules are applied only when the right-hand side is atomic. This means that the operators in the goal G are executed independently from the program Γ, and the program is only considered when its goal is atomic. A logical system is an *abstract logic programming language* if restricting it to uniform proofs retains completeness. The logics of pure Prolog, λProlog [44], and Lolli are examples of abstract logic programming language.

Clearly, intuitionistic linear logic (even over the connectives: $\top$, $\&$, $\otimes$, $\multimap$, $!$, and $\forall$) is not an abstract logic programming language. For example, the sequents $a \otimes b \longrightarrow b \otimes a$ and $!\,a \,\&\, b \longrightarrow !\,a$ are both provable in *ILL* (see Figure 2.1) but do not have uniform proofs.

$$\frac{}{\Gamma; B \longrightarrow B}\ (\text{Identity}) \qquad \frac{\Gamma, B; \Delta, B \longrightarrow C}{\Gamma, B; \Delta \longrightarrow C}\ (\text{absorb})$$

$$\frac{}{\Gamma; \varnothing \longrightarrow 1}\ (\text{R1}) \qquad \frac{}{\Gamma; \Delta \longrightarrow \top}\ (\text{R}\top)$$

$$\frac{\Gamma; \Delta, B_i \longrightarrow C}{\Gamma; \Delta, B_1 \,\&\, B_2 \longrightarrow C}\ (\text{L}\,\&_i) \qquad \frac{\Gamma; \Delta \longrightarrow C_1 \quad \Gamma; \Delta \longrightarrow C_2}{\Gamma; \Delta \longrightarrow C_1 \,\&\, C_2}\ (\text{R}\,\&)$$

$$\frac{\Gamma; \Delta_1 \longrightarrow C_1 \quad \Gamma; \Delta_2, B \longrightarrow C_2}{\Gamma; \Delta_1, \Delta_2, C_1 \multimap B \longrightarrow C_2}\ (\text{L}\multimap) \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta \longrightarrow B \multimap C}\ (\text{R}\multimap)$$

$$\frac{\Gamma; \varnothing \longrightarrow C_1 \quad \Gamma; \Delta, B \longrightarrow C_2}{\Gamma; \Delta, C_1 \multimap B \longrightarrow C_2}\ (\text{L}\Rightarrow) \qquad \frac{\Gamma, B; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow B \Rightarrow C}\ (\text{R}\Rightarrow)$$

$$\frac{\Gamma; \Delta, B[t/x] \longrightarrow C}{\Gamma; \Delta, \forall x.B \longrightarrow C}\ (\text{L}\forall) \qquad \frac{\Gamma; \Delta \longrightarrow C[y/x]}{\Gamma; \Delta \longrightarrow \forall x.C}\ (\text{R}\forall)$$

provided that y is not free in the conclusion.

$$\frac{\Gamma; \varnothing \longrightarrow C}{\Gamma; \varnothing \longrightarrow\, !\, C}\ (\text{R}\,!) \qquad \frac{\Gamma; \Delta_1 \longrightarrow C_1 \quad \Gamma; \Delta_2 \longrightarrow C_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C_1 \otimes C_2}\ (\text{R}\otimes)$$

$$\frac{\Gamma; \Delta \longrightarrow C[t/x]}{\Gamma; \Delta \longrightarrow \exists x.C}\ (\text{R}\exists) \qquad \frac{\Gamma; \Delta \longrightarrow C_i}{\Gamma; \Delta \longrightarrow C_1 \oplus C_2}\ (\text{R}\oplus_i)$$

Figure 2.4: The Proof System $\mathcal{L}$ for the Lolli Language

$$\frac{\Gamma; \varnothing \longrightarrow G_1 \quad \cdots \quad \Gamma; \varnothing \longrightarrow G_n \quad \Gamma; \Delta_1 \longrightarrow G'_1 \quad \cdots \quad \Gamma; \Delta_m \longrightarrow G'_m}{\Gamma; \Delta_1, \ldots, \Delta_m, R \longrightarrow A}\ (\text{BC})$$

provided $n, m \geq 0$, A is atomic, and $\langle \{G_1, \ldots, G_n\}, \{G'_1, \ldots, G'_m\}, A \rangle \in \|R\|$

Figure 2.5: Backchaining for the Proof System $\mathcal{L}'$

Hodas and Miller have designed the linear logic programming language Lolli [25][26] by restricting formulas so that the above counterexamples do not appear, although it retains desirable features of linear logic connectives such as ! and $\otimes$. The Lolli language is based on the fragment of linear logic freely generated by the connectives: $\top$, $\&$, $\Rightarrow$, $\multimap$, and $\forall$. The connective $\Rightarrow$ is called *intuitionistic implication* and is defined as $B \Rightarrow C \equiv (!\, B) \multimap C$. Lolli also allows the use of positive occurrences of 1, $\otimes$, !, $\oplus$, and $\exists$, since they does not cause any problems with the completeness of uniform provability. The Lolli language is formally defined as follows:

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

Here, R-formulas are called *resource formulas*, and G-formulas are called *goal formulas*.

The sequent of Lolli is an expression of the form $\Gamma; \Delta \longrightarrow G$ where Γ and Δ are multisets of resource formulas, and G is a goal formula. Γ and Δ are called *intuitionistic context* and *linear context* respectively, and they correspond to the *program*. G is called the *goal*.

Hodas and Miller developed a series of proof systems $\mathcal{L}$ and $\mathcal{L}'$ in [25]. The entire set of $\mathcal{L}$ sequent rules is given in Figure 2.4. The sequent $\Gamma; \Delta \longrightarrow G$ can be mapped to the linear logic sequent $!\,\Gamma, \Delta \longrightarrow G$. Thus, the right introduction rule for $\multimap$ adds its assumption (called a *linear resource*) to the linear context, in which every formula can be used exactly once. The right introduction rule for $\Rightarrow$ adds its assumption

(called an *intuitionistic resource*) to the intuitionistic context, in which every formula can be used arbitrarily many times (including zero times). They proved that $\mathcal{L}$ is sound and complete with respect to the *ILL* rules restricted to the Lolli language. They also proved $\mathcal{L}$ preserves completeness even if probability is restricted to uniform proofs.

Proposition 2.2.1 (Hodas and Miller) Let G be a goal formula, Γ and Δ multisets of resource formulas. Let D^* be the result of replacing all occurrences of $B \Rightarrow C$ in D with $(!\,B) \multimap C$, and let $\Gamma^* = \{B^* \mid B \in \Gamma\}$. Then the sequent $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{L}$ if and only if $!(\Gamma^*), \Delta^* \longrightarrow G^*$ is provable in *ILL*.

Proposition 2.2.2 (Hodas and Miller) Let G be a goal formula, Γ and Δ multisets of resource formulas. Then the sequent $\Gamma; \Delta \longrightarrow G$ has a proof in $\mathcal{L}$ if and only if it has a uniform proof in $\mathcal{L}$.

Hodas and Miller have simplified $\mathcal{L}$ by the fact that in uniform proofs left-hand rules and Identity are used only when the right-hand side is atomic. Let R be a resource formula. $\|R\|$ is defined as the smallest set of triples of the form $\langle \Gamma, \Delta, R' \rangle$ where Γ and Δ are multisets of goal formulas, such that:

1. $\langle \varnothing, \varnothing, R \rangle \in \|R\|$,
2. if $\langle \Gamma, \Delta, R_1 \,\&\, R_2 \rangle \in \|R\|$ then both $\langle \Gamma, \Delta, R_1 \rangle \in \|R\|$ and $\langle \Gamma, \Delta, R_2 \rangle \in \|R\|$,
3. if $\langle \Gamma, \Delta, \forall x.R' \rangle \in \|R\|$ then for all closed terms t, $\langle \Gamma, \Delta, R'[t/x] \rangle \in \|R\|$,
4. if $\langle \Gamma, \Delta, G \Rightarrow R' \rangle \in \|R\|$ then $\langle \Gamma \uplus \{G\}, \Delta, R' \rangle \in \|R\|$ ("$\uplus$" is multiset union), and
5. if $\langle \Gamma, \Delta, G \multimap R' \rangle \in \|R\|$ then $\langle \Gamma, \Delta \uplus \{G\}, R' \rangle \in \|R\|$.

$\mathcal{L}'$ is the proof system that results from replacing the Identity, L$\multimap$, L$\Rightarrow$, L$\&$, and L$\forall$ rules in $\mathcal{L}$ with a single rule, called *backchaining* (BC) in Figure 2.5. Roughly speaking, the $\|\cdot\|$ function translates a resource formula in the program into a set of program clauses.

Proposition 2.2.3 (Hodas and Miller) Let G be a goal formula, and let Γ and Δ be multisets of resource formulas. The sequent $\Gamma; \Delta \longrightarrow G$ has a proof in $\mathcal{L}$ if and only if it has a proof in $\mathcal{L}'$.

Lolli can be seen as an extension of pure Prolog and λProlog. Lolli allows resource formulas in the program to be used either only once or arbitrarily many times. In Lolli program, resource formulas correspond to program clauses, but their structure seems to be rather complicated than Prolog and λProlog. Frequently, it is very convenient for the implementors to view a resource formula $\forall x.(G_1 \Rightarrow (G_2 \multimap A))$ as a resource formula $\forall x.((!\,G_1 \otimes G_2) \multimap A)$ whose head is an atomic formula such as Prolog clause.

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

By rewriting formulas using above logical equivalences in the forward direction, it is possible to simplify the definition of resource formulas:

$$
\begin{array}{rcl}
R & ::= & S_1 \,\&\, \cdots \,\&\, S_m \\
S & ::= & \top \mid A \mid G \multimap A \mid \forall x.S
\end{array}
$$

where A stands for a atomic formula and $m \geq 1$. S-formulas are called *resource clauses* in which the form $\forall x.(G \multimap A)$ corresponds to A `:-` G in Prolog. This simplification does not change expressiveness of the Lolli language. From a theoretical point of view, it makes the presentation of backchaining (especially the definition of $\|\cdot\|$) simpler. From a practical point of view, it makes the development of compiler systems (especially the compilation of resource formulas) easier.

2.3 Resource Management Models

The issue of *resource management* has been discussed from the earliest proposals [26], and recently several papers have focused on these issues [13] [30, 49] [22] [29].

In this section, we discuss some of the basic issues in resource management in the case of propositional fragment of Lolli. It is noted that we use rather restrictive form of resource formulas mentioned in previous section to make the presentation simpler.

2.3.1 The $\mathcal{I}/\mathcal{O}$ Implementation Model

The resource management during a proof search in $\mathcal{L}'$ is a serious problem for the implementor. Let us consider, for example, the rule for proving the goal $G_1 \otimes G_2$:

$$\frac{\Gamma; \Delta_1 \longrightarrow G_1 \quad \Gamma; \Delta_2 \longrightarrow G_2}{\Gamma; \underbrace{\Delta_1, \Delta_2}_{\Delta} \longrightarrow G_1 \otimes G_2} \ \mathrm{R}\otimes$$

When the system applies this rule during bottom-up search of a proof, the linear context Δ has not yet been divided into Δ_1 and Δ_2. If the generate-and-test algorithm were used to find an appropriate partition of Δ, this non-determinism is clearly unacceptable since all 2^n possibilities might need to be tested if Δ contains n resource formulas.

Hodas and Miller solved this problem by splitting resources lazily, and they proposed a new resource management model called the $\mathcal{I}/\mathcal{O}$ model [25, 26].

In this model, the IO-context is a lists of formulas, each of which is either a resource formula (linear resource), or a !-marked resource formulas (intuitionistic resource), or new constant 1 [1] that denotes a place where a formula has been consumed. The IO-sequent is an expression of the form:

$$I \,\{G\}\, O$$

where I and O are IO-contexts, and G is a goal formula. The intuitive meaning of $I \,\{G\}\, O$ is that the goal G can be proved given *input context* I so that the *output context* O remains.

Figure 2.6 presents the set of sequent rules of the $\mathcal{I}/\mathcal{O}$ model. For example, the rule for the operator $\otimes$ is as follows:

$$\frac{I \,\{G_1\}\, M \quad M \,\{G_2\}\, O}{I \,\{G_1 \otimes G_2\}\, O} \ (\otimes)$$

First, the system tries $I \,\{G_1\}\, M$ proving the goal G_1 given input context I. If this succeeds, the output context M is forwarded to the goal G_2, and then $M \,\{G_2\}\, O$ is attempted. If this second attempt fails, the system retries $I \,\{G_1\}\, M$ looking for some different pattern of consumption, before retrying $M \,\{G_2\}\, O$.

The BC_i rules handles the selection of resource clauses for backchaining. The *pickR*(I, O, R) relation selects an available clause R from its input context I matching the atom A. The output context O is the same as I, except that the occurrence of R is replaced with 1 if it is a linear resource (O is exactly the same as I if it is a intuitionistic resource). The $\top$ rule consumes any formulas from its input context. The *subcontext*(O, I) relation holds if O arises from replacing arbitrarily many (including zero) occurrences of linear resources in I with 1.

Hodas and Miller proved that the $\mathcal{I}/\mathcal{O}$ model is logically equivalent to $\mathcal{L}'$. Let I and O be IO-contexts. Only when *subcontext*(O, I) holds, the difference $I - O$ is defined as a pair $\langle \Gamma, \Delta \rangle$, where Γ is the multiset of all formulas R such that $!\,R$ is an element of the list I (and O), and Δ is the multiset of all formulas R which occur in I and the corresponding place in O is the constant 1.

[1] The new constant del is used instead in Hodas's dissertation [25].

$$\frac{}{I\,\{1\}\,I}\ (1) \qquad \frac{\mathit{subcontext}(O,I)}{I\,\{\top\}\,O}\ (\top)$$

$$\frac{I\,\{G_1\}\,M \quad M\,\{G_2\}\,O}{I\,\{G_1 \otimes G_2\}\,O}\ (\otimes) \qquad \frac{I\,\{G_1\}\,O \quad I\,\{G_2\}\,O}{I\,\{G_1 \,\&\, G_2\}\,O}\ (\&)$$

$$\frac{I\,\{G_i\}\,O}{I\,\{G_1 \oplus G_2\}\,O}\ (\oplus_i) \qquad \frac{I\,\{G\}\,I}{I\,\{!\,G\}\,I}\ (!)$$

$$\frac{[R\,|\,I]\,\{G\}\,[1\,|\,O]}{I\,\{R \multimap G\}\,O}\ (\multimap) \qquad \frac{[!\,R\,|\,I]\,\{G\}\,[!\,R\,|\,O]}{I\,\{R \Rightarrow G\}\,O}\ (\Rightarrow)$$

$$\frac{\mathit{pickR}(I,O,A)}{I\,\{A\}\,O}\ (\mathrm{BC}_1) \qquad \frac{\mathit{pickR}(I,M,G \multimap A) \quad M\,\{G\}\,O}{I\,\{A\}\,O}\ (\mathrm{BC}_2)$$

Figure 2.6: The $\mathcal{I}/\mathcal{O}$ System for Propositional Fragment of Lolli

Proposition 2.3.1 (Hodas and Miller) Let I and O be *IO*-contexts that satisfy *subcontext*(O, I). Let $I - O$ be the pair $\langle \Gamma, \Delta \rangle$ and let G be a goal formula. $I\,\{G\}\,O$ is provable in the $\mathcal{I}/\mathcal{O}$ model if and only if $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{L}'$.

2.3.2 The $\mathcal{RM}_3$ Implementation Model

The $\mathcal{I}/\mathcal{O}$ model succeeds to eliminate the most serious problem, non-determinism from the treatment of $\otimes$. However, there are some points to be improved. Let us consider the rules for proving the goal $G_1 \,\&\, G_2$ and $\top$:

$$\frac{I\,\{G_1\}\,O \quad I\,\{G_2\}\,O}{I\,\{G_1 \,\&\, G_2\}\,O}\ (\&) \qquad \frac{\mathit{subcontext}(O,I)}{I\,\{\top\}\,O}\ (\top)$$

This $\&$ rule means that the goal G_1 and G_2 must use the same set of resources. In a naive implementation, the system first copies the input context and proves the two conjuncts separately, and then it compares their output contexts. This leads to unnecessary backtracking. The $\top$ rule means that the output context O is reconstructed from the context I by replacing any linear resources with 1. If O contains n linear resources, 2^n possibilities might need to be tested.

Cervesato, Hodas, and Pfenning solved these problems and proposed a refinement of the $\mathcal{I}/\mathcal{O}$ model, called the $\mathcal{RM}_3$ model [13]. This model pays particular attention to the management of resources across $\&$ and to the occurrence of $\top$. The main difference from the $\mathcal{I}/\mathcal{O}$ model is that the idea of $\top$-*flag* is introduced, linear context is divided into two part, and intuitionistic context is separated out.

The RM_3-sequent is an expression of the form:

$$\Gamma; \Xi; \Delta^I \setminus \Delta^O \longrightarrow_v G$$

where each of Greek letters Γ, Ξ, Δ^I, and Δ^O denotes a set of uniquely labelled resource formulas, G is a goal formula, and v is a boolean flag (called $\top$-flag) that indicates whether a $\top$ was seen in the goal G.

Γ is an intuitionistic context. Ξ and Δ^I on the left-hand side of "$\setminus$" are called *input linear context*, Δ^O on the other side is called *output linear context*. The input linear context is divided into two parts: the *strict context* Ξ that must be entirely consumed during the proof search of the goal G, and the *lax context* Δ^I in which the contents might be consumed during the proof search of G. It is therefore possible for only the lax context Δ^I to transmit unused resources to the output context Δ^O. The intuitive meaning of $\Gamma; \Xi; \Delta^I \setminus \Delta^O \longrightarrow_v G$ is that Ξ and Δ^I are the linear contexts that are given as input to prove G. The proof of G will consume all of Ξ and part of Δ^I and return unused resources as the output context Δ^O.

$$\frac{}{\Gamma;\varnothing;\Delta^I \setminus \Delta^I \longrightarrow_0 1}\ (1) \qquad \frac{}{\Gamma;\Xi;\Delta^I \setminus \Delta^I \longrightarrow_1 \top}\ (\top)$$

$$\frac{\Gamma;\varnothing;(\Xi,\Delta^I) \setminus \Delta' \longrightarrow_0 G_1 \quad \Gamma;(\Xi \cap \Delta');(\Delta^I \cap \Delta') \setminus \Delta^O \longrightarrow_v G_2}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G_1 \otimes G_2}\ (\otimes_{0v})$$

$$\frac{\Gamma;\varnothing;(\Xi,\Delta^I) \setminus \Delta' \longrightarrow_1 G_1 \quad \Gamma;\varnothing;\Delta' \setminus \Delta^O \longrightarrow_v G_2}{\Gamma;\Xi;\Delta^I \setminus \Delta^I \cap \Delta^O \longrightarrow_1 G_1 \otimes G_2}\ (\otimes_{1v})$$

$$\frac{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_0 G_1 \quad \Gamma;(\Xi,\Delta^I - \Delta^O);\varnothing \setminus \_ \longrightarrow_v G_2}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_0 G_1 \,\&\, G_2}\ (\&_{0v})$$

$$\frac{\Gamma;\Xi;\Delta^I \setminus \Delta' \longrightarrow_1 G_1 \quad \Gamma;(\Xi,\Delta^I - \Delta');\Delta' \setminus \Delta^O \longrightarrow_v G_2}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G_1 \,\&\, G_2}\ (\&_{1v})$$

$$\frac{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G_i}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G_1 \oplus G_2}\ (\oplus_i) \qquad \frac{\Gamma;\varnothing;\varnothing \setminus \_ \longrightarrow_v G}{\Gamma;\varnothing;\Delta^I \setminus \Delta^I \longrightarrow_0 !\,G}\ (!)$$

$$\frac{\Gamma;(\Xi,R);\Delta^I \setminus \Delta^O \longrightarrow_v G}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v R \multimap G}\ (\multimap) \qquad \frac{(\Gamma,R);\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v R \Rightarrow G}\ (\Rightarrow)$$

$$\frac{}{\Gamma;\varnothing;\Delta^I \setminus \Delta^I \longrightarrow_0 A \doteq A}\ (\doteq) \qquad \frac{R \gg A \setminus G \quad (\Gamma,R);\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G}{(\Gamma,R);\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v A}\ (\mathrm{BC_{int}})$$

$$\frac{R \gg A \setminus G \quad \Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G}{\Gamma;\Xi;(\Delta^I,R) \setminus \Delta^O \longrightarrow_v A}\ (\mathrm{BC_{lax}}) \qquad \frac{R \gg A \setminus G \quad \Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G}{\Gamma;(\Xi,R);\Delta^I \setminus \Delta^O \longrightarrow_v A}\ (\mathrm{BC_{strict}})$$

Figure 2.7: The $\mathcal{RM}_3$ System for Propositional Fragment of Lolli

$$\frac{}{A' \gg A \setminus A' \doteq A}\ (\mathrm{dec\_atom}) \qquad \frac{}{\top \gg A \setminus 0}\ (\mathrm{dec\_}\top)$$

$$\frac{A' \gg A \setminus G'}{G \multimap A' \gg A \setminus G' \otimes G}\ (\mathrm{dec\_}\multimap) \qquad \frac{R_1 \gg A \setminus G_1 \quad R_2 \gg A \setminus G_2}{R_1 \,\&\, R_2 \gg A \setminus G_1 \oplus G_2}\ (\mathrm{dec\_}\&)$$

Figure 2.8: Residuation Rules for the Proof System $\mathcal{RM}_3$

Figure 2.7 presents the set of $\mathcal{RM}_3$ sequent rules. For example, the rule for $\&$ is split into two rules, differing in whether a $\top$ is seen in the left conjunct:

$$\frac{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_0 G_1 \quad \Gamma;(\Xi,\Delta^I - \Delta^O);\varnothing \setminus \_ \longrightarrow_v G_2}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_0 G_1 \,\&\, G_2}\ (\&_{0v})$$

$$\frac{\Gamma;\Xi;\Delta^I \setminus \Delta' \longrightarrow_1 G_1 \quad \Gamma;(\Xi,\Delta^I - \Delta');\Delta' \setminus \Delta^O \longrightarrow_v G_2}{\Gamma;\Xi;\Delta^I \setminus \Delta^O \longrightarrow_v G_1 \,\&\, G_2}\ (\&_{1v})$$

When the left conjunct does not encounter a $\top$ (in the $\&_{0v}$ case), the goal G_2 must consume all resources that has been consumed in G_1. That is, G_2 must consume not only Ξ (strict context) but also $\Delta^I - \Delta^O$, all the resources in lax context that G_1 has consumed. The right conjunct has empty lax context since G_2 may not consume any resources that G_1 did not. When the left conjunct encounters a $\top$ (in the $\&_{1v}$ case), the goal G_2 must consume all resources $\Xi, \Delta^I - \Delta'$ that has been consumed in G_1, but the unconsumed part of the lax context Δ' is transmitted to the lax context in the right conjunct. It is therefore possible for the

goal G_2 to consume any resources in Δ', since they are considered to have been also consumed in the left conjunct by the $\top$ goal.

When the goal formula is atomic, a resource formula R is selected from either the intuitionistic context ($\mathrm{BC}_{\mathrm{int}}$), lax context ($\mathrm{BC}_{\mathrm{lax}}$), or strict context ($\mathrm{BC}_{\mathrm{strict}}$). In either case, the resource formula R and atomic goal A are passed to the *residuation* procedure $R \gg A \setminus G$ for producing a continuation subgoal G. Figure 2.8 shows the residuation rules [2] for restrictive definition of resource formulas. It is noted that the symbol "$\doteq$" means the syntactic equality between atomic formulas.

Cervesato, Hodas, and Pfenning have proved that $\mathcal{RM}_3$ is logically equivalent to $\mathcal{L}'$. The techniques used in $\mathcal{RM}_3$ have been already applied to resource management systems [27][35] for Miller's Forum [41].

2.3.3 The $\mathcal{IOL}$ Implementation Model

Besides the $\mathcal{RM}_3$ model, one significant effort has been made towards developing compiler systems. The $\mathcal{I}/\mathcal{O}$ model removes the most serious non-determinism from the treatment of $\otimes$ by splitting the linear contexts lazily. However, its formulation requires the copying of large structures. This makes it more suited to implementation via interpreters written in high-level languages. Lolli has been actually implemented as interpreters in Prolog, λProlog, and standard ML. Let us consider again the rules for proving the goal $G_1 \,\&\, G_2$:

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

This rule means that the goal G_1 and G_2 must use the same set of resources. In a naive implementation, the system first copies the input context and proves the two conjuncts separately, and then it compares their output contexts. It is therefore impossible to change the contexts destructively during the proof search.

Tamura *et al.* solved this problem using *level indices* to control the consumption of resources [30][49] and proposed a refinement of the $\mathcal{I}/\mathcal{O}$ model, called the $\mathcal{IOL}$ model. The main difference is that all resources are kept in only one single context during the proof search, and the consumption of resources can be easily achieved by changing their consumption level destructively.

The $\mathcal{IOL}$ model makes use of two level indices L and U to manage the consumption of resources. In particular, these indices are used to quickly enable and disable consumption of resources in the context, and to keep track of when they have been consumed, respectively. The sequent is an expression of the form $\vdash_{L,U} I\,\{G\}\,O$ [3] where I and O are IOL-contexts, and G is a goal formula. L, a positive integer, is the *current consumption level*. At a given point in the proof, only linear resources labeled with that consumption level (and intuitionistic resources labeled with 0) can be used. U, a negative integer, is the *current consumption maker*. When a linear resource is consumed, its consumption level is changed to the value of U. The IOL-context is a list of pairs of the form $\langle R, \ell\rangle$, where R is a resource formula and ℓ is its consumption level. Each formula in the IOL-context can be classified by the value of this field:

Linear unconsumed formulas have the form $\langle R, \ell\rangle$, where ℓ is the value of L at which the resource may be consumed.

Linear consumed formulas have the form $\langle R, u\rangle$, where u is the value of U at the time when the resource was consumed.

Intuitionistic formulas in the context always take the form $\langle R, 0\rangle$.

Figure 2.9 presents the set of sequent rules of the $\mathcal{IOL}$ model. As with $\mathcal{I}/\mathcal{O}$ model, the BC_i rules handles the selection of resource clauses for backchaining. The relation $\mathit{pickR}_{L,U}(I, M, R)$ selects an available (linear unconsumed or intuitionistic) clause R from the input context I. The output context M is the same

[2] The rules for full fragment of Lolli have been shown in Cervesato's paper [13].

[3] In original paper [49], Tamura and Kaneda use the sequent of the from $\Gamma \longrightarrow_{L,U} I\,\{G\}\,O$ in which intuitionistic context Γ is separated out.

$$\frac{}{\vdash_{L,U} I\,\{1\}\,I}\ (1) \qquad \frac{\mathit{subcontext}_{U,L}(O,I)}{\vdash_{L,U} I\,\{\top\}\,O}\ (\top)$$

$$\frac{\vdash_{L,U} I\,\{G_1\}\,M \quad \vdash_{L,U} M\,\{G_2\}\,O}{\vdash_{L,U} I\,\{G_1 \otimes G_2\}\,O}\ (\otimes)$$

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

$$\frac{\vdash_{L,U} I\,\{G_i\}\,O}{\vdash_{L,U} I\,\{G_1 \oplus G_2\}\,O}\ (\oplus_i) \qquad \frac{\vdash_{L+1,U} I\,\{G\}\,O}{\vdash_{L,U} I\,\{!\,G\}\,O}\ (!)$$

$$\frac{\vdash_{L,U} [\langle R, L\rangle \mid I]\,\{G\}\,[\langle R, U\rangle \mid O]}{\vdash_{L,U} I\,\{R \multimap G\}\,O}\ (\multimap) \qquad \frac{\vdash_{L,U} [\langle R, 0\rangle \mid I]\,\{G\}\,[\langle R, 0\rangle \mid O]}{\vdash_{L,U} I\,\{R \Rightarrow G\}\,O}\ (\Rightarrow)$$

$$\frac{\mathit{pickR}_{L,U}(I,O,A)}{\vdash_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_1) \qquad \frac{\mathit{pickR}_{L,U}(I,M,G \multimap A) \quad \vdash_{L,U} M\,\{G\}\,O}{\vdash_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_2)$$

Figure 2.9: The $\mathcal{IOL}$ System for Propositional Fragment of Lolli

as I, except that the consumption level of the selected clause is changed to the value of U if it is a linear resource. In the $\top$ rule, the relation $\mathit{subcontext}_{U,L}(O,I)$ consumes any linear unconsumed resources from its input context I. The output context O is the same as I, except that the consumption levels of the consumed resources in I are changed to the value of U, if they are linear unconsumed resources.

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

The following two relations are used in the $\&$ rule. Each can be implemented destructively in one pass through the context.

- The $\mathit{change}_{\ell,\ell'}(M,N)$ relation modifies the context M so that any resources in M with level ℓ have their level changed to ℓ' in the context N.
- The $\mathit{thinable}_{\ell}(O)$ relation checks whether none of resources in O have ℓ as their consumption level.

Let us show the outline of the rule for proving $G_1 \,\&\, G_2$:

1. $\vdash_{L,U-1} I\,\{G_1\}\,M$
   Decrements U so that we know which resources are consumed during the proof search of G_1, and then it proves the goal G_1.
2. $\mathit{change}_{U-1,L+1}(M,N)$
   Changes the level of resources that have been consumed in G_1 to $L+1$.
3. $\vdash_{L+1,U} N\,\{G_2\}\,O$
   Increments L and U and proves the goal G_2.
4. $\mathit{thinable}_{L+1}(O)$
   Decrements L and checks whether none of resources in O have $L+1$ as their consumption level.

Tamura *et al.* have proved that $\mathcal{IOL}$ is logically equivalent to $\mathcal{L}'$. The techniques used in $\mathcal{IOL}$ have been already applied to a prototype compiler [49] for a significant subset of first-order Lolli. In the prototype compiler, the single IOL-context in which all resources are kept is implemented as an array structure, and the speed access to resources is achieved using hash tables.

2.3.4 The $\mathcal{LRM}$ Implementation Model

The $\mathcal{RM}_3$ model provides an efficient resource management for Lolli and related systems. However, $\mathcal{RM}_3$ is still more suited to implementation via interpreters in high-level languages rather than compilers, since it requires copying and scanning large dynamic data structures to control the consumption of linear resources. For example, most of rules needs complex operations on the linear context for moving formulas between strict and lax context and taking intersection of two context.

On the other hand, $\mathcal{IOL}$ provides an enriched formulation that is suited to implementation via not only interpreters but also compilers. However, a $\mathcal{IOL}$-based prototype compiler does not treat the goal $\top$ completely since $\mathcal{IOL}$ still depends on the *subcontext*$_{U,L}$ relation and have not removed non-determinism from the treatment of $\top$.

More recently, Hodas *et al.* solved these problems and have proposed new level-based resource management system, called the $\mathcal{LRM}$ model [29]. This model is a refinement of $\mathcal{RM}_3$ with $\mathcal{IOL}$'s level indices.

The LRM-sequent is an expression of the form $\vdash^{v}_{L,U} I\,\{G\}\,O$ where L and U are level indices, v is a $\top$-flag, I and O are LRM-contexts, and G is a goal formula. The main difference from $\mathcal{IOL}$ is that L is also used to set a *deadline* by which newly added resources must be used. The LRM-context is a list of triples of the form $\langle R, \ell, d\rangle$ [4], where R is a resource formula, ℓ is its consumption level, and d is its deadline. Each formula in the LRM-context can be classified by the values of these two fields:

Linear unconsumed formulas have the form $\langle R, \ell, d\rangle$, 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. This index can be seen as a kind of deadline, since if any resources exist with $d = L$ when it is time to decrement the level counter 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 $\langle R, u, 0\rangle$, where u is the value of U at the time when the resource was consumed.

Intuitionistic formulas in the context always take the form $\langle R, 0, 0\rangle$.

Figure 2.10 presents the set of $\mathcal{LRM}$ sequent rules. Each of the rules can be used in an implementation which destructively modifies the context, without copying (as long as some trailing mechanism exists to reverse the destructive modifications when backtracking).

As same with $\mathcal{IOL}$, the *pickR*$_{L,U}(I, O, R)$ relation in BC_i rules selects an available (linear unconsumed or intuitionistic) clause from the context I matching the atom A. The output context O is the same as I, but with the selected clause marked as linear consumed if it was linear unconsumed (O is exactly the same as I if it was intuitionistic).

The two $\mathcal{RM}_3$ rules for $\&$ are split into four rules, differing in whether $\top$ is seen in the right conjunct. The following three relations are used in the rule for $\&$:

- *consumed*$_\ell(O)$
  Let ℓ be an integer with $0 < \ell$. This is a relation on a LRM-context that is true if none of resources in the context O have ℓ as their deadline.
- *change*$_{\ell\to\ell'}(I, O)$
  Let ℓ and ℓ' be integers. This is a relation between an input and output LRM-context that modifies the input context I so that any resources in I with its level ℓ, has the level changed to ℓ' in the output O.
- *changepair*$_{(\ell,d)\to(\ell',d')}(I, O)$
  Let ℓ, ℓ', d, and d' be integers. This is a relation between an input and output LRM-context that modifies the input context I so that any resources in I with its level ℓ, and its deadline d, has the level changed to ℓ' and its deadline changed to d' in the output O.

[4] In original paper [29], Hodas *et al.* use the the from R^d_ℓ instead.

$$\frac{}{\vdash^0_{L,U} I\,\{1\}\,I}\ (1) \qquad \frac{}{\vdash^1_{L,U} I\,\{\top\}\,I}\ (\top)$$

$$\frac{\vdash^{v_1}_{L,U} I\,\{G_1\}\,M \qquad \vdash^{v_2}_{L,U} M\,\{G_2\}\,O}{\vdash^{v_1 or v_2}_{L,U} I\,\{G_1 \otimes G_2\}\,O}\ (\otimes)$$

$$\frac{\vdash^0_{L,U-1} I\,\{G_1\}\,M \quad \underset{(U-1,0)\to(L+1,L+1)}{\textit{changepair}}(M,M') \quad \vdash^0_{L+1,U} M'\,\{G_2\}\,O \quad \textit{consumed}_{L+1}(O)}{\vdash^0_{L,U} I\,\{G_1 \,\&\, G_2\}\,O}\ (\&_{00})$$

$$\frac{\vdash^0_{L,U-1} I\,\{G_1\}\,M \quad \underset{(U-1,0)\to(L+1,L+1)}{\textit{changepair}}(M,M') \quad \vdash^1_{L+1,U} M'\,\{G_2\}\,O' \quad \underset{(L+1,L+1)\to(U,0)}{\textit{changepair}}(O',O)}{\vdash^0_{L,U} I\,\{G_1 \,\&\, G_2\}\,O}\ (\&_{01})$$

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

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

$$\frac{\vdash^v_{L,U} I\,\{G_i\}\,O}{\vdash^v_{L,U} I\,\{G_1 \oplus G_2\}\,O}\ (\oplus_i) \qquad \frac{\vdash^v_{L+1,U-1} I\,\{G\}\,O}{\vdash^0_{L,U} I\,\{!\,G\}\,O}\ (!)$$

$$\frac{\vdash^v_{L,U} [\langle R,L,L\rangle \mid I]\,\{G\}\,[\langle R,U,0\rangle \mid O]}{\vdash^v_{L,U} I\,\{R \multimap G\}\,O}\ (\multimap) \qquad \frac{\vdash^1_{L,U} [\langle R,L,L\rangle \mid I]\,\{G\}\,[\langle R,L,L\rangle \mid O]}{\vdash^1_{L,U} I\,\{R \multimap G\}\,O}\ (\multimap_1)$$

$$\frac{\vdash^v_{L,U} [\langle R,0,0\rangle \mid I]\,\{G\}\,[\langle R,0,0\rangle \mid O]}{\vdash^v_{L,U} I\,\{R \Rightarrow G\}\,O}\ (\Rightarrow)$$

$$\frac{\textit{pickR}_{L,U}(I,O,A)}{\vdash^v_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_1) \qquad \frac{\textit{pickR}_{L,U}(I,M,G \multimap A) \quad \vdash^v_{L,U} M\,\{G\}\,O}{\vdash^v_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_2)$$

Figure 2.10: The $\mathcal{LRM}$ System for Propositional Fragment of Lolli

The outline for proving the sequent $\vdash^v_{L,U} I\,\{G_1 \,\&\, G_2\}\,O$ is as follows:

1. $\vdash^{v_1}_{L,U-1} I\,\{G_1\}\,M$
   Decrements U so that we can tell which linear resources are consumed during the proof search of G_1, and those resources will have the new value of U as the value of their consumption level. After that it proves the goal G_1. The $\top$-flag v_1 is set to 1 if G_1 encounters a $\top$, otherwise v_1 is set to 0.
2. $\underset{(U-1,0)\to(L+1,L+1)}{\textit{changepair}}(M,M')$
   Changes both the consumption level and deadline of all of the resources that have been consumed in G_1 to $L+1$, to which L will be set during the proof search of G_2. This is because those resources (strict context of $\mathcal{RM}_3$) can and must be consumed in G_2.
3. If the value of v_1 is 1, $\underset{L\to L+1}{\textit{change}}(M',M'')$
   Changes the consumption level of all unconsumed resources in G_1 (identified by having a consumption level with the same value as L) to $L+1$, so that resources (lax context of $\mathcal{RM}_3$) that were not explicitly consumed but are considered to have been also consumed by $\top$, are also available for G_2.

4. $\vdash^{v_2}_{L+1,U} M'' \{G_2\} O''$
   Increments L and U and proves the goal G_2. Decrements L. The $\top$-flag v_2 is set to 1 if G_2 encounters a $\top$, otherwise v_2 is set to 0.
5. If the value of v_2 is 1, $\overset{changepair}{(L+1,L+1)\rightarrow(U,0)}(O'', O')$
   Changes the consumption level and deadline of resources that have been consumed in G_1 but not in G_2 to U and 0 respectively, since those resources are considered to have been also consumed by $\top$. Otherwise, $consumed_{L+1}(O')$ checks whether none of resources have $L+1$ as their deadline, since all the resources that have been consumed in G_1 must be consumed in G_2. If this fail, fail.
6. If the value of v_1 is 1, $\overset{change}{L+1\rightarrow L}(O', O)$
   Put back the consumption level of those resources that were made available to G_2 because G_1 encounter a $\top$, but were not consumed in G_2, to their original level, L.
7. The $\top$-flag v is set to $v_1 \cap v_2$.

Hodas *et al.* have proved that the $\mathcal{LRM}$ model is logically equivalent to $\mathcal{RM}_3$. It is the $\mathcal{LRM}$ model that we shall use to design an extension of standard WAM (Warren Abstract Machine) for the Lolli language described in chapter 4 and 5.

2.4 The Syntax of the Lolli Language

As with λProlog, the full language of Lolli allows nested quantification and the use of L_λ higher-order quantification and unification of λ-term. In original Lolli syntax, terms and atoms are written in curried form such as the functional programming language ML, since such notation is more suitable for higher-order programming features.

Although these features are important aspects of Lolli, we use conventional Prolog syntax since our focus is on efficient implementation of the first-order Lolli language. The syntax of Lolli operators that we use in this dissertation is summarized in Table 2.1. The order of operator precedence is "`forall`", "`exists`", "`\`", "`:-`", "`<=`", "`;`", "`&`", "`,`", "`-<>`", "`=>`", "`!`" from wider to narrower. The main difference from original Lolli syntax is that linear implication is written as `-<>` instead of `-o`, and bang operator is written as `!` instead of $\{\ldots\}$.

Lolli Syntax	Linear Logic Operator
`true`	1
`erase`	$\top$
B`,`C	$B \otimes C$
B`&`C	$B \,\&\, C$
B`;`C	$B \oplus C$
B`-<>`C	$B \multimap C$
C`:-`B	$B \multimap C$
B`=>`C	$B \Rightarrow C$
C`<=`B	$B \Rightarrow C$
`!`B	$!B$
`forall X\`B	$\forall x.B$
`exists X\`B	$\exists x.B$

Table 2.1: The Mapping Between Linear Logic Operators and Lolli Syntax

Chapter 3

A Collection of Lolli Programming Examples

We have presented the theoretical aspect of the Lolli language so far. In this chapter, we will give a brief introduction to Lolli programming. We will also presents several example programs so that the reader easily understand a sense of *resource programming* in Lolli. Compared with Prolog, the biggest difference of Lolli is its resource consciousness. In Lolli, it is possible to add resources (limited-use clauses) to the program and consume them dynamically.

Other 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 [26].

3.1 A Brief Introduction to Lolli Programming

3.1.1 Resource Addition

The linear implication `-<>` is used to add resources which can be consumed exactly once. A query adds a resource `r(1)` to the program and then executes a goal `r(X)`:

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

which succeeds by letting $\texttt{X} = \texttt{1}$. It is noted that added resources must be consumed during the execution of the subgoal on the right-hand side of the implication. Thus, the following query fails since `r(1)` is not consumed.

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

If a resource clause A`:-`G (or G`-<>`A) is added, the subgoal G will be executed just on the consumption of A. Thus, the following query succeeds and displays `1`.

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

Informally, we allow a $\otimes$-product of multiple resources to appear on the left-hand side of linear implication. The following query adds resources `r(1)` and `r(2)` and then executes a goal `r(X), r(Y)`:

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

which also succeeds by letting $\texttt{X} = \texttt{1}$ and $\texttt{Y} = \texttt{2}$, or $\texttt{X} = \texttt{2}$ and $\texttt{Y} = \texttt{1}$. It is noted that such a query should be written with successive uses of the linear implication formally.

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

The resource $R_1 \& R_2$ means a selective resource. When `r(1)&r(2)` is added to the program, 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).
```

The intuitionistic implication `=>` is used to add infinitely reusable resources, which can be consumed arbitrarily many times (including zero times). The following query succeeds by letting `X` $=$ `1` or `X` $=$ `2`.

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

In Lolli, the two implication operators add resources to the program dynamically during the execution time. However, they are not the same as the Prolog `assert` mechanism. First, the addition is scoped over the subgoal on the right-hand side of the implication, but an `assert`ed clause in Prolog remains until it is `retract`ed. So, the following query will fail:

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

Second, although Prolog's `assert` automatically universalizes any free variables in an added clause, in Lolli clauses added with implication can contain free variables, which may get bound when the clause is consumed. For example, the following Prolog query will succeed, since the variable `X` is universalized.

```
?- assert(r(X)), r(1), r(2).
```

In contrast, the similar Lolli query:

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

will fail, since the execution of `r(1)` causes the variable `X` to be instantiated to `1`. If we desire the other behavior, we must quantify explicitly:

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

3.1.2 Resource Consumption

In Lolli, the execution of atomic goals means resource consumption and program invocation. All possibilities are attempted by backtracking. For example, the following query displays `1` and `2`.

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

The goal $G_1 \& G_2$ behaves as well as G_1`,` G_2, but the same resources must be consumed in G_1 and G_2. The following query succeeds by letting `X` $=$ `Y` $=$ `1` and `Z` $=$ `2`, or `X` $=$ `Y` $=$ `2` and `Z` $=$ `1`, because `r(X)` and `r(Y)` must consume the same resources.

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

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

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

The goal `erase` (or `top`) means the consumption of some consumable resources. In the following queries:

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

the first one succeeds, but the second one fails.

```
reverse(Xs, Zs) :-
    reverse(Xs, [], Zs).

reverse([], Zs, Zs).
reverse([X|Xs], Ys, Zs) :-
    reverse(Xs, [X|Ys], Zs).
```

Figure 3.1: A Prolog Program for Reversing a List

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

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

Figure 3.2: A Lolli Program for Reversing a List

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

Figure 3.3: A Lolli Program for Filtering a List

3.2 Using Free Variables in Resources

3.2.1 Reversing a List

Let us consider an example program for reversing a list. In Prolog, we need one extra argument (the second argument) in `reverse/3` to store a list that have been reversed during the execution time. In Lolli, we do not need it since the resource `result(Zs)` is used to receive the result from the deepest recursive call of `rev/2`. The free variables `Zs` will get bound to the reversed list when the goal `result(Zs)` is called from `rev/2`.

For example, the goal `reverse([1,2,3],Zs)` adds the resource `result(Zs)` and executes the subgoal `rev([1,2,3],[])`. On the third recursive call, `rev([],[3,2,1])` consumes `result(Zs)`, and `Zs` is unified with `[3,2,1]`.

The same technique can be used to describe “accumulators”. A program calculating the summation of a given list can be written as follows.

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

3.2.2 Filtering a List

Figure 3.3 shows a simple example for filtering a list with a given condition. For example, when the goal `choose([1,2,3],2,Z)` is executed, the reusable clause “`forall X\ test(X) :- X>2`” is added to the program, and the subgoal `filter([1,2,3],Z)` is executed. The added resource is used in the second clause of `filter/2` to check whether each element satisfies the condition or not.

It is noted that the addition is not the same as the Prolog `assert` mechanism. This is because the free variable `Y` in the added clause is instantiated to 2, and the added clause `test/1` is scoped over the subgoal `filter/2`.

3.3 Using Resources as Limited-Use Data

3.3.1 N-Queens

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

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

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

Figure 3.4: A Lolli Program for N-Queens

```
queens(N,Q) :-
    gen(1,N,Js),
    N2 is 2*N-1, gen(2,N2,Us),
    D0 is 1-N, gen(D0,N2,Ds),
    sol(N,Js,Us,Ds,Q).

sol(0,_,_,_,[]).
sol(I,Js0,Us0,Ds0,[J|Q]) :-
    I > 0, del(J,Js0,Js),
    U is I+J, del(U,Us0,Us),
    D is I-J, del(D,Ds0,Ds),
    I1 is I-1, sol(I1,Js,Us,Ds,Q).

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

gen(_,0,[]).
gen(I,N,[I|Ns]) :-
    N>0, I1 is I+1, N1 is N-1,
    gen(I1,N1,Ns).
```

Figure 3.5: A Prolog Program for N-Queens in Prolog Programming for Artificial Intelligence [11]

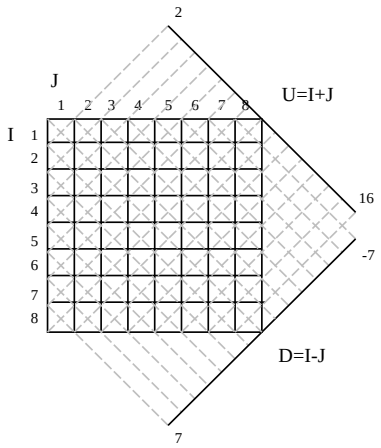


Figure 3.6: Resources for 8-Queens

Let us consider the N-Queens problem in Figure 3.5. The `Js`, `Ds` and `Us` in `sol/5` are lists indicating columns and diagonals (see Figure 3.6). Thus, the safety check of the place (i, j) on the board is done by deleting j from `Js`, $i - j$ from `Ds`, and $i + j$ from `Us`. However, this program heavily rely on significant manipulation and construction of list structures.

The same technique can be used in Lolli program in Figure 3.4. In Lolli, resources can be used to represent columns (`c/1`), and diagonals (`d/1` and `u/1`), rather than list structures. Thus, the safety check

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

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

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

Figure 3.7: A Lolli Program for Knight Tour

of the place (i, j) on the board is done quickly by consuming the resources `c(`j`)`, `u(`$i+j$`)`, and `d(`$i-j$`)`.

For example, when the goal `queen(8,Q)` is executed, the subgoal `place(8)` adds the resources: `c(1)`,...,`c(8)`, `u(2)`,...,`u(16)`, `d(-7)`,...,`d(7)`, and then `solve(8,[])` is called. The `solve/2` predicate tries to find a solution by consuming `c(`j`)`, `u(`$i+j$`)`, and `d(`$i-j$`)` for each row $i = 1..8$ and column j. It is noted that unused resources are consumed implicitly by the `erase`.

3.3.2 Knight Tour

Figure 3.7 shows a Lolli example for finding a Hamilton path on the 5×5 chess board. In Hamilton path, all vertices are visited exactly once. This constraint can be represented easily by using linear resources as vertices.

The goal `knight5(Tour)` adds the resources `k(1,1)`,...,`k(5,5)` that indicate 25 vertices and then executes subgoal `tour(1, 1, Tour)`. The subgoal tries to find a Hamilton path starting from the position $(1, 1)$. Since visited vertices are consumed during the execution time, the goal `knight5(Tour)` succeeds only when all vertices are visited exactly once.

3.3.3 Kirkman's School Girl Problem

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

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

The Lolli program in Figure 3.8 behaves as follows.

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

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

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

arrange(0,                  []).
arrange(I, [[G1,G2,G3]|Groups]) :-
        I > 0,
        g(G1), g(G2), g(G3), % select 3 girls
        % check if they have not yet met each other
        meet(G1, G2), meet(G1, G3), meet(G2, G3),
        I1 is I-1,
        arrange(I1, Groups).
```

Figure 3.8: A Lolli Program for Kirkman's School Girl Problem

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

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

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

3.3.4 Cryptarithmetic Puzzle

Figure 3.9 shows a Lolli program to solve a cryptarithmetic puzzle: "SEND+MORE=MONEY". Each digit i is represented as a linear resource `d(`i`)`. The goal `erase` is used to express the condition a digit can be used at most once since it consumes unused resources implicitly.

3.4 Using Resources as Limited-Use Clauses

3.4.1 Path Finding

Figure 3.10 shows a Lolli example for finding 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.

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

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

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

Figure 3.9: A Lolli Program for Cryptarithmetic Puzzle

```
path :-
    (a -<> b) -<>        % add the arc from a to b
    (b -<> c) -<>        % add the arc from b to c
    (c -<> a) -<>        % add the arc from c to a
    (c -<> d) -<>        % add the arc from c to d
    (d -<> b) -<>        % add the arc from d to b
     a -<> (d, erase).   % find a path from a to d
```

Figure 3.10: A Lolli Program for Path Finding

- A path is the transitive closure of the arc connected relation.

3.4.2 Tiling Board with Dominoes

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. Let the board size be (m, n). A Lolli program in Figure 3.11 can elegantly expressed the following conditions:

- 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.

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

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, !,
        (
            ( domino(N) :- d(I, J, N), J1 is J+1, d(I, J1, N) )
              &
            ( domino(N) :- d(I, J, N), I1 is I+1, d(I1, J, N) )
        )
        -<> (N1 is N-1, gen_res(N1)).

gen_res(I, J) :- I < 1, !, num_of_dominos(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, d(I, J, _) -<> gen_res(I, J1).
```

Figure 3.11: A Lolli Program for Tiling Board with Dominoes

3.5 A Lean Connection Theorem Prover for First-Order Classical Logic

We have discussed simple examples of Lolli so far. Now we have shown a more sophisticated application of Lolli.

Recently Hodas and Tamura have reimplemented the leanCoP connection-calculus theorem prover of Otten and Bibel [46] in Lolli. This "lean" theorem prover has been shown to have remarkably good performance relative to state-of-the-art systems, particularly considering that it is implemented in just a half-page of Prolog code in Figure 3.12.

The reimplemented prover, lolliCoP [28], is of comparable size, and, when compiled under LLP (the reference Lolli compiler), provides a speedup of 40% over leanCoP. Figure 3.13 shows the source code of lolliCoP. Performance evaluation of lolliCoP is shown in chapter 5.

Here, we point out only the biggest differences between two implementations, and omit their whole behavior in detail. In leanCoP, the technique used to select literals from clauses and clauses from matrices relies on significant manipulation and construction of list structures on the heap. However, in lolliCoP, that technique can be replaced by the efficient resource management base on $\mathcal{LRM}$ at the formula level in Lolli.

```
prove(Mat) :- prove(Mat,1).

prove(Mat,PathLim) :-
    append(MatA,[Cla|MatB],Mat), \+member(-_,Cla),
    append(MatA,MatB,Mat1), prove([!],[[-!|Cla]|Mat1],[],PathLim).
prove(Mat,PathLim) :-
    \+ground(Mat), PathLim1 is PathLim+1, prove(Mat,PathLim1).

prove([],_,_,_).
prove([Lit|Cla],Mat,Path,PathLim) :-
    (-NegLit=Lit; -Lit=NegLit) ->
    ( member_oc(NegLit,Path) ;
      append(MatA,[Cla1|MatB],Mat), copy_term(Cla1,Cla2),
      append_oc(ClaA,[NegLit|ClaB],Cla2), append(ClaA,ClaB,Cla3),
      ( Cla1==Cla2 -> append(MatB,MatA,Mat1)
                    ; length(Path,K), K<PathLim,
                      append(MatB,[Cla1|MatA],Mat1)
      ), prove(Cla3,Mat1,[Lit|Path],PathLim)
    ), prove(Cla,Mat,Path,PathLim).
```

Figure 3.12: The leanCoP Theorem Prover of Otten and Bibel

```
prove(Mat) :- reverse(Mat,Mat1),
              (ground(Mat) -> propositional => pr(Mat1)
                            ; pr(Mat1)
              ).

pr([])        :- p(1).
pr([Cla|Mat]) :- (ground(Cla) -> (cl(Cla) -<> pr(Mat))
                               ; (cl(Cla)  => pr(Mat))
                 ).

p(PathLim) :- cl(Cla), \+member(-_,Cla),
              copy_term(Cla,Cla1), prove(Cla1,PathLim).

p(PathLim) :- \+propositional,
              PathLim1 is PathLim+1, p(PathLim1).

prove([],_) :- erase.
prove([Lit|Cla],PathLim) :-
  (-NegLit=Lit; -Lit=NegLit) ->
    ( path(NegLit), erase ;
      cl(Cla1), copy_term(Cla1,Cla2), append(ClaA,[NegLit|ClaB],Cla2),
      append(ClaA,ClaB,Cla3), (Cla1==Cla2 -> true ; PathLim>0),
      PathLim1 is PathLim-1, path(Lit) => prove(Cla3,PathLim1)
    ) & prove(Cla,PathLim).
```

Figure 3.13: The lolliCoP Theorem Prover of Hodas and Tamura

Chapter 4

Towards a Efficient Implementation for a Linear Logic Programming Language

In recent years, a number of logic programming languages based on linear logic have been proposed: LO [3], LinLog [2], Lolli [25, 26], ACL [32, 33], Lygon [20, 21], Forum [27, 41], and Linear LF [14]. In addition, BinProlog [51, 52] allows the use of linear implications of affine logic (a variant of linear logic). In these languages, it is possible to add and consume resources (limited-use clauses) dynamically as logical formulas. The efficient treatment of resources is therefore an important issue for the implementor.

In this chapter we discuss some basic issues on implementation of the Lolli language, especialy compiling resources.

4.1 Implementation Design

There seem to be at least three approaches to implement efficient Lolli systems:

1. Lolli interpreter in high-level languages,
2. Translating Lolli into existing languages: Prolog, λProlog, C, C++, Java, and others,
3. Compiling Lolli into WAM-like abstract machine code.

The approach (1) is the simplest. So far Lolli has been implemented as interpreters in Prolog, λProlog, and standard ML. Figure 4.1 shows a $\mathcal{I}/\mathcal{O}$ model-based Lolli interpreter written in pure Prolog. However, resources are represented as list structures, and their management heavily relies on significant manipulation and construction of list structures. This leads to slowdown in performance for large-scaled applications.

For the approach (2), let us consider translating Lolli into similar languages, Prolog and λProlog. This approach might provide better performances, but meet some problems in resource management. In Prolog, resources might be still represented as list structures. Even if we use the `assert` mechanism for handling resources, there are two difficulties. First, the addition of resources is scoped over the subgoal on the right-hand side of the implication, but an `asserted` clause remains until it is `retracted`. Second, the added resources can contain free variables that may get bound when the clause is consumed, but the `assert` automatically universalizes any free variables of added clauses. Using λProlog instead might solve these problems since it allows the use of intuitionistic implication ($\Rightarrow$ in Lolli) in goals. However, λProlog have not supported linear implication. To handle the consumption of linear resources, we need to add an extra argument to all resources for checking if they are linear or intuitionistic. This leads to unnecessary backtracking.

```
:- op(1060, xfy, (&)  ).
:- op( 950, xfy, (-<>)).
:- op( 950, xfy, (=>) ).
:- op( 900,  fy, (!)  ).

prove(G) :- I = [], O = [], prove(G, I, O).

prove(true,      I, I) :- !.
prove(erase,     I, O) :- !, subcontext(O, I).
prove((G1,G2),   I, O) :- !, prove(G1, I, M), prove(G2, M, O).
prove((G1&G2),   I, O) :- !, prove(G1, I, O), prove(G2, I, O).
prove((G1;G2),   I, O) :- !, (prove(G1, I, O) ; prove(G2, I, O)).
prove(!G,        I, I) :- !, prove(G, I, I).
prove((R -<> G), I, O) :- !, prove(G, [R|I], [1|O]).
prove((R => G),  I, O) :- !, prove(G, [!R|I], [!R|O]).
prove(A,         I, O) :- pick(I, O, A).         % A is an atomic formula
prove(A,         I, O) :- pick(I, M, (G -<> A)), % A is an atomic formula
                          prove(G, M, O).

subcontext([],    []   ).
subcontext([1|O], [R|I]) :- \+ (R = !(_)), subcontext(O, I).
subcontext([R|O], [R|I]) :- subcontext(O, I).

pick(I,      I,      S) :- rule(S).
pick([R|I],  [1|I],  S) :- \+ (R = !(_)), pick_S(R, S).
pick([!R|I], [!R|I], S) :- pick_S(R, S).
pick([R|I],  [R|O],  S) :- pick(I, O, S).

pick_S((R1&R2), S) :- !, (pick_S(R1, S) ; pick_S(R2, S)).
pick_S(S,       S).

rule( append([], Zs, Zs) ).
rule(( append(Xs, Ys, Zs) -<> append([X|Xs], Ys, [X|Zs]) )).
```

Figure 4.1: A $\mathcal{I}/\mathcal{O}$ Model-Based Lolli Interpreter in Prolog

For the approach (2), let us consider translating Lolli into C and Java. First, using C might have the advantage of giving nice speedup in performance, avoiding the overhead of emulators, and producing stand-alone executable code. For Prolog, Philippe Codognet and Daniel Diaz have developed the `WAMCC` system [16], that translates Prolog into C via the WAM. Second, using Java might have the advantages of portability, extensibility, and interactivity with Java. For Prolog, Bart Demoen and Paul Tarau have developed the jProlog system [18], that translates Prolog into Java via the WAM. Note that, both of them is based on the WAM (or its variant). It is therefore very important to design an abstract machine for Lolli.

The approach (3), compiling Lolli into WAM-like abstract machine code, might give nice speedup in performance and became the basis of the approach (2). Recently, N. Tamura and Y. Kaneda proposed the LLPAM [49], an extension of the standard WAM [1, 56] for significant subset of Lolli. However LLPAM was logically incomplete in the treatment of $\top$ since it was based on the $\mathcal{IOL}$ model. More recently, Hodas *et al.* solved this problem and proposed a refinement [29] based on the $\mathcal{LRM}$ model. However, the refinement supported only a small fragment of resources: A and $R_1 \,\&\, R_2$. Furthermore they were stored as terms in heap memory, rather than compiled into LLPAM code.

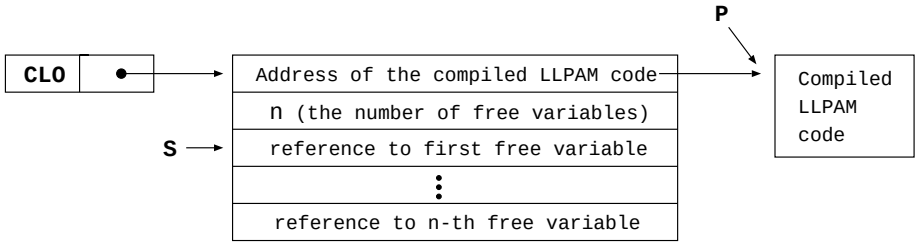


Figure 4.2: Closure Structure

4.2 Compiling Resource Formulas

The compilation of resources without free variables is straightforward. They can be compiled just as usual Prolog clauses because they don't require a set of bindings of free variable. For example, the resource formula $\forall X.\forall Y.(q(X,Y) \multimap p(f(X),Y))$ can be compiled just like Prolog 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, let us consider the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$, where X is a free variable. This resource can not be compiled like a usual Prolog clause since 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 a reference of compiled code and a set of bindings for free variables (see Figure 4.2). The new instruction `put_closure` is used to create a closure structure:

```
put_closure L, 1, A5
unify_value A4
```

This code creates new closure cell tagged by `CLO` in `A5`, sets the label `L` of compiled code for the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$, and sets the mode to write. The `unify_value` sets the free variable X.

When the closure is called, the WAM register `S` is set to point to the third cell of the closure structure, the mode is set to read, and the instruction pointer `P` is set to the address of the compiled code. To retrieve the values of free variables, the compiled coded will begin with `unify_variable` instructions.

The code generated for the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$ is as follows:

```
L:   unify_variable A3
     get_structure f/1, A1
     unify_value A3
     put_value A3, A1
     execute q/2
```

The idea of closure has been widely used in implementation of functional programming languages. In λProlog [42, 43, 45], G. Nadathur *et al.* have used it for compiling a clause D in the goal $D \supset G$ where the $\supset$ operator corresponds to $\Rightarrow$ in Lolli.

4.3 LLP: A Compiled Linear Logic Programming Language

4.3.1 The Definition of LLP

The LLP language is based on the following fragment of linear logic:

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

The letters C, G and R stand for "clause", "goal" and "resource" respectively. Compared with the fragment used in the original LLPAM papers, $G \multimap R$, $G \Rightarrow R$, and $\forall x.R$ are newly added to resource formulas. LLP supports the full fragment of first-order Lolli except of universal quantifiers in goal ($\forall x.G$). It is beyond the scope of this dissertation to deal with higher-order quantification and unification of λ-terms in Lolli. In spite of these limitations, LLP is expressive enough to cover the example programs in chapter 3.

4.3.2 Pre-Compilation of LLP

It is very convenient for the implementors to view a resource formula $\forall x.(G_1 \Rightarrow (G_2 \multimap A))$ as a resource formula $\forall x.((!\,G_1 \otimes G_2) \multimap A)$ whose head is an atomic formula such as Prolog clause.

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

By rewriting formulas using above logical equivalences in the forward direction, it is possible to simplify the definition of resource formulas:

$$\begin{array}{rcl} R & ::= & S_1 \,\&\, \cdots \,\&\, S_m \\ S & ::= & \top \mid A \mid G \multimap A \mid \forall x.S \end{array}$$

where A stands for a atomic formula and $m \geq 1$. S-formulas are called *resource clauses* in which the form $\forall x.(G \multimap A)$ corresponds to A :- G in Prolog. Since the above translation is done during the compilation time, we can write LLP programs in the original definition.

Chapter 5

The LLPAM Abstract Machine

LLP is implemented as a compiler to the LLP Abstract Machine (LLPAM), an extension of the Warren Abstract Machine (WAM) [1][56]. The extension is mainly for compiling resources and efficient resource management based on the $\mathcal{LRM}$ model.

In this chapter, we present the detail of LLPAM, particularly the differences from the WAM. Furthermore, we summarize the LLPAM instruction set and memory layout in Appendix A.

5.1 New Registers

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

- `R` (top of resource table)
  is a non-negative integer index indicating the current top of resource table, a new data area which is described below. The value of `R` increases as resource clauses are added to the table by implicational goals, and decreases on backtracking. Its initial value is 0.

- `L` (consumption level)
  is a positive integer indicating the current consumption level, which are assigned to resources when they are added to the table. This corresponds to the current value of L as used in $\mathcal{LRM}$. Its initial value is 1.

- `U` (consumption maker)
  is a negative integer indicating the current consumption maker, which are assigned to resources as they are consumed. This corresponds to the current value of U as used in $\mathcal{LRM}$. Its initial value is -1.

- `T` ($\top$ flag)
  is a boolean flag indicating whether $\top$ has been seen as a subgoal at the current level. Its initial value is false.

The values of these registers must be recorded in each choice point frame regardless of whether LLP programs make use of the resource management features or not.

Besides of these registers, LLPAM makes use of four auxiliary registers: `R0`, `R1`, `R2`, and `RLIST`. Every time when the resource of the form $S_1 \,\&\, \cdots \,\&\, S_n$ ($\&$-product of resource) is added by linear implication $\multimap$, `R0` is set to the index value of first resource clause (S_1). Every time a `call` procedure is invoked, `R1` and `R2` are set to the lists that contain the indices of possibly consumable resources in the resource table. `RLIST` is always set to the list that contains the indices of linear resources, not intuitionistic resources.

5.2 New Data Areas

LLPAM has three new data areas: RES (the resource table), HASH (the hash table), and SYMBOL (the symbol table) in addition to CODE, HEAP, STACK, TRAIL, and PDL in the WAM.

5.2.1 The Resource Table

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

```
record
    s1: Integer;
    s2: Integer;
    level: Integer;
    deadline: Integer;
    out_of_scope: Boolean;
    pred: symbol;
    closure: closure;
    head: term;
    body: term;
end;
```

RES grows when resources are added by $-\circ$ or $\Rightarrow$, and shrinks on backtracking. A entry in RES represents either a linear resource or intuitionistic resource, depending on which implication was used to add it. Each entry corresponds to a single resource clause.

The fields of the record are assigned as follows:

- When the resource of the form $S_1 \,\&\, \cdots \,\&\, S_n$ ($\&$-product of resource clauses) is added by linear implication $-\circ$, new n entries are created because the individual S_i are added individually. Nevertheless, we must remember that only one of them may be consumed. The field s1 of their entries is set to the index value of the first resource clause S_1 (the value of R0). The field s2 is set to the current top of the resource table (the value of R). These fields are used to maintain the exclusivity of $\&$-producted resource. If one resource clause S_i is consumed, all n entries are so marked and become unavailable. It is noted that when added by $\Rightarrow$ we need not to set these fields since intuitionistic resource can be used infinitely.

- The fields level and deadline correspond to the l and d values attached to resources in $\mathcal{LRM}$. They give the level at or below which the resource may be used, and the level by which the resource must be used. For linear resources the initial values of both fields are taken from the value of the L register at the time the entry is added. For intuitionistic resources, the two fields are set to 0.

- The out_of_scope flag is initially false. It is set to true when the subgoal (on the right side of the implication that added this resource) is completed, and the resource becomes out of scope. This flag is used because the resource table shrinks only on backtracking.

- The field pred is set to the predicate symbol of head of added resource clause. The field closure contains a pointer to the closure structure of added resource. It is noted that the fields head and body are used to contain the head and body term respectively only when the resource clause are added as term level from an interpreter.

Figure 5.1 shows the contents of the resource table after adding the resource $(p(1) \,\&\, (q(X) -\circ p(X))) \otimes \forall Y.r(Y)$ by the implication $-\circ$.

	s1	s2	level	deadline	out_of_scope	pred	closure	head	body
RES[0]	0	2	1	1	false	$p/1$	closure of $p(1)$	nil	nil
RES[1]	0	2	1	1	false	$p/1$	closure of $q(X) \multimap p(X)$	nil	nil
RES[2]	2	3	1	1	false	$r/1$	closure of $\forall Y.r(Y)$	nil	nil

Figure 5.1: Resource Table After Adding the Linear Resource $(p(1) \,\&(q(X) \multimap p(X))) \otimes \forall Y.r(Y)$.

	print_name	arity	codeaddr	res	res2
SYMBOL[$p/1$]	p	1	code of predicate $p/1$	[0,1]	[1]
SYMBOL[$r/1$]	r	1	code of predicate $r/1$	[2]	[2]

Figure 5.2: Symbol Table After Adding the Resource $(p(1) \,\&(q(X) \multimap p(X))) \otimes \forall Y.r(Y)$

5.2.2 The Hash and Symbol Tables

The LLPAM has also a symbol table `SYMBOL` with the following structure:

```
record
    print_name: Char;
    arity: Integer;
    hash_value: Integer;
    codeaddr: code address;
    res:  term;
    res2: term;
end;
```

The field `res` contains a list of indices of all resources that its predicate symbol is `print_name`/`arity`. The field `res2` contains a list of indices of resources that its predicate symbol is `print_name`/`arity` and its first argument is an unbound logical variable.

The hash table `HASH` is used to speed access to the resources. The entries are hashed on the predicate symbol and the first argument. Looking-up a resource is done through the hash table and the symbol table. We can't always rely on the hash table for access to the resources. When the atomic goal with p/n has a logical variable as its first argument, we must access all entries for p/n, regardless of first argument. In this case, we will use the value of `Symbol[`p/n`].res`. Similarly, since those resources in which the first argument is a logical variable must be examined for every call on p/n, we will use the value of `Symbol[`p/n`].res2`. Figure 5.2 shows a contents of `SYMBOL` corresponding to the Figure 5.1.

5.3 LLPAM Code Generation

The code generated for each operator in a goal represents an imperative implementation of the $\mathcal{LRM}$ rule for the operator.

5.3.1 Code for $G_1 \otimes G_2$

The code generated for $G_1 \otimes G_2$ is simply as follows:

Code for G_1
Code for G_2

5.3.2 Code for $R \multimap G$

An implicational goal$R \multimap G$ requires adding the linear resource R to the resource table and then executing the goal G. Further, the resource R must be used during the proof of G. It is noted that R has been already converted into a resource clause (or $\&$-product of resource clauses) during the compilation time.

Suppose that R is converted to $\&$-product of resource clauses $S_1 \,\&\, \cdots \,\&\, S_m$, the code generated for $R \multimap G$ is as follows:

```
begin_imp Y_i
Code for the addition of S_1
        ⋮
Code for the addition of S_m
mid_imp Y_j, Y_k
Code for G
end_imp Y_i, Y_j, Y_k
```

The following new instructions are used in the code generated for the goal $R \multimap G$. The code generation of resource addition will be described in section 5.3.4.

- `begin_imp` $\mathtt{Y}_i$
  Store the current value of `R` in a new permanent variable $\mathtt{Y}_i$. Save the current value of `R` in `R0`.
- `mid_imp` $\mathtt{Y}_j$, $\mathtt{Y}_k$
  Store the current values of `R` (the top of the resource table) and `T` ($\top$-flag) to the permanent variables $\mathtt{Y}_j$ and $\mathtt{Y}_k$, respectively. Set the value of `T` to false. It is noted that the newly added resource clauses lie in positions $\mathtt{Y}_i$ through $\mathtt{Y}_j - 1$. Set the `s1` and `s2` fields of all records from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ to the current values of `R0` and `R`, respectively.
- `end_imp` $\mathtt{Y}_i$, $\mathtt{Y}_j$, $\mathtt{Y}_k$
  If there are any resources in positions from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ that have not been consumed, fail. Otherwise, set the `out_of_scope` flags of all records from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ to true (trailing so that they may be reset on backtracking), and set the register `T` to $\mathtt{Y}_k \vee \mathtt{T}$. In order to account for the use of $\top$ at the top level of the subgoal, 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 unconsumed.
  - 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.

5.3.3 Code for $R \Rightarrow G$

An implicational goal $R \Rightarrow G$ requires adding the intuitionistic resource R to the resource table and then executing the goal G.

Suppose that R is converted to $\&$-product of resource clauses $S_1 \,\&\, \cdots \,\&\, S_m$, the code generated for $R \Rightarrow G$ is as follows:

```
begin_exp_imp Y_i
Code for the addition of S_1
        ⋮
Code for the addition of S_m
mid_exp_imp Y_j, Y_k
Code for  G
end_exp_imp Y_i, Y_j, Y_k
```

The following new instructions are used in the code generated for an implication of the form $R \Rightarrow G$. If the goal used $\Rightarrow$ rather than $-\circ$, then the behavior of generated code would be the almost same. Thus, we point out the only differences from $R -\circ G$. The code generation of resource addition will be described in section 5.3.4.

- `begin_exp_imp` $\mathtt{Y}_i$
  Behaves the same as `begin_imp` except that `R0` need not to be set.

- `mid_exp_imp` $\mathtt{Y}_j$, $\mathtt{Y}_k$
  Behaves the same as `mid_exp_imp` except that the `s1` and `s2` fields of all records from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ need not be set.

- `end_exp_imp` $\mathtt{Y}_i$, $\mathtt{Y}_j$, $\mathtt{Y}_k$
  Behaves the same as `end_imp` except that the added resource entries need not be examined. Only set the `out_of_scope` flags of all records from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ to true (trailing so that they may be reset on backtracking), and set register `T` to $\mathtt{Y}_k \vee \mathtt{T}$.

5.3.4 Code for Resource Addition

Let us consider the addition of a resource clause $\forall x.(G -\circ A)$, which contains free variables $X_1, \ldots, X_m$, in which A is an atom with predicate symbol p/n. As mentioned in the previous chapter, this clause is compiled into the following code:

L: `unify_variable` $\mathtt{A}_{n+1}$
`unify_variable` $\mathtt{A}_{n+2}$
⋮
`unify_variable` $\mathtt{A}_{n+m}$
Code for the head A
Code for the body G

This code will be executed after the WAM register `S` is set to the top of the references to free variables, and the mode is set to `read`. Thus, each "`unify_variable` $\mathtt{A}_{n+i}$" instruction retrieves the value of free variable X_i.

The code generated for adding the clause by $-\circ$ is as follows (If the clause is added by $\Rightarrow$ rather than $-\circ$, the `add_res` instruction is replaced by `add_exp_res`):

`put_structure` p/n, $\mathtt{A}_i$
Code for the 1st argument
⋮
Code for the n*-th argument*
`put_closure` L, m, $\mathtt{A}_j$
Code for the free variable X_1
⋮
Code for the free variable X_m
`add_res` $\mathtt{A}_i$, $\mathtt{A}_j$

The following new instructions are used in the code generated for resource addition:

- `put_closure` L, m, $\mathtt{A}_i$

Create a closure structure on the heap. Set register $\texttt{A}_i$ to a new CLO cell pointing to the current top of the heap. Push L (code address) and m (the number of free variables) on the heap. Set mode to write. The unify_value (or unify_variable) instruction that follows this instruction, pushes the m references to free variables on the heap.

```
Ai := ⟨CLO, H⟩;
HEAP[H] := L;
H := H + 1;
HEAP[H] := m;
H := H + 1;
mode := write;
P := P + instruction_size(P);
```

- add_res $\texttt{A}_i$, $\texttt{A}_j$

Used when the implication operator is $\multimap$. Add a record for a (linear) resource clause of the form $\forall x.A$ or $\forall x.(G \multimap A)$ as a new entry at the top of the resource table, RES. The value of L is stored in the level field and the deadline field, the out_of_scope flag is set to false. $\texttt{A}_i$ and $\texttt{A}_j$ are pointers to structures previously built on heap holding the head and closure of the clause respectively.

```
RES[R].level := L;
RES[R].deadline := L;
RES[R].out_of_scope := false;
RES[R].head := Ai;
RES[R].body := undef;
RES[R].closure := Aj;
RES[R].pred := register_resource(Ai);
R := R + 1;
P := P + instruction_size(P);
```

- add_exp_res $\texttt{A}_i$, $\texttt{A}_j$

Used when the implication operator is $\Rightarrow$. Add a record for an (intuitionistic) resource clause of the form $\forall x.A$ or $\forall x.(G \multimap A)$ as a new entry at the top of the resource table, RES. Behaves the same as add_res, except that the level and deadline fields are set to zero.

```
RES[R].level := 0;
RES[R].deadline := 0;
RES[R].out_of_scope := false;
RES[R].head := Ai;
RES[R].body := undef;
RES[R].closure := Aj;
RES[R].pred := register_resource(Ai);
R := R + 1;
P := P + instruction_size(P);
```

In the add_res and add_exp_res instructions, the $register\_resource(\texttt{A}_i)$ function registers the value of R (the index of added clause) to the hash and symbol tables for speed access to the resources in RES. The return value, the index of the predicate symbol of $\texttt{A}_i$ in the symbol table, is set to the pred field. The head structure $\texttt{A}_i$ is used to calculate the hash value of added clause, since the entries are hashed on the predicate symbol and the first argument in current implementation.

Figure 5.3 shows the code generated for the goal $((p(1) \,\&\, (q(X) \multimap p(X))) \otimes \forall Y.r(Y)) \multimap G$, where the free variable X is stored in A1.

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

The goal $G_1 \,\&\, G_2$ requires careful coordination of the consumption of resources between the two conjuncts. The code generated for a conjunction of the form $G_1 \,\&\, G_2$ is, structurally, quite simple:

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

```
begin_imp Y3                    % code for p(1)
put_ground p(1), A2             L1:  get_integer 1, A1
put_closure L1, 0, A3                proceed
add_res A2, A3
put_structure p/1, A4           % code for q(X)⊸p(X)
unify_local_value A1            L2:  unify_variable A2
put_closure L2, 1, A5                get_value A2, A1
unify_local_value A1                 execute q/1
add_res A4, A5
mid_imp Y2, Y1                  % code for ∀Y.r(Y)
begin_imp Y6                    L3:  proceed
put_structure r/1, A6
unify_void 1
put_closure L3, 0, A7
add_res A6, A7
mid_imp Y5, Y4
Code for G
end_imp Y6, Y5, Y4
end_imp Y3, Y2, Y1
```

Figure 5.3: Code Generated for the Goal $((p(1)\,\&(q(X) \multimap p(X))) \otimes \forall Y.r(Y)) \multimap G$.

However, in order to faithfully reproduce the behavior described in the $\mathcal{LRM}$ for the $\&$ operator, the three new LLPAM instructions are, individually, more complex than those seen so far.

- `begin_with` $\mathtt{Y}_i$
  Decrement U so that we can tell which resources are consumed in G_1. Those resources will have the new value of U as the value of their `level` field. Store the current value of the T register in a new permanent variable $\mathtt{Y}_i$ and set the T register to false.

- `mid_with` $\mathtt{Y}_j$
  Perform $\overset{\textit{changepair}}{(\mathtt{U},0)\rightarrow(\mathtt{L}+1,\mathtt{L}+1)}$. This marks all of the resources that were used in the left conjunct (identified by having a `level` with the same value as register U and a `deadline` of 0) so that they can, and must, be used in the right conjunct (by setting both those fields to L+1, to which L will be set during the execution of the right conjunct). If the value of T is true, then perform $\overset{\textit{change}}{\mathtt{L}\rightarrow\mathtt{L}+1}$ so that resources that were available but not explicitly used, but which $\top$ can be thought of as having used, are also available for use in the right conjunct. Increment L and U. Store the current value of the T register to a new permanent variable $\mathtt{Y}_j$. Set the T register to false.

- `end_with` $\mathtt{Y}_i$, $\mathtt{Y}_j$
  Decrement register L. If the value of register T is true, then perform $\overset{\textit{changepair}}{(\mathtt{L}+1,\mathtt{L}+1)\rightarrow(\mathtt{U},0)}$ ($\top$ 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 $\textit{consumed}_{\mathtt{L}+1}$ to check whether all the resources that should have been consumed, were. If this fails, fail. Otherwise, if $\mathtt{Y}_j$ is true, then perform $\overset{\textit{change}}{\mathtt{L}+1\rightarrow\mathtt{L}}$ (Those resources that were made available to the right conjunct because the left conjunct included a $\top$, but weren't used in the right conjunct either, are put back to their original level). Set T to $\mathtt{Y}_i \vee (\mathtt{Y}_j \wedge \mathtt{T})$.

The latter two must examine or manipulate the `level` and `deadline` of all of the in-scope entries in the resource table. To find out them, it is clearly inefficient to scan all entries in the resource table. To solve

this problem, we use the register RLIST, which is always set to the list that contains the indices of linear resources (the values of R0). RLIST grows when resources are added by $-\circ$, and shrinks on backtracking. Thus, all we have to do is to check the elements of RLIST, rather than all of entries in the resource table.

5.3.6 Code for $!G$

Only intuitionistic resources can be used in the execution of $!G$. As in $\mathcal{LRM}$ the consumption level is manipulated to enforce this constraint. The code generated for $!G$ is as follows:

```
begin_bang Y_i
Code for G
end_bang Y_i
```

The following new instructions are used in the code generated for the goal $!G$:

- `begin_bang` $\texttt{Y}_i$
  Increment L (so that it is higher than the value of the `level` field in all the entries in RES. Now, only intuitionistic resources can be used during the proof of G). Store the value of T in a new permanent variable $\texttt{Y}_i$.
- `end_bang` $\texttt{Y}_i$
  Decrement L. Set the value of the register T from the variable $\texttt{Y}_i$ (since it does not matter whether a $\top$ was seen in G or not).

5.3.7 Code for $\top$

The use of $\top$ as a goal is compiled to the instruction:

- `top`
  Set the register T to true.

5.3.8 Code for Atomic Goals

An atomic goal means resource consumption or an ordinary program invocation. The execution of an atomic goal A with predicate symbol p/n proceeds as follows:

1. Extract the list of indices of the possibly consumable resource clauses in the resource table, RES, by referring to the hash and symbol table. 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 closure (compiled code followed by a variable bindings) of R.
3. After the failure of all trials, `call` the ordinary code for predicate A.

The step 1 are added between the point that a `call` (or `execute`) is issued, and the point that the code block for the predicate is entered.

Looking-up a resource is done through the hash table and the symbol table. In the current implementation, the entries are hashed on the predicate symbol and the first argument. However, we can not always

```
num_of_args := SYMBOL[@(P)].arity;
CP := P + instruction_size(P);
B0 := B;
if SYMBOL[@(P)].codeaddr = undef then
    backtrack;
if tag(SYMBOL[@(P)].res) ≠ LIS then
    begin R1 := []; R2 := [] end
else
    lookup_hash(@(P));
P := SYMBOL[@(P)].codeaddr;
```

Figure 5.4: The `call` Instruction of the LLPAM

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.

Figure 5.4 show the `call` instruction of the LLPAM where $@(P)$ stands for the index value of P in the symbol table. This instruction saves the current choice point's address `B` in `B0`, the value of current continuation in `CP`. If the predicate P is defined, then perform the $lookup\_hash$ function (described in Appendix A). The $lookup\_hash$ function extracts the list of indices of the possibly consumable resource clauses in the resource table by referring to the hash and symbol tables, and set them in `R1` and `R2` (Set [] if there are no consumable resources). When the goal has an unbound variable as the first argument, `R1` is set to the list of indices of all resources with that predicate name/arity (`Symbol[`$@(P)$`].res`), and `R2` is set to an empty list. Otherwise, `R1` is set to the list of indices of all resources with that predicate name/arity and the same first argument through the hash table, `R2` is set to the list of the indices of all resources which have that predicate name/arity, but the first argument was an unbound variable when the resource was added (`Symbol[`$@(P)$`].res2`).

For steps 2 and 3, the following new instructions are used:

- `try_resource` L
  Allocate a new choice point frame on the stack. Behaves the same as `try` L, except that `R1` and `R2` are also saved.

- `restore_resource`
  Having backtracked to the current choice point, reset all the necessary information from it.

- `retry_resource_else` L
  Update the next clause field (`BP`) to L. Update the `R1` and `R2` fields in the current choice point frame with their current values.

- `trust_resource` L
  Discard the current choice point frame by resetting `B` to its predecessor. Continue execution with the following instruction labeled L.

- `pickup_resource` p/n, $\mathtt{A}_i$, L
  Find an index of consumable resource with predicate symbol p/n from `R1` and `R2`. Set that index to $\mathtt{A}_i$. Continue execution with the following instruction. If there are no consumable resources, jump to the instruction labeled L.

```
p/n:  try_resource L1
L0:   restore_resource
L1:   pickup_resource p/n, An+1, L2
      retry_resource_else L0
      consume An+1, An+2
      execute_closure An+2
L2:   trust_resource L'

L':   an ordinary program code of p/n
```

Figure 5.5: Naive Code Generation for an Atomic Goal p/n

```
p/n:  pickup_resource p/n, An+1, L'
      try_resource L1
L0:   restore_resource
      pickup_resource p/n, An+1, L2
      retry_resource_else L0
L1:   consume An+1, An+2
      execute_closure An+2
L2:   trust_resource L'

L':   an ordinary program code of p/n
```

Figure 5.6: Code Generated for an Atomic Goal p/n

- `consume` $\mathtt{A}_i$, $\mathtt{A}_j$
  Mark the entry `RES[`$\mathtt{A}_i$`]` as consumed (set `level` to the current value of `U`, and `deadline` to 0). Set $\mathtt{A}_j$ to the value of `closure` field.

- `execute_closure` $\mathtt{A}_i$
  Save the current choice point `B` in `B0`. Let $\mathtt{A}_i$ be a closure structure $\langle \mathtt{CLO},\ c \rangle$. Set the register `S` to $c + 2$ pointing to the top of the references to free variables. Set mode to `read`. Continue execution with instruction on `HEAP[`c`]`.

Figure 5.5 shows a naive code generated for an atomic goal p/n. This code contains an obvious inefficiency. Even if there is not any consumable resource, it allocates a new choice point frame by the `try_resource` instruction.

Figure 5.6 shows an improved code, which 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 (labeled L') immediately.

In figure 5.6, the `pickup_resource` instruction checks whether there are any consumable resources or not. It first finds an index of consumable resource with p/n from `R1` and `R2`, and then sets that index to $\mathtt{A}_{n+1}$. `R1` and `R2` are updated to have the remaining resources. If there are no consumable resources, it quickly jumps to the ordinary program code labeled L'.

The `try_resource` instruction allocates a new choice point frame on the stack and continue execution with the instruction labeled L_1. It behaves the same as WAM instruction "`try` L_1", but `R1` and `R2` are also saved. It is noted that, in the LLPAM, the `try` instruction always saves the values of new registers `R`, `L`, `U`, and `T` in the choice point frame.

The `consume` instruction marks the resource `RES[`$\mathtt{A}_{n+1}$`]` as as consumed (by changing its `level` to the current value of `U`, and its `deadline` to 0), and sets its `closure` in $\mathtt{A}_{n+2}$.

Let $\mathtt{A}_{n+2}$ be $\langle$CLO, $c\rangle$. The `execute_closure` instruction first saves the current choice point `B` in `B0`. It then sets the register `S` to $c + 2$ pointing to the top of the references to free variables. It finally sets mode to `read` and continues execution with instruction on `HEAP[`c`]`.

Having backtracked to the current choice point labeled L_0, the `restore_resource` instruction resets all the necessary information from it, and the `pickup_resource` instruction is invoked again. If there are not any more consumable resources, it jumps to the instruction labeled L_2, and the `trust_resource` instruction discards the current choice point and jumps to the ordinary program code labeled L'. Otherwise, The `retry_resource_else` instruction updates the `R1` and `R2` fields in the current choice point frame with their current values and continues execution with the following instruction.

5.4 Backtracking

In order to be able to recover the correct state on backtracking, we need to take the following additional bookkeeping measures:

- 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.
- Changes to the register `RLIST` should be trailed.

5.5 Optimizing the Design

5.5.1 Optimizing Resource Selection

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. However, when there is no ordinary program invocation, an atomic goal means only resource consumption. Our optimization design 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 without creating a choice point.
- It is safe to discard the current choice point before consuming the last consumable resource.

The following new instruction is used in the optimized code generated for an atomic goal:

- `if_no_resource` L
  Scans whether there are any consumable resources in `R1` and `R2`. If there are no consumable resources, jump to the instruction labeled L.

Our optimization can be achieved quite easily by inserting the above new instruction just after the `pickup_resource` instructions. Figure 5.7 shows an optimized code corresponding to the Figure 5.6.

```
p/n:  pickup_resource p/n, A_{n+1}, fail
      if_no_resource L_1
      try_resource L_1
L_0:  restore_resource
      pickup_resource p/n, A_{n+1}, L_2
      if_no_resource L_3
      retry_resource_else L_0
L_1:  consume A_{n+1}, A_{n+2}
      execute_closure A_{n+2}
L_2:  trust_resource fail
L_3:  trust_resource L_1
```

Figure 5.7: Optimized Code Generation for an Atomic Goal p/n

5.5.2 Successive Addition of Linear Resources

The goal of the form $R_1 \multimap (R_2 \multimap \cdots (R_n \multimap G)\cdots)$ or $(R_1 \otimes R_2 \otimes \cdots \otimes R_n) \multimap G$ adds each individual R_i successively and then executes the goal G. Such goal is frequently used to add multiple resources in LLP programs. The code generated for the goal $R_1 \multimap (R_2 \multimap \cdots (R_n \multimap G)\cdots)$ is as follows:

```
begin_imp Y_{i_1}
Code for the addition of R_1
mid_imp Y_{j_1}, Y_{k_1}
begin_imp Y_{i_2}
Code for the addition of R_2
mid_imp Y_{j_2}, Y_{k_2}
      ⋮
begin_imp Y_{i_n}
Code for the addition of R_n
mid_imp Y_{j_n}, Y_{k_n}
Code for G
end_imp Y_{i_n}, Y_{j_n}, Y_{k_n}
      ⋮
end_imp Y_{i_2}, Y_{j_2}, Y_{k_2}
end_imp Y_{i_1}, Y_{j_1}, Y_{k_1}
```

However this code contains an obvious inefficiency. This code requires nested n `begin_imp`, `mid_imp` , and `end_imp` instructions respectively, and $3n$ permanent registers. In particular, each of the `end_imp` instructions needs to scan all of the in-scope entries in the resource table.

Our optimization design for solving this problem is as follows:

- It is possible to add whole resources without using nested instructions.
- Only one `end_imp` instruction is used to check whether all added resources are consumed or not.

The following new instruction is used in the optimized code generated for $R_1 \multimap (R_2 \multimap \cdots (R_n \multimap G)\cdots)$:

- `more_imp`
  Set the `s1` and `s2` fields of added resource clauses in R_i to the current values of `R0` (the index of first resource clause) and `R` (the top of the resource table), respectively. Update the value of `R0` with `R`.

```
begin_imp Y3                    % code for p(1)
put_ground p(1), A2             L1:  get_integer 1, A1
put_closure L1, 0, A3                proceed
add_res A2, A3
put_structure p/1, A4           % code for q(X)–∘p(X)
unify_local_value A1            L2:  unify_variable A2
put_closure L2, 1, A5                get_value A2, A1
unify_local_value A1                 execute q/1
add_res A4, A5
more_imp                        % code for ∀Y.r(Y)
put_structure r/1, A6           L3:  proceed
unify_void 1
put_closure L3, 0, A7
add_res A6, A7
mid_imp Y2, Y1
Code for G
end_imp Y3, Y2, Y1
```

Figure 5.8: Optimized Code Generated for the Goal $((p(1)\,\&(q(X) \multimap p(X))) \otimes \forall Y.r(Y)) \multimap G$.

Our idea can be achieved quite easily by using the above new instruction. The optimized code generated for the goal $R_1 \multimap (R_2 \multimap \cdots (R_n \multimap G)\cdots)$ is as follows:

```
begin_imp Y_i
Code for the addition of R_1
more_imp
Code for the addition of R_2
more_imp
    ⋮
more_imp
Code for the addition of R_n
mid_imp Y_j, Y_k
Code for G
end_imp Y_i, Y_j, Y_k
```

This optimized code requires only one `begin_imp`, `mid_imp` , and `end_imp` instructions respectively, and only 3 permanent registers. The $n-1$ successive use of `begin_imp` and `mid_imp` are replaced with $n-1$ `more_imp`. The n `end_imp` are replaced with only one `end_imp`.

Note that, such optimization can be applied to the goal $R_1 \Rightarrow (R_2 \Rightarrow \cdots (R_n \Rightarrow G)\cdots)$. However, in this case, we do not need the `more_imp` instruction since that goal is converted into $(R_1 \,\&\, R_2 \,\&\, \cdots \,\&\, R_n) \Rightarrow G$ in compilation time.

For example, Figure 5.8 shows the optimized code corresponding to the Figure 5.3.

5.6 LLPAM Code Example

Figure 5.9 shows the partial code generated for Lolli program of filtering a list in in Figure 3.3.

```
% choose(Xs,Y,Zs)
choose/3:
     allocate 3
     begin_exp_imp Y3
     put_structure test/1, A4  % creates test(X) in A4
     unify_void 1
     put_closure L, 1, A5      % creates a closure
     unify_local_value A2      % sets the free variable Y
     add_exp_res A4, A5        % adds the resource
     mid_exp_imp Y2 Y1
     put_value A3, A2
     call filter/2             % call filter(Xs,Zs)
     end_exp_imp Y3 Y2 Y1
     deallocate
     proceed

% forall(X, (test(X) :- X >Y))
L:   unify_variable A2         % retrieves the free variable Y
     execute '>'/2             % execute X > Y

test/1:
     pickup_resource test/1, A2, fail
     if_no_resource L1
     try_resource L1
L0:  restore_resource
     pickup_resource test/1, A2, L2
     if_no_resource L3
     retry_resource_else L0
L1:  consume A2, A3
     execute_closure A3
L2:  trust_resource fail
L3:  trust_resource L1
```

Figure 5.9: LLPAM Code Generated for the Predicate `choose` and the Resource `test` in Figure 3.3

5.7 Performance Evaluation of LLP Compiler System

We have developed a LLPAM-based compiler system, called LLP. In this section, we present the performances of the LLP compiler system.

LLP is a first generation compiler system for a linear logic programming language. The system consists of a LLP to LLPAM compiler (written in Prolog) and a emulator (written in C), but it does not incorporate well-known optimizations, register allocation, last-call-optimization, global analysis, and so on.

The newest package (version is 0.5.1) is available from:

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

First, we compare the execution speeds of two N-Queen programs. One is a prolog program in Figure 3.5 compiled under a SICStus Prolog 3.7.1 (WAM code), where resources are represented by list structures. Another is a Lolli program in Figure 3.4 compiled under LLP 0.5.1 (LLPAM code), where resources are compiled into closures and kept in the resource table.

Table 5.1 shows the performance results for all solutions of $8 \leq N \leq 14$. At $N = 8$, LLP is 2.67 times faster than Prolog, and the speedup of LLP is getting larger and larger as N increases. At $N = 14$, LLP is

Table 5.1: Performance Results of N-Queens

N	Runs Averaged	SICStus 3.7.1	LLP 0.5.1	Speedup Ratio
8	10	40	15	2.67
9	10	186	68	2.74
10	10	902	292	3.09
11	10	4762	1432	3.33
12	10	26849	7558	3.55
13	5	159856	42292	3.78
14	5	1013514	252846	4.01

Table 5.2: Performance Results of Knight Tour (5×5)

Runs Averaged	SICStus 3.7.1	LLP 0.5.1	Speedup Ratio
5	60416	25392	2.38

4 times faster than Prolog. Table 5.2 shows the performance results of similar test for Knight Tour program (all solutions) in Figure 3.7. LLP is 2.38 times faster than Prolog.

Second, we show an improvement in performance by compiling resources rather than storing resources as terms on heap memory. We compare the execution speeds of two Lolli programs for tiling board with dominoes, and both of them are compiled under LLP 0.5.1.

One is a program in Figure 3.11 where resources are compiled into closures. Another is the same program except that all occurrences of resources are replaced with metavariables as resources. For example, the code for adding resources:

```
gen_res(N) :- N > 0, !,
        (
            ( d(I, J, N), J1 is J+1, d(I, J1, N) ) -<> domino(N)
               &
            ( d(I, J, N), I1 is I+1, d(I1, J, N) ) -<> domino(N)
        )
        -<> (N1 is N-1, gen_res(N1)).
```

is replaced with

```
gen_res(N) :- N > 0, !,
        Domino = (( d(I, J, N), J1 is J+1, d(I, J1, N) ) -<> domino(N)
                    &
                  ( d(I, J, N), I1 is I+1, d(I1, J, N) ) -<> domino(N)),
        Domino -<> (N1 is N-1, gen_res(N1)).
```

where resources are not compiled and stored as terms.

Table 3.11 shows the performance results for all solutions on 5 different shaped boards. Compiling resources (LLP) is 1.5 times faster than representing resources as terms in a heap memory (LLP_T). The speedup is due to the compilation of resource clauses, in which their bodies consist of compound goal formulas, rather than only atoms.

To measure the overhead incurred by the new structures of the LLPAM, we compare our compiler with high performance Prolog compilers, SICStus Prolog version 3.7.1 and SWI-Prolog (with `-O` option) version 3.4.1. Table 5.4 shows the performance results of classical Prolog benchmarks. LLP is 1.47 times faster than SWI-Prolog, but twice slower than SICStus. When we take into consideration that the factor of slowdown

Table 5.3: Performance Results of Tiling Board with Dominoes

(Row, Column)	Runs Averaged	LLP_T	LLP	Speedup Ratio
(2, 5)	5	62	42	1.48
(3, 4)	5	698	460	1.52
(2, 7)	5	8474	5830	1.45
(4, 4)	5	145062	95934	1.51
(3, 6)	5	2167058	1457346	1.49

Table 5.4: Performance Results of Prolog Benchmarks

Prolog Programs	Runs Averaged	LLP 0.5.1	SWI 3.4.1	SICStus 3.7.1
`boyer`	5	1772	520	162
`browse`	5	414	650	210
`cal`	5	78	100	44
`chat_parser`	5	8	14	6
`ham`	5	262	380	156
`poly_10`	5	18	34	10
`queens_10` (all sol.)	5	680	1060	330
`tak`	5	60	148	26
`zebra`	5	18	18	10
Average of Ratio	5	1.00	1.47	0.51

from SICStus seems to be due to the difference between a optimized Prolog compiler and our relative naive compiler, new structures incur only a sufficiently small overhead.

All times in Table 5.1–5.4 were collected on Linux system (Pentium III 850MHz, 128M memory).

5.8 Performance Evaluation of Hodas and Tamura's lolliCoP

Now we shows the performance of a more sophisticated application of lolliCoP in Figure 3.13. The results (Table 5.5–5.7) are taken from a paper [28] by Hodas and Tamura.

They tested lolliCoP on the 2200 clausal form problems in the TPTP library version 2.3.0 [48, 34]. TPTP consist of 2193 problems known to be unsatisfiable (or valid using positive representation) and 7 propositional problems known to be satisfiable (or invalid). Each problem is rated from 0.00 to 1.00 relative to its difficulty. A rating of "?" means the difficulty is unknown. No reordering of clauses or literals has been done.

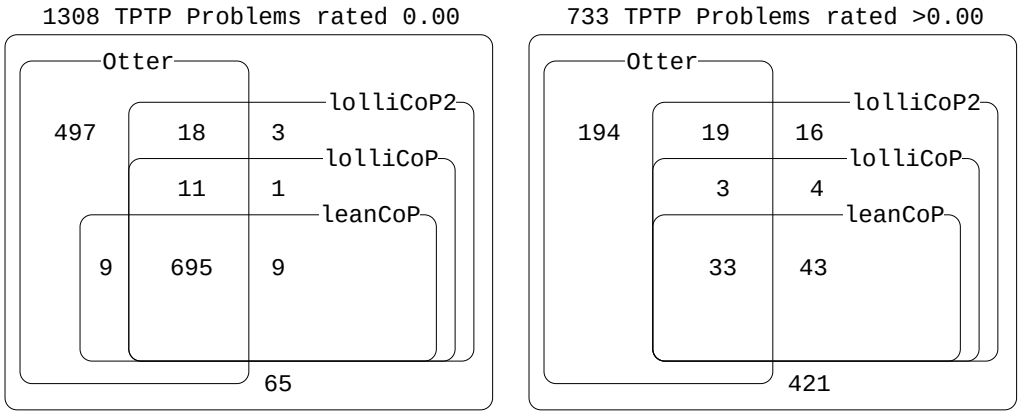
The tests were performed on a Linux system with a 550MHz Pentium III processor and 128M bytes of memory. The programs were compiled with LLP compiler version 0.5.0. The time limit for all proof attempts was 300 seconds.

The overall performance of OTTER 3.1 (with MACE 1.4) [34, 36], leanCoP [46], and lolliCoP, in terms of the number of problems solved, are shown in Table 5.5. The table also includes data for an improved version oflolliCoP, called lolliCoP$_2$. The results for leanCoP were obtained in the same environment as those for lolliCoP, using SICStus Prolog 3.7.1, and are better than those reported by the authors [46]. The results for OTTER 3.1 (with MACE 1.4), which is not publicly available, are taken from a report by its developers [34]. These results were produced on a 400MHz Pentium II, which is somewhat slower than the

Table 5.5: Overall Performance of OTTER, leanCoP, and lolliCoP

	Total	OTTER		leanCoP		lolliCoP		lolliCoP$_2$	
Solved	2200	1602	(73%)	810	(37%)	822	(37%)	880	(40%)
0 to $<$ 1 second		1209		541		554		614	
1 to $<$ 10 seconds		142		135		124		117	
10 to $<$100 seconds		209		93		91		94	
100 to $<$200 seconds		31		18		25		34	
200 to $<$300 seconds		11		23		28		21	
Problems rated 0.00	1308	1230	(94%)	713	(55%)	716	(55%)	737	(56%)
Problems rated $>$0.00	733	249	(34%)	76	(10%)	83	(11%)	118	(16%)
Problems rated ?	159	123	(77%)	21	(13%)	23	(14%)	25	(16%)

Table 5.6: Performance of OTTER leanCoP and lolliCoP Classified by Problem Rating



machine Hodas and Tamura used. Figure 5.6 depicts the overlap of problems solved by each system.

Table 5.7a compares the performance of all four systems on the 33 problems that they can all solve. Total CPU time is shown, along with a speedup ratio relative to leanCoP (under SICStus). On just these problems, lolliCoP provides a speedup of 40% over leanCoP, and it has almost the same performance as OTTER. However, comparing the result of 36 problems solved by both OTTER and lolliCoP, OTTER is 71% faster as shown in Table 5.7b. Finally, Table 5.7c shows a similar analysis for the 76 problems that lolliCoP and leanCoP can both solve.

Table 5.7: Comparison of OTTER, leanCoP, and lolliCoP

(a) 33 problems solved by OTTER, leanCoP, and lolliCoP

	OTTER	leanCoP	lolliCoP	lolliCoP$_2$
Total CPU time	1143.03	1590.66	1139.41	338.47
Average CPU time	34.64	48.20	34.53	10.26
Speedup Ratio	1.39	1.00	1.40	4.70

(b) 36 problems solved by OTTER and lolliCoP

	OTTER	lolliCoP	lolliCoP$_2$
Total CPU time	1152.40	1969.67	450.57
Average CPU time	32.01	54.71	12.52
Speedup Ratio	1.71	1.00	4.37

(c) 76 problems solved by leanCoP and lolliCoP

	leanCoP	lolliCoP	lolliCoP$_2$
Total CPU time	2757.83	2038.58	853.24
Average CPU time	36.29	26.82	11.23
Speedup Ratio	1.00	1.35	3.23

Chapter 6

Translating a Linear Logic Programming Language into Java

In recent years, a number of Java implementations for logic programming languages have been developed: CKI Prolog, JavaLog, Jinni [53], JIP, JLog, JP, jProlog [18], Kernel Prolog, KLIJava, LL, LLPj, MINERVA, NetProlog, tuProlog, and W-Prolog [57]. However, there has been no Java implementation for linear logic programming languages.

In this chapter, we present the Prolog Café system, that translates LLP into Java via the LLPAM. The system has the advantages of portability, extensibility, and interactivity with Java. It is portable to any platform supporting a Java compiler. It is easily expandable with several extensions using a lot of Java class libraries, such as multi-threaded and distributed system. It also provides smooth bi-directional interaction between LLP and Java.

There seem to be at least three approaches to implement efficient LLP system in Java.

1. Compiling LLP into Java Bytecodes
2. LLPAM emulator in Java
3. Translating LLP into Java

The approach (1) might be the fastest, but compiling to Java bytecodes is certainly not a simple task and is sensitive to Java version dependency. The approach (2) might be the simplest, but emulation has not nice performances if unoptimized and is not possible to produce an independent executable program as output. Thus, we decided to investigate the approach (3): translating LLP into Java. This has the advantage of giving nice speedup in performance, avoiding the overhead of emulators, and producing stand-alone executable codes.

First, we describe how jProlog, LLPj, and Prolog Café translate Prolog control flow into Java. After that we describe how Prolog Café translates LLP into Java.

6.1 Demoen and Tarau's jProlog Approach

The jProlog system, developed by Demoen and Tarau, is a first generation Prolog-to-Java translator via the WAM. It is based on binarization transformation [54], a continuation passing style compilation technique used in BinProlog.

First, each Prolog clause is translated into a binary clause by binarization, and then translated into Java code.

Each term is translated into a certain Java object. Each clause is translated into one Java class. Each predicate is translated a set of classes; there is one class for entry point, and other classes for clauses. Each continuation goal is translated at term level, that is executed by referring to the hash table to transform it into its corresponding predicate object.

Let us consider a simple example:

```
p :- q, r.
q.
```

First, each clause is translated into a binary clause:

```
p(Cont) :- q(r(Cont)).
q(Cont) :- call(Cont).
```

and then each of them is translated into the following Java code, where some codes are omitted to retrieve the essence of control flow:

```
public class pred_p_0 extends Code {     % entry point of p/0
    static Code cl1 = new pred_p_0_1() ;
    static Code q1cont ;

    void init(PrologMachine mach) {
    q1cont = mach.LoadPred("q",0) ;      % get continuation goal q/0
    }

    Code Exec(PrologMachine mach) {
    return cl1.Exec(mach) ;              % call the clause p(Cont) :- q(r(Cont))
    }
}

class pred_p_0_1 extends pred_p_0 {      % p(Cont) :- q(r(Cont)).
    Code Exec(PrologMachine mach) {
    PrologObject continuation = mach.Areg[0];                % get Cont
    mach.Areg[0] = new Funct("r".intern(), continuation) ;   % create r(Cont)
    mach.CUTB = mach.CurrentChoice ;
    return q1cont ;                                          % call q/0
    }
}

public class pred_q_0 extends Code {     % entry point of q/0
    static Code cl1 = new pred_q_0_1() ;
    Code Exec(PrologMachine mach) {
    return cl1.Exec(mach) ;              % call the clause q(Cont) :- call(Cont).
    }
}

class pred_q_0_1 extends pred_q_0 {      % q(Cont) :- call(Cont).
    Code Exec(PrologMachine mach) {
    mach.CUTB = mach.CurrentChoice ;
    return UpperPrologMachine.Call1 ;    % call Cont
    }
}
```

Translated code is executed in the following supervisor function:

```
code = code for target predicate;
while (ExceptionRaised == 0) {
     code = code.Exec(this) ;      % this indicates Prolog engine
}
```

jProlog is a good starting point to study of translating Prolog into Java. jProlog supports intuitionistic assumption, backtrackable destructive assignment, and delayed execution. However, jProlog is an experimental implementation, that does not incorporate well-known optimizations such as indexing and specialization of head unification, and so on. It is this system that we optimize and extend for linear logic programming language.

The newest package of jProlog (version is 0.1) is available from:

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

The system consists of a Prolog to Java translator (written in Prolog) and a run-time system (written in Java).

6.2 The LLPj Approach

The LLPj system [8] is a first generation, LLP-to-Java translator via LLPAM. This system was also based on binarization, but it took a different approach from jProlog for translating Prolog into Java.

In this approach, each predicate is translated into only one class, in which each clause is translated into a single Java method. Each continuation goal is translated at predicate level rather than term level, that is directly executed by invoking its `exec` method. The previous example is translated as follows:

```
public class PRED_p_0 extends Predicate {
    public PRED_p_0 ( Predicate cont) {
        this.cont = cont;                       % get Cont
    }

    public void exec() {
        if(clause1()) return;                   % call the clause p(Cont) :- q(r(Cont))
    }

    private boolean clause1() {                 % p(Cont) :- q(r(Cont)).
        try {
            Predicate new_cont = new PRED_r_0(cont);   % create r(Cont)
            (new PRED_q_0(new_cont)).exec();           % call q/0
        } catch (CutException e) {
            if(e.id != this) throw e;
            return true;
        }
        return false;
    }
}

public class PRED_q_0 extends Predicate {
    public PRED_q_0 ( Predicate cont) {
        this.cont = cont;                       % get Cont
    }

    public void exec() {
        if(clause1()) return;                   % call the clause q(Cont) :- call(Cont).
    }

    private boolean clause1() {                 % q(Cont) :- call(Cont).
        try {
            cont.exec();                        % call Cont
        } catch (CutException e) {
            if(e.id != this) throw e;
            return true;
        }
        return false;
    }
}
```

Translated code is executed without a supervisor function:

```
code = code for target predicate;
code.exec();
```

In this approach, we do not have to care the choice point stack. The trail stack is maintained in each predicate locally. The cut mechanism is easily implemented by Java exception handling functions: `try` and `catch`. In performance, our translator generated slight faster code for some Prolog benchmarks compared with jProlog.

The main drawback of this approach is that the invocation of `exec` will invoke other nested `exec` methods, and never return until the system reaches the first solution. This leads to a memory overflow for large programs. As with jProlog, the system did not incorporate well-known optimizations such as indexing, specialization of unification, and so on. In addition, LLPj support the LLP language, but resources are not compiled and stored at term level in the resource table. This slows down the execution speed of resources.

6.3 The Prolog Café Approach

The Prolog Café system is a refinement of LLPj. The main differences from LLPj is as follows:

- As with jProlog, each predicate is translated into a set of classes, but each continuation goal is translated at predicate level, which is executed by a supervisor function to avoid memory overflows in LLPj.
- It is possible to treat Java objects as Prolog terms, invoke their methods, and access to their fields using reflection based Java interface: `java_constructor/2`, `java_method/3`, `java_set_field/3`, and `java_get_field/3`.
- Prolog Café incorporates the optimization of indexing(only first level) and specialization of head unification, built-in predicates for error and exception handling, and floating point numbers.
- To execute resources efficiently, resource are compiled into *closures* which consist of a reference of compiled code and a set of bindings for free variables.

6.3.1 Translating Prolog into Java

Each term is translated into a Java object of classes in Figure 6.1: `VariableTerm`, `IntegerTerm`, `DoubleTerm`, `SymbolTerm`, `ListTerm`, and `StructureTerm`. The `Term` class, with an abstract method `unify`, is a common superclass of these classes.

The `JavaObjectTerm` class is used to treat Java objects as terms. The `ClosureTerm` class is used to create closure structures for compiling resources in LLP.

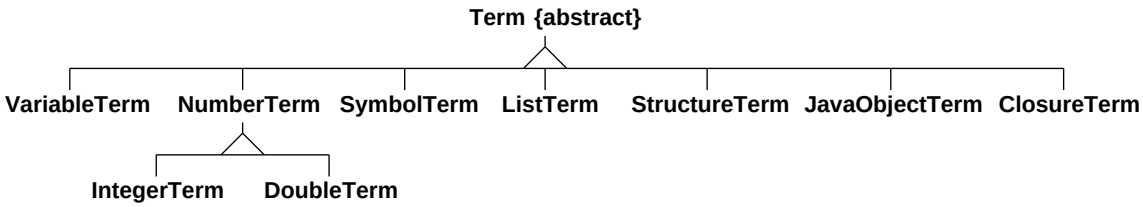


Figure 6.1: Term Structure of Prolog Café

Each predicate is translated a set of classes; there is one class for entry point, and other classes for clauses and choice instructions. The `Predicate` class is a common superclass of these classes, and has an abstract method `exec` and the field `cont` for continuation goal. Each continuation goal is translated at predicate level, that is is executed by a supervisor function. The previous example is translated as follows:

```
import jp.ac.kobe_u.cs.prolog.lang.*;

public class PRED_p_0 extends Predicate {
    public PRED_p_0(Predicate cont) {
        this.cont = cont;                       % get Cont
    }

    public Predicate exec() {                   % p(Cont) :- q(r(Cont)).
        engine.setB0();
        Predicate p1 = new PRED_r_0(cont);      % create r(Cont)
        return new PRED_q_0(p1);                % call q/0
    }
}

public class PRED_q_0 extends Predicate {
    public PRED_q_0(Predicate cont) {
        this.cont = cont;                       % get Cont
    }

    public Predicate exec() {                   % q(Cont) :- call(Cont).
        return cont;                            % call Cont
    }
}
```

Translated code is executed by a supervisor function:

```
code = code for target predicate;
while (code == null) {
    code.setEngine(engine);             % engine indicates Prolog engine
    code = code.exec();
}
```

This translation method is an integration of jProlog and LLPj approaches. In performance, our translator generated 1.7 times faster code for a set of classical Prolog benchmarks compared with jProlog. This speedup is entirely almost due to the optimization of indexing and specialization of head unification.

In Prolog Café, it is possible to treat Java objects as Prolog terms, invoke their methods, and access to their fields using reflection based Java interface.

The predicate `java_constructor/2` is used to create a Java object as term, called *Java term*. Java term is implemented as an instance of the `JavaObjectTerm` class. The predicate `java_method/3` is used to invoke the methods of Java terms. The predicate `java_get_field/3` is used to get the values of specified fields of Java terms. The predicate `java_set_field/3` is used to set the values to specified fields of Java terms.

```
main :-
      java_constructor('java.awt.Frame', X),
      java_method(X, setSize(200,200), _),
      java_get_field('java.lang.Boolean', 'TRUE', True),
      java_method(X, setVisible(True), _).
```

For example, the above code will display an empty Java frame on your display.

```
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).
```

Figure 6.2: An Implementation of `assert` and `retract` in Prolog Café

6.3.2 Implementing `assert` and `retract`

Prolog Café is easily expandable with increasing Java class libraries since all data structures are represented as Java objects. Here, we present an implementation of Prolog's `assert` and `retract` by using Java hash table. In our design, each entry in the hash table contains a list of clauses. The entry is hashed on the predicate name/arity of the head part of clause.

The following built-in predicates for handling a hash table are used to implement `assert` and `retract` easily.

- `put_term(+Hash, +Key, ?Term)`
  Maps the key to the value of `Term` in the hash table. `Key` is a ground term for the hash key. Note that any unbound variables in `Term` are not replaced by new private variables.

- `get_term(+Hash, +Key, ?Term)`
  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` must be a Java term which has a hash table on the inside. Since such Java terms can be created by `java_constructor/2`, it is possible to maintain multiple hash tables in Prolog. Figure 6.2 shows source code for `assert` and `retract`, where `user` represents the standard hash table in Prolog Café. 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.

6.3.3 Translating LLP into Java

We have presented the Prolog aspect of the Prolog Café system so far. We now describe how Prolog Café translates LLP into Java.

First, each clause including linear logic operators is translated into a Prolog clause, in which those operators are replaced with built-in predicates that correspond to certain LLPAM instructions. After that it is binarized and then translated into Java code.

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

For example, the above clauses are translated into:

```
p(X, Y) :-                          p(X, Y) :-          p(X, Y) :-
    begin_imp(A),                       begin_with(A),      begin_bang(A),
    add_res(q(X), [q/1,[X]]),           q(X),               q(X),
    mid_imp(B, C),                      mid_with(B),        r(Y),
    r(Y),                               r(Y),               end_bang(A).
    end_imp(A, B, C).                   end_with(A, B).
```

in which each built-in predicate (written by bold face) corresponds to certain LLPAM instruction. The second argument `[q/1,[X]]` of `add_res` is used to create the closure structure for the resource `q(X)`. After that these clauses are translated into binary clauses and then translated into Java code.

Let us show the partial code generated for the first clause "`p(X, Y) :- q(X) -<> r(Y)`".

```
import jp.ac.kobe_u.cs.prolog.lang.*;
public class PRED_p_2 extends Predicate {            % Code for the predicate p/2
    static Predicate res_q_1 = new RES_q_1();
    static SymbolTerm sym_q_1 = SymbolTerm.makeSymbol("q", 1);

    public PRED_p_2(Term a1, Term a2, Predicate cont) {
        arg1 = a1;
        arg2 = a2;
        this.cont = cont;
    }

    public Predicate exec() {
        engine.setB0();
        a1 = arg1.dereference();                      % Get the value of 1st argument X
        a2 = arg2.dereference();                      % Get the value of 2nd argument Y

        x  = {a1};
        a4 = new StructureTerm(sym_q_1, x);           % Create the head q(X)
        a5 = new ClosureTerm(res_q_1, x);             % Create the closure of q(X)
        p1 = new PRED_end_imp_2(a3, a6, a7, cont);
        p2 = new PRED_r_1(a2, p1);
        p3 = new PRED_mid_imp_1(a6, a7, p2);
        p4 = new PRED_add_res_2(a4, a5, p3);          % Add the resource q(X)
        return new PRED_begin_imp_1(a3, p4);
    }
}

public class RES_q_1 extends Predicate {              % Code for the resource q/1
    public Predicate exec() {
        a1 = engine.aregs[1].dereference();           % Get the value of 1st argument
        a2 = engine.aregs[2].dereference();           % Get the value of free variable X
        this.cont = engine.cont;

        if (! a1.unify(a2, engine.trail))             % Unify the 1st argument with X
            return engine.fail();                     % backtrack
        return cont;
    }
}
```

Table 6.1: Comparison for Prolog Café vs jProlog vs SWI-Prolog

Prolog Programs	Runs Averaged	Prolog Café 0.5.0	jProlog 0.1	SWI 3.4.1
`boyer`	5	6600.6	13995.8	634.0
`browse`	5	1866.4	16129.4	654.0
`ham`	5	1929.4	3391.8	384.0
`nrev`(300 elem.)	5	243.8	1520.8	48.0
`query`	5	64.6	46.2	4.0
`tak`	5	2327.2	2963.0	11180.0
`zebra`	5	75.2	127.6	18.0
`cal`	5	492.4	?	192.0
`chat_parser`	5	226.8	?	18.0
`poly_10`	5	450.4	?	38.0
`queens_10` (all sol.)	5	2348.0	?	1892.0
`sendmore`	5	236.0	?	72.0
Average of Ratio	5	1.00	1.71	0.16

The `engine` indicates an Prolog Café engine which is activated currently. The `engine.aregs` and `engine.cont` fields indicate the argument registers and the continuation register respectively. Closure structures can be implemented easily by using the `ClosureTerm` class.

6.4 Performance Evaluation

We now present the performances of the Prolog Café system. The system consists of two Java packages:

- `jp.ac.kobe_u.cs.prolog.lang` for runtime system,
- `jp.ac.kobe_u.cs.prolog.compiler` for translator.

We compare Prolog Café with jProlog version 0.1 and SWI-Prolog version 3.4.1. jProlog is a first generation, Prolog-to-Java translator system. SWI-Prolog is a popular Prolog compiler system, that compile Prolog into WAM.

Table 6.1 shows the performance results of a set of classical Prolog benchmarks. A time of "?" means we met some errors during the compilation of generated Java code using Java compiler. All times in Table 6.1 were collected on Linux system (Pentium III 850MHz, 128M memory). with Java$^{\mathrm{TM}}$ 2 SDK Standard Edition version 1.4.0.

Prolog Café generates 1.7 times faster code than jProlog. This speedup is entirely almost due to indexing and specialization of head unification. Compared with SWI-Prolog, Prolog Café is 6.3 times slower.

Prolog Café is a first generation, Java implementation for a linear logic programming language. It does not incorporate well-known optimizations, such as register allocation, last-call-optimization, global analysis, and so on. The newest package (version is 0.5.0) is available from:

`http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/`

Chapter 7

TLLP: A Temporal Linear Logic Programming Language

Recent development of logic programming languages based on linear logic suggests a successful direction to extend logic programming to be more expressive and more efficient. The treatment of formulas-as-resources gives us not only powerful expressiveness, but also efficient access to a large set of data. However, in linear logic, whole resources are kept in one context, and there is no straight way to represent complex data structures as resources. For example, in order to represent an ordered list and time-dependent data, we need to put additional indices for each resource formula.

Temporal Linear Logic (TLL) is an extension of linear logic with some features of temporal logic. TLL was first studied by Kanovich and Itoh [31], and a cut-free sequent system has been proposed by Hirai [23]. The semantics model of TLL consists an infinite number of phase spaces linearly ordered by the time clock. Each phase space is the same as that of linear logic.

In this chapter, we describes a logic programming language, called TLLP, based on intuitionistic temporal linear logic. This logic, an extension of linear logic with some features from temporal logics, allows the use of the modal operators '$\bigcirc$'(next-time) and '$\Box$'(always) in addition to the operators used in intuitionistic linear logic. The intuitive meaning of modal operators is as follows: $\bigcirc\, B$ means that B can be used exactly once at the next moment in time; $\Box\, B$ means that B can be used exactly once any time; $!\, B$ means that B can be used arbitrarily many times (including 0 times) at any time.

We first give a proof theoretic formulation of the logic of the TLLP language. We then present a series of resource management systems designed to implement not only interpreters but also compilers based on an extension of the standard WAM model. Finally, we describe some implementation methods based on our systems.

7.1 Intuitionistic Temporal Linear Logic

In this section, we will focus on the sequent system *ITLL* [23] of intuitionistic temporal linear logic developed by Hirai. The expressive power of *ITLL* is shown by a natural encoding of Timed Petri Net. It is this logic that we shall use to design and implement the logic programming language described below.

ITLL allows the use of the modal operators '$\bigcirc$'(next-time) and '$\Box$'(always) in addition to the operators used in intuitionistic linear logic. Compared with the sequent system *ILL* (see Figure 2.1) of intuitionistic linear logic, three rules (L $\Box$), (R $\Box$), and ($\bigcirc$) are added. The entire set of *ITLL* sequent rules is given in Figure 7.1. Here, the left-hand side of sequents are multisets of formulas, and the structural rule for exchange need not be explicitly stated. The structural rule for weakening (W !) and contraction (C !) are available only for assumptions marked with the modal operator '!'. This means that, in general, formulas not !-marked can

(Rules of *ILL* in Figure 2.1)

$$\frac{\Delta, B \longrightarrow C}{\Delta, \Box B \longrightarrow C}\ (\mathrm{L}\,\Box) \qquad \frac{!\,\Gamma, \Box\,\Sigma \longrightarrow C}{!\,\Gamma, \Box\,\Sigma \longrightarrow \Box C}\ (\mathrm{R}\,\Box) \qquad \frac{!\,\Gamma, \Box\,\Sigma, \Delta \longrightarrow C}{!\,\Gamma, \Box\,\Sigma, \bigcirc\Delta \longrightarrow \bigcirc C}\ (\bigcirc)$$

Figure 7.1: The Proof System *ITLL* for Intuitionistic Temporal Linear Logic

be used exactly once. Limited-use formulas can represent time-dependent resources in *ITLL*. The intuitive meaning of these modal operators is as follows:

- $\bigcirc B$ means that B can be used exactly once at the next moment in time.
- $\Box B$ means that B can be used exactly once any time.
- $!\,B$ means that B can be used arbitrarily many times (including 0 times) at any time.

By combining these modalities with binary operators in linear logic, several resources can be expressed. For example, $B \,\&\, \bigcirc B$ means that B can be used exactly once either at the present time or at the next moment in time. $\bigcirc(1 \,\&\, B)$ means that B can be used at most once at the next moment in time.

Two formulas B and C are equivalent, denoted $B \equiv C$, if the sequents $B \longrightarrow C$ and $C \longrightarrow B$ are provable in *ITLL*. The notation $\bigcirc^n$ means n multiplicity of $\bigcirc$. We note the following sequents that are provable in *ITLL*.

$$!\,B \equiv !!\,B, \qquad \Box B \equiv \Box\Box B, \qquad !\,B \equiv \Box\,!\,B,$$
$$!\,B \longrightarrow \Box B \otimes \cdots \otimes \Box B, \qquad \Box B \longrightarrow \bigcirc^n B \quad (n \geq 0)$$

The main differences from other temporal linear logic systems [31][50] are that *ITLL* includes the modal operator '!', and it satisfies a cut elimination theorem. Both of these additions are very important for the design of a language based on the notion of *Uniform Proofs*.

7.2 Language Design

The idea of uniform proofs [38], proposed by Miller et. al, is a simple and powerful notion for designing logic programming languages. Uniform proof search is a cut-free, *goal-directed proof search* in which a sequent $\Gamma \longrightarrow G$ denotes the state of the computation trying to solve the goal G from the program Γ. Goal-directed proof search is characterized operationally by the bottom-up construction of proofs in which right-introduction rules are applied first and left-introduction rules are applied only when the right-hand side is atomic. This means that the operators in the goal G are executed independently from the program Γ, and the program is only considered when its goal is atomic. A logical system is an *abstract logic programming language* if restricting it to uniform proofs retains completeness. The logics of Prolog, λProlog, and Lolli are examples of abstract logic programming language.

Clearly, intuitionistic linear logic (even over the connectives: $\top$, $\&$, $\otimes$, $\multimap$, !, and $\forall$) is not an abstract logic programming language. For example, the sequents $a \otimes b \longrightarrow b \otimes a$ and $!\,a \,\&\, b \longrightarrow !\,a$ are both provable in *ILL* but do not have uniform proofs.

Hodas and Miller have designed the linear logic programming language Lolli [25][26] by restricting formulas so that the above counterexamples do not appear, although it retains desirable features of linear logic connectives such as ! and $\otimes$. The Lolli language is based on the following fragment of linear logic:

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

(Rules of $\mathcal{L}$ in Figure 2.4)

$$\frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta, \Box\, B \longrightarrow C}\ (\mathrm{L}\,\Box) \qquad \frac{\Gamma; \Box\,\Sigma, \Delta \longrightarrow C}{\Gamma; \Box\,\Sigma, \bigcirc\,\Delta \longrightarrow \bigcirc\, C}\ (\bigcirc)$$

Figure 7.2: $\mathcal{TL}$: A Proof System for the Connectives $\top$, $\&$, $\multimap$, $\Rightarrow$, $\forall$, 1, !, $\otimes$, $\oplus$, $\exists$, $\bigcirc$, and $\Box$.

Here, R-formulas and G-formula are called *resource* and *goal formulas* respectively. The connective $\Rightarrow$ is called *intuitionistic implication*, and it is defined as $B \Rightarrow C \equiv (!\,B) \multimap C$.

The sequent of Lolli is of the form $\Gamma; \Delta \longrightarrow G$ where Γ is a set of resource formulas, Δ is a multiset of resource formulas, and G is a goal formula. Γ and Δ are called *intuitionistic* and *linear context* respectively, and they correspond to the *program*. G is called the *goal*. The sequent $\Gamma; \Delta \longrightarrow G$ can be mapped to the linear logic sequent $!\,\Gamma, \Delta \longrightarrow G$. Thus, the right introduction rule for $\multimap$ adds its assumption (called a *linear resource*) to the linear context, in which every formula can be used exactly once. The right introduction rule for $\Rightarrow$ adds its assumption (called an *intuitionistic resource*) to the intuitionistic context, in which every formula can be used arbitrarily many times (including 0 times).

Hodas and Miller developed a series of proof systems $\mathcal{L}$ (see Figure 2.4) and $\mathcal{L}'$ in [25]. They proved that $\mathcal{L}$ is sound and complete with respect to the *ILL* rules restricted to the Lolli language. They also proved $\mathcal{L}$ preserves completeness even if provability is restricted to uniform proofs. $\mathcal{L}'$ is the proof system that results from replacing the Identity, L$\multimap$, L$\Rightarrow$, L$\&$, and L$\forall$ rules in $\mathcal{L}$ with a single rule, called *backchaining*.

In this chapter, we will use a more restrictive definition for resource and goal formulas. Let A be atomic and $m \geq 1$:

$$\begin{array}{lcl} R & ::= & S_1 \,\&\, \cdots \,\&\, S_m \\ S & ::= & \top \mid A \mid G \multimap A \mid \forall x.S \\ G & ::= & 1 \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid G_1 \oplus G_2 \mid R \multimap G \mid S \Rightarrow G \mid !\,G \mid \forall x.G \mid \exists x.G \end{array}$$

Here, S-formulas are called *resource clauses* in which A and G are called the *head* and the *body* respectively. S-formulas correspond to program clauses. Although this simplification does not change expressiveness of the language, it makes the presentation of *backchaining* simpler, as is discussed below.

Since full intuitionistic linear logic is not an abstract logic programming language, it is obvious that intuitionistic temporal linear logic is not as well. For example, in addition to the counterexamples in *ILL*, the sequents $\Box \bigcirc a \longrightarrow \bigcirc a$, $! \bigcirc a \longrightarrow \bigcirc a$, and $a \,\&\, \bigcirc a \longrightarrow \bigcirc a$ are all provable in *ITLL*, but they do not have uniform proofs.

Figure 7.2 presents a proof system $\mathcal{TL}$ for the connectives $\top$, $\&$, $\multimap$, $\Rightarrow$, $\forall$, 1, !, $\otimes$, $\oplus$, $\exists$, $\bigcirc$, and $\Box$. Two rules, L$\Box$ and $\bigcirc$, are added in addition to those that arise in $\mathcal{L}$. This system has been designed to support the logic programming language TLLP over the following formulas: If A is atomic and $m \geq 1$,

$$\begin{array}{lcl} R & ::= & S_1 \,\&\, \cdots \,\&\, S_m \mid \Box(S_1 \,\&\, \cdots \,\&\, S_m) \mid \bigcirc R \\ S & ::= & \top \mid A \mid G \multimap A \mid \forall x.S \\ G & ::= & 1 \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid G_1 \oplus G_2 \mid R \multimap G \mid S \Rightarrow G \mid !\,G \mid \forall x.G \mid \exists x.G \mid \bigcirc G \end{array}$$

Let D be a $\&$-product of resource clauses $S_1 \,\&\, \cdots \,\&\, S_m$. Compared with Lolli, $\bigcirc^n D$ and $\bigcirc^n \Box D$ are added to resource formulas, and $\bigcirc G$ is added to goal formulas. The intuitive meaning of these formulas is as follows: $\bigcirc^n D$ means that the resource clause S_i $(1 \leq i \leq m)$ in D can be used exactly once at time n; $\bigcirc^n \Box D$ means that the resource clause S_i $(1 \leq i \leq m)$ in D can be used exactly once any time at and after time n; $\bigcirc G$ adjusts time one clock ahead and then executes G.

The proofs of propositions in this chapter are based on Hodas and Miller's results in [25] for the Lolli language, and we will only give proof outlines.

$$\frac{}{\Gamma; D \longrightarrow A}\ (\mathrm{BC}_1) \qquad \frac{}{\Gamma, D; \emptyset \longrightarrow A}\ (\mathrm{BC!}_1)$$

provided, in each case, A is atomic and $A \in \|D\|$.

$$\frac{\Gamma; \Delta \longrightarrow G}{\Gamma; \Delta, D \longrightarrow A}\ (\mathrm{BC}_2) \qquad \frac{\Gamma, D; \Delta \longrightarrow G}{\Gamma, D; \Delta \longrightarrow A}\ (\mathrm{BC!}_2)$$

provided, in each case, A is atomic and $G \multimap A \in \|D\|$.

Figure 7.3: Backchaining for the Proof System $\mathcal{TL}'$

Proposition 7.2.1 Let G be a goal formula, Γ a set of resource clauses, and Δ a multiset of resource formulas. Let D^* be the result of replacing all occurrences of $B \Rightarrow C$ in D with $(!\,B) \multimap C$, and let $\Gamma^* = \{B^* \mid B \in \Gamma\}$. Then the sequent $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{TL}$ if and only if $!(\Gamma^*), \Delta^* \longrightarrow G^*$ is provable in *ITLL*.

Proof [sketch] The proof of this proposition can be shown by giving a simple conversion between proofs in the two systems. The cases of $\bigcirc$ and L$\Box$ are also immediate. □

Proposition 7.2.2 Let G be a goal formula, Γ a set of resource clauses, and Δ a multiset of resource formulas. Then the sequent $\Gamma; \Delta \longrightarrow G$ has a proof in $\mathcal{TL}$ if and only if it has a uniform proof in $\mathcal{TL}$.

Proof [sketch] The proof in the reverse direction is immediate, since a uniform proof in $\mathcal{TL}$ is a proof in $\mathcal{TL}$. The forward direction can be proved by showing that any proof in $\mathcal{TL}$ can be converted to a uniform proof of the same endsequent by permuting the rules to move occurrences of the left-rule up, though, and above instances of the right-rule. We explicitly show one case, that is when L$\Box$ occurs below R$\&$:

$$\frac{\dfrac{\begin{matrix}\Xi_1 \\ \Gamma; \Delta, B \longrightarrow C_1\end{matrix} \quad \begin{matrix}\Xi_2 \\ \Gamma; \Delta, B \longrightarrow C_2\end{matrix}}{\Gamma; \Delta, B \longrightarrow C_1 \,\&\, C_2}\ (\mathrm{R}\,\&)}{\Gamma; \Delta, \Box\, B \longrightarrow C_1 \,\&\, C_2}\ (\mathrm{L}\,\Box)$$

where Ξ_1 and Ξ_2 are uniform proofs of their endsequents respectively. The above proof structure can be converted to the following:

$$\frac{\dfrac{\begin{matrix}\Xi_1 \\ \Gamma; \Delta, B \longrightarrow C_1\end{matrix}}{\Gamma; \Delta, \Box\, B \longrightarrow C_1}\ (\mathrm{L}\,\Box) \quad \dfrac{\begin{matrix}\Xi_2 \\ \Gamma; \Delta, B \longrightarrow C_2\end{matrix}}{\Gamma; \Delta, \Box\, B \longrightarrow C_2}\ (\mathrm{L}\,\Box)}{\Gamma; \Delta, \Box\, B \longrightarrow C_1 \,\&\, C_2}\ (\mathrm{R}\,\&)$$

□

As with $\mathcal{L}$ and $\mathcal{L}'$, the left-hand rules can be restricted to a form of backchaining. Let us consider the following definition: Let R be a resource formula. $\|R\|$ is defined as a set of resource clauses (S-formulas):

1. if $R = A$ then $\|R\| = \{A\}$,
2. if $R = G \multimap A$ then $\|R\| = \{G \multimap A\}$,
3. if $R = \forall x.S$ then for all closed terms t, $\|R\| = \|S[t/x]\|$,
4. if $R = S_1 \,\&\, \cdots \,\&\, S_m$ then $\|R\| = \|S_1\| \cup \cdots \cup \|S_m\|$,
5. if $R = \Box\, R'$ then $\|R\| = \|R'\|$,
6. if $R = \bigcirc\, R'$ then $\|R\| = \emptyset$.

Let $\mathcal{TL}'$ be a proof system that results from replacing the Identity, absorb, L$\multimap$, L$\Rightarrow$, L$\&$, L$\forall$, and L$\Box$ rules in $\mathcal{TL}$ with the backchaining rules in Figure 7.3. These backchaining rules (especially the definition of $\|\cdot\|$) are simpler than the original rule for Lolli because of the restrictive definition of resource formulas. It is noticed that the absorb rule is integrated into (BC!$_1$) and (BC!$_2$).

Proposition 7.2.3 Let G be a goal formula, Γ a set of resource clauses, and Δ a multiset of resource formulas. Then the sequent $\Gamma; \Delta \longrightarrow G$ has a proof in $\mathcal{TL}$ if and only if it has a proof in $\mathcal{TL}'$.

Since uniform proofs are complete for $\mathcal{TL}$, this proposition can be proved by showing that there is a uniform proof in $\mathcal{TL}$ if and only if there is a proof in $\mathcal{TL}'$. We do not present the proof here. A similar proof has been given by Hodas and Miller in [25] for the Lolli language.

7.3 TLLP Programming Examples

We new present simple TLLP examples. For the syntax, we use '`@`' for $\bigcirc$ and '`#`' for $\Box$.

7.3.1 Path Finding

We first consider a Lolli program that finds a Hamilton path through the complete graph of four vertices. Since each vertex is represented as a linear resource, the constraints such that each vertex must be used exactly can be expressed.

```
p(V,V,[V]) :- v(V).
p(U,V,[U|P]) :- v(U), e(U,W), p(W,V,P).
e(U,V).
goal(P) :- v(a) -<> v(b) -<> v(c) -<> v(d) -<> p(a,d,P).
```

When the goal `goal(P)` is executed, the vertices are added as resources, and the goal `p(a,d,P)` will search a path from `a` to `d` by consuming each vertex exactly once.

In addition to the resource-sensitive features of Lolli, TLLP can describe the time-dependent properties of resources, in particular, the precise order of the moments when some resources are consumed. For example, `#v(a)` denotes the vertex a that can be used exactly once at and after present. `@ #v(c)` denotes the vertex c that can be used exactly once at and after the next moment in time.

```
p(V,V,[V]) :- v(V).
p(U,V,[U|P]) :- v(U), e(U,W), @p(W,V,P).
e(U,V).
goal(P) :- #v(a) -<> @ @v(b) -<> @ #v(c) -<> #v(d) -<> p(a,d,P).
```

So, the above program finds a Hamilton path that satisfies such constraints. It is noticed that time is adjusted one clock ahead every time the path crosses an arc.

7.3.2 Conway's Life Game

TLLP is suitable to write programs in which the dynamical state changes with depending on time. In Figure 7.4, we show a TLLP program of Conway's Life Game, but the code for output is omitted. The resource `b(I,J)` means that there is a life on `(I,J)` at present. The predicate `next(I,J)` checks whether a new life will be born on `(I,J)` at the next moment in time. If this succeeds, the resource `@b(I,J)` is added. It is noted that the double negation `\+ \+` is used to execute `next(I,J)` without consuming any resources.

```
life :- N = 20,
        b(1, 2) -<> b(2, 3) -<> b(3, 1) -<> b(3, 2) -<> b(3, 3) -<>
        n(N) => loop.
loop :- loop(1, 1).
loop(I, J) :- n(N), I > N, !, @loop.
loop(I, J) :- n(N), J > N, !, I1 is I+1, loop(I1, 1).
loop(I, J) :- \+ \+ next(I, J), !, J1 is J+1, @b(I, J) -<> loop(I, J1).
loop(I, J) :- J1 is J+1, loop(I, J1).
next(I, J) :- b(I, J), !, count(I, J, C), 2 =< C, C =< 3.
next(I, J) :- count(I, J, C), C = 3.
count(I1, J1, C) :-
        I0 is I1-1, I2 is I1+1, J0 is J1-1, J2 is J1+1,
        count_b([(I0,J0),(I0,J1),(I0,J2),
                 (I1,J0),        (I1,J2),
                 (I2,J0),(I2,J1),(I2,J2)], C).
count_b([],          0) :- !.
count_b([(I,J)|IJs], C) :- b(I, J), !, count_b(IJs, C1), C is C1+1.
count_b([(I,J)|IJs], C) :- count_b(IJs, C).
```

Figure 7.4: A TLLP Example of Conway's Life Game

7.3.3 Timed Petri Net

Our next example is a simple Timed Petri Net reachability emulator. Figure 7.5 shows the program that checks the reachability of a Timed Petri Net from the initial marking (one token in p) to the final marking (one token in p and two tokens in r). Each d_i, a non-negative integer, is the delay time for the transition t_i.

Since the proof search of TLLP is depth-first and is not complete, we use a *iterative deepening* search, a combination of depth-fisrt and breadth-first search. First, the predicate `tpn(Dep, Lim)` checks the reachability at depth 1, and then it increases the depth by one if the check fails.

7.4 Resource Management Model

The resource management during a proof search in $\mathcal{TL}'$ is a serious problem for the implementor. Let us consider, for example, the execution of the goal $G_1 \otimes G_2$:

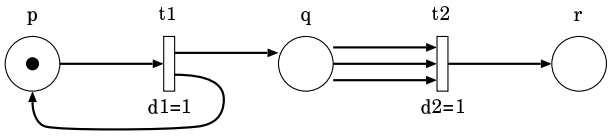
$$\frac{\Gamma; \Delta_1 \longrightarrow G_1 \quad \Gamma; \Delta_2 \longrightarrow G_2}{\Gamma; \underbrace{\Delta_1, \Delta_2}_{\Delta} \longrightarrow G_1 \otimes G_2}\ \mathrm{R}\otimes$$

When the system applies this rule during bottom-up search, the linear context Δ must be divided into Δ_1 and Δ_2. If Δ contains n resource formulas, all 2^n possibilities might need to be tested to find a desirable partition.

For Lolli, Hodas and Miller solved this problem by splitting resources lazily, and they proposed a new execution model called the $\mathcal{I}/\mathcal{O}$ model [26].

In this model, the sequent $I\,\{G\}\,O$ means that the goal G can be executed given the *input context* I so that the *output context* O remains. The input and output context, together called *IO*-context, are lists of resource formulas, !-marked resource formulas, or the special symbol 1 that denotes a place where a resource formula has been consumed. In the the execution of the goal $G_1 \otimes G_2$:

$$\frac{I\,\{G_1\}\,M \quad M\,\{G_2\}\,O}{I\,\{G_1 \otimes G_2\}\,O}\ (\otimes)$$




```
tpn :- #p -<> (goal :- p, r, r) => tpn(1, 100).

tpn(Dep, Lim) :- Dep =< Lim, fire(Dep).
tpn(Dep, Lim) :- Dep =< Lim, Dep1 is Dep + 1, tpn(Dep1, Lim).

next(D) :- D1 is D - 1, D1 > 0, fire(D1).

fire(D) :- goal.
fire(D) :- p, @ #p -<> @ #q -<> next(D).
fire(D) :- q, q, q, @ #r -<> next(D).
fire(D) :- @next(D).
```

Figure 7.5: A TLLP example of Timed Petri Net

First, $I\,\{G_1\}\,M$ tries to execute G_1 given the input context I. If this succeeds, the output context M is forwarded to G_2, and then $M\,\{G_2\}\,O$ is attempted. If this second attempt fails, $I\,\{G_1\}\,M$ retries to find a different, more desirable consumption pattern.

We will extend the $\mathcal{I}/\mathcal{O}$ model for the TLLP language. The additional problem here is that the bottom-up application of the rule for $\bigcirc$ in $\mathcal{TL}'$ requires manipulating large dynamic data structures.

$$\frac{\Gamma; \Box\,\Sigma, \Delta \longrightarrow G}{\Gamma; \Box\,\Sigma, \bigcirc\,\Delta \longrightarrow \bigcirc\,G}\ (\bigcirc)$$

For example, when the system executes the goal $\bigcirc\,G$ given input context $I = [p, \bigcirc\,q, \bigcirc\bigcirc\,r, !\,s]$, we need to reconstruct and create a new input context $I' = [1, q, \bigcirc\,r, !\,s]$ before the execution of the goal G.

We introduce a *time index* to solve this problem. Figure 7.6 presents an extension of the $\mathcal{I}/\mathcal{O}$ model for the TLLP language, called $\mathcal{IOT}$. $\mathcal{IOT}$ makes use of a time index T. The sequent is of the form $I\,\{G\}_T\,O$. T, non-negative integer, is the *current time*. At a given point in the proof, only resources that can be used at that time may be used. T is also used to set a *consumption time* of newly added resources.

Each element in $\mathcal{IOT}$-context is a pair $\langle R, t\rangle$ where R is a resource formula or !-marked resource formula, and t is its consumption time, or the special symbol 1. Linear resources have the form $\langle S_1\,\&\,\ldots\,\&\,S_m, t\rangle$ or $\langle\Box(S_1\,\&\,\ldots\,\&\,S_m), t\rangle$, where t is its consumption time calculated from the value of T, and its multiplicity of $\bigcirc$. Intuitionistic resources have the form $\langle !\,S, 0\rangle$, where S is a resource clause. For example, the consumable resources at time T have the following forms in the context: $\langle S_1\,\&\,\ldots\,\&\,S_m, T\rangle$, $\langle\Box(S_1\,\&\,\ldots\,\&\,S_m), t\rangle$ where $t \leq T$, and $\langle !\,S, 0\rangle$

The relation *pick*$R_T(I, O, S)$ holds if S occurs in the context I and is consumable at time T, and O results from replacing that occurrence of S in I with 1. The relation also holds if $!S$ occurs in I, and I and O are equal. The relation *subcontext*$_T(O, I)$ holds if O arises from replacing arbitrarily many (including 0) non-!-marked elements of I that are consumable any time at and after time T with 1.

To prove that $\mathcal{IOT}$ is logically equivalent to $\mathcal{TL}'$, we need to define the notion of difference $I -_T O$ for two $\mathcal{IOT}$-context I and O that satisfy the relation *subcontext*$_T(O, I)$. $I -_T O$ is a pair $\langle\Gamma, \Delta\rangle$, where Γ is a set of all formulas S such that $\langle !\,S, 0\rangle$ is an element of I (and O), and Δ is a multiset of all formulas $\bigcirc^{\max(0,t-T)}\,R$ such that $\langle R, t\rangle$ occur in I (If R is of the form $S_1\,\&\,\cdots\,\&\,S_m$, then $t \geq T$. If R is of the form $\Box(S_1\,\&\,\cdots\,\&\,S_m)$, then t is arbitrary), and the corresponding place in O is the symbol 1.

$$\frac{}{I\,\{1\}_T\,I}\ (1) \qquad \frac{\mathit{subcontext}_T(O,I)}{I\,\{\top\}_T\,O}\ (\top)$$

$$\frac{I\,\{G_1\}_T\,M \quad M\,\{G_2\}_T\,O}{I\,\{G_1 \otimes G_2\}_T\,O}\ (\otimes) \qquad \frac{I\,\{G_1\}_T\,O \quad I\,\{G_2\}_T\,O}{I\,\{G_1 \,\&\, G_2\}_T\,O}\ (\&)$$

$$\frac{I\,\{G_i\}_T\,O}{I\,\{G_1 \oplus G_2\}_T\,O}\ (\oplus) \qquad \frac{[\langle !\,S, 0\rangle \mid I]\,\{G\}_T\,[\langle !\,S, 0\rangle \mid O]}{I\,\{S \Rightarrow G\}_T\,O}\ (\Rightarrow)$$

$$\frac{[\langle R, T+n\rangle \mid I]\,\{G\}_T\,[1 \mid O]}{I\,\{\bigcirc^n R \multimap G\}_T\,O}\ (\multimap)$$

provided that R is a formula of the form: $S_1 \,\&\, \cdots \,\&\, S_m$ or $\Box(S_1 \,\&\, \cdots \,\&\, S_m)$.

$$\frac{I\,\{G\}_T\,I}{I\,\{!\,G\}_T\,I}\ (!) \qquad \frac{I\,\{G\}_{T+1}\,O}{I\,\{\bigcirc\,G\}_T\,O}\ (\bigcirc)$$

$$\frac{\mathit{pickR}_T(I,O,A)}{I\,\{A\}_T\,O}\ (\mathrm{BC}_1) \qquad \frac{\mathit{pickR}_T(I,M,G \multimap A) \quad M\,\{G\}_T\,O}{I\,\{A\}_T\,O}\ (\mathrm{BC}_2)$$

Figure 7.6: $\mathcal{IOT}$: An $\mathcal{I}/\mathcal{O}$ Model for Propositional TLLP

Proposition 7.4.1 Let T be a non-negative integer. Let I and O be $\mathcal{IOT}$-contexts that satisfy $\mathit{subcontext}_T(O, I)$. Let $I -_T O$ be the pair $\langle \Gamma, \Delta\rangle$ and let G be a goal formula. $I\,\{G\}_T\,O$ is provable in $\mathcal{IOT}$ if and only if $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{TL}'$.

Proof [sketch] This proposition, in both directions, can be proved by induction on proof structure. □

7.5 Level-Based Resource Management Model

The $\mathcal{I}/\mathcal{O}$ model provides an efficient computation model for proof search. The $\mathcal{I}/\mathcal{O}$ model has been refined several times. Cervesato et. al recently have proposed a refinement designed to eliminate the nondeterminism in management of linear context involving $\&$ and $\top$ [13]. However, the $\mathcal{I}/\mathcal{O}$ model and its refinements still require copying and scanning large dynamic data structures to control the consumption of linear resources. Thus, they are more suited to develop interpreters in high-level languages rather than compilers.

We point out two problems here. First, during the execution of $I\,\{G\}\,O$ (especially *pickR*), the context O is reconstructed from the context I by replacing linear consumed resources with 1. This will slow down the execution speed. Secondly, let us consider the execution of the goal $G_1 \,\&\, G_2$:

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

This rule means that the goal G_1 and G_2 must use the same resources. In a naive implementation, the system first copies the input context and executes the two conjuncts separately, and then it compares their output contexts. This leads to unnecessary backtracking.

To solve these problems, Tamura et. al have introduced a refinement of the $\mathcal{I}/\mathcal{O}$ model with *level indices* [30][49], called the $\mathcal{IOL}$ model [1]. Hodas et. al recently proposed the refinement of $\mathcal{IOL}$ for the complete treatment of $\top$ in [29].

[1] In this dissertation, we use the notation in [30] to explain the $\mathcal{IOL}$ model.

$$\frac{}{\vdash^T_{L,U} I\,\{1\}\,I}\ (1) \qquad \frac{\mathit{subcontext}^T_{U,L}(O,I)}{\vdash^T_{L,U} I\,\{\top\}\,O}\ (\top)$$

$$\frac{\vdash^T_{L,U} I\,\{G_1\}\,M \quad \vdash^T_{L,U} M\,\{G_2\}\,O}{\vdash^T_{L,U} I\,\{G_1 \otimes G_2\}\,O}\ (\otimes)$$

$$\frac{\vdash^T_{L,U-1} I\,\{G_1\}\,M \quad \mathit{change}_{U-1,L+1}(M,N) \quad \vdash^T_{L+1,U} N\,\{G_2\}\,O \quad \mathit{thinable}_{L+1}(O)}{\vdash^T_{L,U} I\,\{G_1 \,\&\, G_2\}\,O}\ (\&)$$

$$\frac{\vdash^T_{L,U} I\,\{G_i\}\,O}{\vdash^T_{L,U} I\,\{G_1 \oplus G_2\}\,O}\ (\oplus_i) \qquad \frac{\vdash^T_{L,U} [\langle S,0,0\rangle \mid I]\,\{G\}\,[\langle S,0,0\rangle \mid O]}{\vdash^T_{L,U} I\,\{S \Rightarrow G\}\,O}\ (\Rightarrow)$$

$$\frac{\vdash^T_{L,U} [\langle R,T+n,L\rangle \mid I]\,\{G\}\,[\langle R,T+n,U\rangle \mid O]}{\vdash^T_{L,U} I\,\{\bigcirc^n R \multimap G\}\,O}\ (\multimap)$$

provided that R is a formula of the form: $S_1 \,\&\, \cdots \,\&\, S_m$ or $\Box(S_1 \,\&\, \cdots \,\&\, S_m)$.

$$\frac{\vdash^T_{L+1,U} I\,\{G\}\,O}{\vdash^T_{L,U} I\,\{!\,G\}\,O}\ (!) \qquad \frac{\vdash^{T+1}_{L,U} I\,\{G\}\,O}{\vdash^T_{L,U} I\,\{\bigcirc\,G\}\,O}\ (\bigcirc)$$

$$\frac{\mathit{pickR}^T_{L,U}(I,O,A)}{\vdash^T_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_1) \qquad \frac{\mathit{pickR}^T_{L,U}(I,M,G \multimap A) \quad \vdash^T_{L,U} M\,\{G\}\,O}{\vdash^T_{L,U} I\,\{A\}\,O}\ (\mathrm{BC}_2)$$

Figure 7.7: $\mathcal{IOTL}$: A Level-Based $\mathcal{I}/\mathcal{O}$ Model for Propositional TLLP

$\mathcal{IOL}$ makes use of two level indices L and U to manage the consumption of resources. The sequent is of the form $\vdash_{L,U} I\,\{G\}\,O$. L, a positive integer, is the *current consumption level*. At a given point in the proof, only linear resources labeled with that consumption level (and intuitionistic resources labeled with 0) can be used. U, a negative integer, is the *current consumption maker*. When a linear resource is consumed, its consumption level is changed to the value of U.

Each element in $\mathcal{IOL}$-context is a pair $\langle R,\ell\rangle$, where R is a resource formula, and ℓ is its consumption level. Linear resources have the form $\langle R,\ell\rangle$, where ℓ is the value of L at which the resource can be consumed. Intuitionistic resources have the form $\langle S,0\rangle$ where S is a resource clause.

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

For example, the outline of the execution of the goal $G_1 \,\&\, G_2$ is as follows:

1. $\vdash_{L,U-1} I\,\{G_1\}\,M$ Decrement U so that we know which resources are consumed during the execution of G_1, and then execute G_1.
2. $\mathit{change}_{U-1,L+1}(M,N)$ Change the level of resources that have been consumed in G_1 to $L+1$.
3. $\vdash_{L+1,U} N\,\{G_2\}\,O$ Increment L and U, and then execute G_2.
4. $\mathit{thinable}_{L+1}(O)$ Check whether none of resources in O have $L+1$ as their consumption level.

$\mathcal{IOL}$ is logically equivalent to $\mathcal{L}'$. In $\mathcal{IOL}$, all resources are kept in a single table, called *resource table*, during execution. The consumption of resources can be achieved easily by changing their consumption level destructively. The idea of this model has already been used as a basis for a compiler system for a useful fragment of first-order Lolli, in which the resource table is implemented as an array, and the speed access to resources is achieved by using a hash table.

For TLLP, we give a refinement of $\mathcal{IOT}$, called $\mathcal{IOTL}$ in Figure 7.7, with level indices of $\mathcal{IOL}$. The sequent of $\mathcal{IOTL}$ is of the form $\vdash^T_{L,U} I\,\{G\}\,O$, where T is the current time, L is the current consumption level, and U is the current consumption maker.

Each element in $\mathcal{IOTL}$-contexts is a tuple $\langle R, t, \ell \rangle$, where R is a resource formula, t is its consumption time, and ℓ is its consumption level. Linear resources have the form $\langle S_1 \,\&\, \ldots \,\&\, S_m, t, \ell \rangle$ or $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t, \ell \rangle$, where t is calculated from the value of T and its multiplicity of $\bigcirc$, and ℓ is the value of L at which the resource can be consumed. Intuitionistic resources have the form $\langle S, 0, 0 \rangle$, where S is a resource clause.

When the system executes $\vdash^T_{L,U} I \,\{G\}\, O$, the consumable resources in the context I have the following forms: $\langle S_1 \,\&\, \ldots \,\&\, S_m, T, L \rangle$, $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t, L \rangle$ where $t \leq T$, and $\langle S, 0, 0 \rangle$.

The relation *pickR*$^T_{L,U}(I, M, S)$ selects a consumable resource clause S from the input context I. The output context M is the same as I, except that the consumption level of the selected clause is changed to the value of U if it is a linear resource. The relation *change*$_{\ell,\ell'}(M, N)$ modifies the context M so that any resources in M with level ℓ have their level changed to ℓ' in the context N. The relation *thinable*$_\ell(O)$ checks whether none of resources in O have ℓ as their consumption level. The relation *subcontext*$^T_{U,L}(O, I)$ then consumes some resources. The output context O is the same as I, except that the consumption levels of some resources are changed to the value of U, if they are linear resources.

We will not explain the rules in detail here, but $\mathcal{IOTL}$ is logically equivalent to $\mathcal{TL}'$.

7.6 Implementation Design

In this section, we discuss implementation issues for the TLLP language.

7.6.1 TLLP Interpreter

It is easy to implement a TLLP Interpreter based on the $\mathcal{IOT}$ model in Prolog (see Figure 7.8). This interpreter is good at splitting resources lazily, but they are managed by list structure. This slow down the execution speed.

7.6.2 Translating TLLP into LLP

It is possible to translate TLLP programs into LLP programs by adding a new argument for the current time T of $\mathcal{IOT}$ to each predicate. Translated code is compiled into LLPAM code and efficiently executed under LLP compiler system.

The goal G not including the form of $R \multimap G'$ and $S \Rightarrow G'$ is easily translated into LLP's goal by using the following transformation $G[T]$, where T indicates the current time:

$$\begin{aligned}
1[T] &= 1 \\
\top[T] &= \top \\
p(\vec{x})[T] &= p(\vec{x}, T) \\
(G_1 \otimes G_2)[T] &= G_1[T] \otimes G_2[T] \\
(G_1 \,\&\, G_2)[T] &= G_1[T] \,\&\, G_2[T] \\
(G_1 \oplus G_2)[T] &= G_1[T] \oplus G_2[T] \\
(\bigcirc G)[T] &= G[T+1]
\end{aligned}$$

The goal G of the form R-<>G' and S=>G' is translated into LLP's goal as follows:

$$\begin{aligned}
(\bigcirc^n (S_1 \,\&\, \cdots \,\&\, S_m) \multimap G)[T] &= (S_1\{T+n\}_1 \,\&\, \cdots \,\&\, S_m\{T+n\}_1) \multimap G[T] \\
(\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m) \multimap G)[T] &= (S_1\{T+n\}_2 \,\&\, \cdots \,\&\, S_m\{T+n\}_2) \multimap G[T] \\
(S \Rightarrow G)[T] &= S\{T\}_3 \Rightarrow G[T]
\end{aligned}$$

```
:- op(1060, xfy, (&)).
:- op( 950, xfy, [-<>, =>]).
:- op( 900,  fy, [!, @, #]).

prove(G) :- prove(G, 0, [], []).

prove(    true, _T, I, I) :- !.
prove(    erase, T, I, O) :- !, subcontext(T, O, I).
prove( (G1, G2), T, I, O) :- !, prove(G1, T, I, M), prove(G2, T, M, O).
prove((G1 & G2), T, I, O) :- !, prove(G1, T, I, O), prove(G2, T, I, O).
prove((G1 ; G2), T, I, O) :- !, (prove(G1, T, I, O) ; prove(G2, T, I, O)).
prove((R -<> G), T, I, O) :- !,
        count_next(R, N, R1), T1 is T + N, prove(G, T, [(R1,T1)|I], [1|O]).
prove( (S => G), T, I, O) :- !, prove(G, T, [(!S,0)|I], [(!S,0)|O]).
prove(       !G, T, I, I) :- !, prove(G, T, I, I).
prove(       @G, T, I, O) :- !, T1 is T + 1, prove(G, T1, I, O).
prove(        A, T, I, O) :- pick(T, I, O, A).
prove(        A, T, I, O) :- pick(T, I, M, (G -<> A)), prove(G, T, M, O).

count_next(@R, N, R1) :- !, count_next(R, N1, R1), N is N1 + 1.
count_next( R, 0,  R).

pick( T, I, O,         S) :- pick1(T, I, O, S).
pick(_T, I, I,         S) :- rule(S).
pick(_T, I, I, (G -<> A)) :- rule((A :- G)).

pick1(_T,  [(!S,0)|I], [(!S,0)|I], S).
pick1( T, [(#R,T0)|I], [1|I],      S) :- T >= T0, select(R, S).
pick1( T,   [(R,T)|I], [1|I],      S) :-
        \+(R = (!_)), \+(R = (#_)), select(R, S).
pick1( T,       [R|I], [R|O],      S) :-  pick1(T, I, O, S).

select((R1 & R2), R) :- !, (select(R1, R) ; select(R2, R)).
select(R, R).

subcontext(_T,         [],          []).
subcontext( T, [(!S,0)|O],  [(!S,0)|I]) :- subcontext(T, O, I).
subcontext( T,     [R1|O], [(#R,T0)|I]) :-
        (R1 = (#R,T0) ; R1 = 1), subcontext(T, O, I).
subcontext( T,     [R1|O],  [(R,T0)|I]) :-
        \+(R = (!_)), \+(R = (#_)),
        T0 >= T,
        (R1 = (R,T0) ; R1 = 1),
        subcontext(T, O, I).
subcontext(T, [(R,T0)|O], [(R,T0)|I]) :- subcontext(T, O, I).

rule(( p(V,V,[V]) :- v(V) )).
rule(( p(U,V,[U|P]) :- v(U), e(U,W), @p(W,V,P) )).
rule(( e(_U,_V) )).
rule(( goal(P) :- #v(a) -<> @ @v(b) -<> @ #v(c) -<> #v(d) -<> p(a,d,P) )).
```

Figure 7.8: A $\mathcal{IOT}$ Model-Based TLLP Interpreter in Prolog

```
tpn :- tpn(0).

tpn(A) :- (forall B\ B >= A -<> p(B)) -<>
          (forall C\ (p(C),r(C),r(C)) -<> goal(C)) => tpn(1, 100, A).

tpn(A,B,C) :- A =< B, fire(A, C).
tpn(A,B,C) :- A =< B, D is A+1, tpn(D, B, C).

next(A, B) :- C is A-1, C > 0, fire(C, B).

fire(A, B) :- goal(B).
fire(A, B) :- p(B),
              (forall C\ C >= B+1 -<> p(C)) -<>
              (forall D\ D >= B+1 -<> q(D)) -<> next(A, B).
fire(A, B) :- q(B), q(B), q(B),
              (forall C\ C >= B+1 -<> r(C)) -<> next(A, B).
fire(A, B) :- C is B+1, next(A, C).
```

Figure 7.9: Translating a TLLP Example of Timed Petri Net into LLP

The $S\{t\}_1$ transformation is defined as follows since the resource S can be consumed at time t:

$$\begin{aligned} p(\vec{x})\{t\}_1 &= p(\vec{x}, t) \\ (G \multimap p(\vec{x}))\{t\}_1 &= G[t] \multimap p(\vec{x}, t) \\ (\forall x.S)\{t\}_1 &= \forall x.S\{t\}_1 \end{aligned}$$

The $S\{t\}_2$ transformation is defined as follows since the resource S can be consumed any time at and after time t:

$$\begin{aligned} p(\vec{x})\{t\}_2 &= \forall t'.(t' \geq t \multimap p(\vec{x}, t')) \\ (G \multimap p(\vec{X}))\{t\}_2 &= \forall t'.((t' \geq t \otimes G[t']) \multimap p(\vec{x}, t')) \\ (\forall x.S)\{t\}_2 &= \forall x.S\{t\}_2 \end{aligned}$$

The $S\{t\}_3$ transformation is defined as follows since the resource S can be consumed at any time:

$$\begin{aligned} p(\vec{x})\{t\}_3 &= \forall t'.p(\vec{x}, t') \\ (G \multimap p(\vec{x}))\{t\}_3 &= \forall t'.(G[t'] \multimap p(\vec{x}, t')) \\ (\forall x.S)\{t\}_3 &= \forall x.S\{t\}_3 \end{aligned}$$

Figure 7.9 shows the translated LLP code for a TLLP example of Timed Petri Net in Figure 7.5.

The drawback of this approach is that the rule $\top[T] = \top$, translating TLLP's $\top$ into LLP's $\top$, is logically incomplete. The goal $\top$ in TLLP consumes some of consumable resources any time at and after present. Thus, let T be the current time, it can not consume the resources with consumption level $t < T$. However LLP's $\top$ might consume those resources since it can not check their consumption time.

7.6.3 TLLPAM: An Extension of LLPAM for the TLLP language

We extend the LLPAM for the TLLP language here. Our extension is summarized as follows:

- Two new fields `time` and `box` is added to each entry in `RES`. The `time` field denotes the consumption time in $\mathcal{IOTL}$. The `box` flag is set to false if the newly added resource is not prefixed by $\Box$, otherwise true.

Table 7.1: Performance Results of Timed Petri Net

Runs Averaged	LLP 0.5.1	TLLP 0.1.3	Speedup Ratio
5	1330	776	1.71

- A new register `TI` is added. `TI` denotes the current time T in $\mathcal{IOTL}$. The value of this register must be recorded in each choice point frame regardless of whether TLLP programs make use of the resource management features or not. `TI` is used to set the `time` field of newly added resource. `TI` is also used for hash key for speed access to the resources.

- In the LLPAM, the instruction "`add_res` $\mathtt{A}_i$, $\mathtt{A}_j$" is used to add linear resource clauses, where $\mathtt{A}_i$ is its head, $\mathtt{A}_j$ is its *closure* that consists of the compiled code and a set of bindings for free variables. We replaced this instruction with two new instructions "`add_exact_timed_res` $\mathtt{A}_i$, $\mathtt{A}_j$, n" and "`add_timed_res` $\mathtt{A}_i$, $\mathtt{A}_j$, n". The former is used to add a resource clause S_i ($1 \leq i \leq m$) in $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$, where $\mathtt{A}_i$ is its head, $\mathtt{A}_j$ is its closure, and n is the multiplicity of $\bigcirc$. The latter is used to add a resource clause S_i ($1 \leq i \leq m$) in $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$, $\mathtt{A}_i$ is its head, $\mathtt{A}_j$ is its closure, and n is the multiplicity of $\bigcirc$.

- In the LLPAM, the instruction "`pickup_resource` p/n, $\mathtt{A}_i$, L" finds a consumable resource with predicate symbol p/n by checking its consumption level, and then it sets its index value to $\mathtt{A}_i$. If there are no consumable resources, it jumps to L. We need to improve this instruction so that it checks not only the level condition but also the time condition by comparing the consumption time (the `time` field) of resources with the current time (the current value of `TI`).

7.7 Performance Evaluation

We have developed a TLLPAM-based compiler system, called TLLP. TLLP is a first generation compiler system for a temporal linear logic programming language. The system consists of a TLLP to TLLPAM compiler (written in Prolog) and a emulator (written in C), but it does not incorporate well-known optimizations, register allocation, last-call-optimization, global analysis, and so on.

We compare the execution speeds of two Timed Petri Net programs. One is a TLLP program in Figure 7.5 compiled under TLLP 0.1.3 (TLLPAM code), where time-dependent resources are compiled into closures and kept in the resource table. Another is a LLP program in Figure 7.9 compiled under a LLP compiler 0.5.1, where time-dependent resources are translated into corresponding LLP resources that include time information as arguments.

Table 7.1 shows the performance results. All times in the Table were collected on Linux system (Pentium III 850MHz, 128M memory).

TLLP is 1.7 times faster than translating TLLP into LLP. The speedup is due to quick access to consumable resources without creating redundant choice point frames. In TLLP, the `pickup_resource` instruction is used to find consumable resources by checking not only the level condition but also the time condition. However, in LLP, the `pickup_resource` instruction checks only level condition to find them, and time condition will be checked in the body of the added resources.

The newest package of TLLP (version is 0.1.3) is available from:

`http://kaminari.scitec.kobe-u.ac.jp/tllp/`.

Chapter 8

Conclusion and Future Work

In this dissertation, we proposed new compilation methods to develop efficient implementation for linear logic programming languages. Main contributions are summarized as follows:

1. A compiler system for a linear logic programming language:
   We presented a method for compiling resources and provided an extension of the WAM for a linear logic programming language LLP. In performance, our compiler provided 40% speedup for a theorem proving application of classical logic, relative to its Prolog implementation.

2. A translator system from a linear logic programming language into Java:
   We presented a LLP-to-Java source-to-source translator system. Our translation method is based on continuation passing style compilation. In performance, our translator is 1.7 times faster for a set of classical Prolog benchmarks, than an existing Prolog-to-Java translator jProlog.

3. A compiler system for a temporal linear logic programming language:
   We presented theory and design of a logic programming language based on intuitionistic temporal linear logic, called TLLP. We also presented an abstract machine and its instruction set for TLLP compiler system, and a method for translating TLLP into LLP. In performance, our compiler is 1.7 times faster for a simple example of Timed Petri Net, than translating TLLP into LLP.

The latest packages of those systems are available through WWW:

- LLP version 0.5.1

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

- Prolog Café version 0.5.0

  `http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/`,

- TLLP version 0.1.3

  `http://kaminari.scitec.kobe-u.ac.jp/tllp/`.

This dissertation is the latest step in a course of research begun by N. Tamura and Y. Kaneda towards efficient implementation for linear logic programming languages. Our compiler has already applied to a theorem proving application of first-order classical logic, in which linear logic operators were elegantly used for specifying the problems. Furthermore it gives significantly nice performance relative to its famous Prolog implementation. We believe that our results will be equally well applied to other resource-conscious applications based on linear logic.

However, it is not a full story, and there are many points yet to be investigated. There are at least two directions on the future work.

First, we want to improve the resulting systems presented in this dissertation:

- LLP does not include well-known optimizations such as register allocation, last-call-optimization, global analysis, and shallow backtracking, and so on.
- LLP does not support the universal quantifiers in goal, and dynamic compilation of resources.
- TLLP does not support $\top$-flag for eliminating non-determinism of the treatment of $\top$.

Second, we want to implement the followings using our results:

- A Forum-to-Lolli translator
  Forum is a presentation of full fragment of linear logic. By translating Forum into Lolli, it might be possible to develop an efficient linear logic theorem prover. Translated Lolli programs can be compiled and executed under LLP compiler system.
- A system for specifying real-time finite-state systems in linear logic
  M. I. Kanovich, M. Okada and A. Scedrov proposed a logical formalization for specifying real-time finite-state systems in linear logic. By using LLP, it might be possible to efficiently check the important properties such as *safety* for given specifications.
- A compiler system for a full fragment of Lolli.
  As with λProlog, Lolli allows nested quantification, the use of L_λ higher-order quantification, and unification of λ-term. It might be possible to develop a compiler that supports such higher-order features by extending Teyjus (λProlog compiler) with LLPAM instructions.

Finally, we outline an on-going research using Prolog Café. We are extending Prolog Café for multi-threaded and distributed execution, and are developing interfaces for several constraint solvers such as Mathematica. The goal of this research is to develop a heterogeneous constraint solving system for Java, in which the solvers run on individual threads, and exchange their answers with each other.

Appendix A

The LLPAM at a Glance

A.1 The LLPAM Instructions

In addition to all the instructions of the WAM, the LLPAM includes the instructions listed below. We use the following notations:

- $tag(x)$ stands for the tag field of tagged data cell x.
- $car(x)$ stands for the first element of list cell x.
- $cdr(x)$ stands for the entire list (except for the first element) of list cell x.
- $@(f/n)$ stands for the index value of f/n in the symbol table.

LINEAR IMPLICATION INSTRUCTIONS

`begin_imp` $\mathtt{Y}_i$
`add_res` $\mathtt{A}_i$, $\mathtt{A}_j$
`more_imp`
`mid_imp` $\mathtt{Y}_j$, $\mathtt{Y}_k$
`end_imp` $\mathtt{Y}_i$, $\mathtt{Y}_j$, $\mathtt{Y}_k$

INTUITIONISTIC IMPLICATION INSTRUCTIONS

`begin_exp_imp` $\mathtt{Y}_i$
`add_exp_res` $\mathtt{A}_i$, $\mathtt{A}_j$
`mid_exp_imp` $\mathtt{Y}_j$, $\mathtt{Y}_k$
`end_exp_imp` $\mathtt{Y}_i$, $\mathtt{Y}_j$, $\mathtt{Y}_k$

WITH INSTRUCTIONS

`begin_with` $\mathtt{Y}_i$
`mid_with` $\mathtt{Y}_j$
`end_with` $\mathtt{Y}_i$, $\mathtt{Y}_j$

BANG INSTRUCTIONS

`begin_bang` $\mathtt{Y}_i$
`end_bang` $\mathtt{Y}_i$

TOP INSTRUCTION

`top`

CLOSURE INSTRUCTIONS

`put_closure` L, m, $\mathtt{A}_i$
`execute_closure` $\mathtt{A}_i$

RESOURCE CONTROL INSTRUCTIONS

`pickup_resource` f/n, $\mathtt{A}_i$, L
`consume` $\mathtt{A}_i$, $\mathtt{A}_j$
`if_no_resource` L

CHOICE INSTRUCTIONS

`try_resource` L
`restore_resource`
`retry_resource_else` L
`trust_resource` L

CONTROL INSTRUCTIONS

`call` P, N
`execute` P

LINEAR IMPLICATION INSTRUCTIONS

`begin_imp` $\mathtt{Y}_i$

Used when the implication operator is $\multimap$. Store the current value of R in a new permanent variable $\mathtt{Y}_i$. Save the current value of R in R0. Continue execution with the following instruction.

```
Y_i := ⟨RES, R⟩;
R0 := R;
P := P + instruction_size(P);
```

`add_res` $\mathtt{A}_i$, $\mathtt{A}_j$

Used when the implication operator is $\multimap$. Add a record for a (linear) resource clause of the form $\forall\vec{x}.A$ or $\forall\vec{x}.(G \multimap A)$ as a new entry at the top of the resource table, RES. The value of L is stored in the `level` field and the `deadline` field, the `out_of_scope` flag is set to false. $\mathtt{A}_i$ and $\mathtt{A}_j$ are pointers to structures previously built on the heap holding the head part and closure of the clause respectively. Perform $register\_resource(\mathtt{A}_i)$. This registers the value of R (the index of added resource clause) to the hash and symbol tables for speed access to the resources in RES. The return value, the index of the predicate symbol of $\mathtt{A}_i$ in the symbol table, is set to the `pred` field. Increment R by one. Continue execution with the following instruction.

```
RES[R].level := L;
RES[R].deadline := L;
RES[R].out_of_scope := false;
RES[R].head := A_i;
RES[R].body := undef;
RES[R].closure := A_j;
RES[R].pred := register_resource(A_i);
R := R + 1;
P := P + instruction_size(P);
```

`more_imp`

Used between the codes that load the (linear) resource R_i in the goal of the form $R_1 \multimap (R_2 \multimap \cdots (R_n \multimap G)\cdots)$ or $(R_1 \otimes R_2 \otimes \cdots \otimes R_n) \multimap G$. Set the s1 and s2 fields of added resource clauses in R_i to the current values of R0 (the index of first resource clause) and R (the top of the resource table), respectively. Add the current value of R0 to RLIST, a list of indices of all linear resources. Trail this change by pushing a constant [] onto the trail stack. Update the value of R0 with R. Continue execution with the following instruction.

```
for i := R0 to R-1 do begin
   RES[i].s1 := R0;
   RES[i].s2 := R
end;
HEAP[H] := ⟨RES, R0⟩;
HEAP[H+1] := RLIST;
RLIST := ⟨LIS, H⟩;
H := H + 2;
TRAIL[TR] := ⟨CON, []⟩;
TR := TR + 1;
R0 := R;
P := P + instruction_size(P);
```

`mid_imp` $\mathtt{Y}_j$, $\mathtt{Y}_k$

Used between the code that loads the resource and the code for the subgoal when the implication operator is $\multimap$. Store the current values of R (the top of the resource table) and T ($\top$-flag) to the permanent variables $\mathtt{Y}_j$ and $\mathtt{Y}_k$, respectively. Set the value of T to false. Set the s1 and s2 fields of added resource clauses in R to the current values of R0 and R, respectively. Add the current value of R0 to RLIST, a list of indices of all linear resources. Trail this change by pushing a constant [] onto the trail stack. Continue execution with the following instruction.

```
Y_j := ⟨RES, R⟩;
Y_k := ⟨TOP, T⟩;
T := false;
for i := R0 to R-1 do begin
    RES[i].s1 := R0;
    RES[i].s2 := R
end;
HEAP[H] := ⟨RES, R0⟩;
HEAP[H+1] := RLIST;
RLIST := ⟨LIS, H⟩;
H := H + 2;
TRAIL[TR] := ⟨CON, []⟩;
TR := TR + 1;
P := P + instruction_size(P);
```

`end_imp` $\mathtt{Y}_i$, $\mathtt{Y}_j$, $\mathtt{Y}_k$

Used after the code for the subgoal when the implication operator is $\multimap$. If there are any resources in positions from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ that have not been consumed, fail. Otherwise, set the out_of_scope flags of all records from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ to true (trailing so that they may be reset on backtracking), and set the register T to $\mathtt{Y}_k \vee$ T. In order to account for the use of $\top$ at the top level of the subgoal, 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 unconsumed.
- 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.

```
⟨RES, m⟩ := Y_i;
⟨RES, n⟩ := Y_j;
for r := m to n - 1 do begin
    ℓ := RES[r].level;
    d := RES[r].deadline;
    consumed := ((ℓ = U) ∧ (d = 0)) ∨ (T ∧ (ℓ = L) ∧ (d = L));
    if ¬ consumed then
        backtrack
end;
for r := m to n - 1 do
    RES[r].out_of_scope := true;
TRAIL[TR] := ⟨RES, m⟩;
TR := TR + 1;
TRAIL[TR] := ⟨RES, n⟩;
TR := TR + 1;
⟨TOP, flag⟩ := Y_k;
T := flag ∨ T;
P := P + instruction_size(P);
```

INTUITIONISTIC IMPLICATION INSTRUCTIONS

`begin_exp_imp` $\mathtt{Y}_i$

Used when the implication operator is $\Rightarrow$. Store the current value of R in a new permanent variable $\mathtt{Y}_i$. Continue execution with the following instruction.

```
Y_i := ⟨RES, R⟩;
P := P + instruction_size(P);
```

`add_exp_res` $\mathtt{A}_i$, $\mathtt{A}_j$

Used when the implication operator is $\Rightarrow$. Add a record for an (intuitionistic) resource clause of the form $\forall\vec{x}.A$ or $\forall\vec{x}.(G \multimap A)$ as a new entry at the top of the resource table, RES. Behaves the same as `add_res`, except that the `level` and `deadline` fields are set to zero. Continue execution with the following instruction.

```
RES[R].level := 0;
RES[R].deadline := 0;
RES[R].out_of_scope := false;
RES[R].head := A_i;
RES[R].body := undef;
RES[R].closure := A_j;
RES[R].pred := register_resource(A_i);
R := R + 1;
P := P + instruction_size(P);
```

`mid_exp_imp` $\mathtt{Y}_j$, $\mathtt{Y}_k$

Used between the code that loads the resource and the code for the subgoal, when the implication operator is $\Rightarrow$. Store the current values of R (the top of the resource table) and T ($\top$-flag) to the permanent variables $\mathtt{Y}_j$ and $\mathtt{Y}_k$, respectively. Set register T to false. Continue execution with the following instruction.

```
Y_j := ⟨RES, R⟩;
Y_k := ⟨TOP, T⟩;
T := false;
P := P + instruction_size(P);
```

`end_exp_imp` $\mathtt{Y}_i$, $\mathtt{Y}_j$, $\mathtt{Y}_k$

Used after the code for the subgoal, when the implication operator is $\Rightarrow$. The added resource entries need not be examined. Set the `out_of_scope` flags of all records from $\mathtt{Y}_i$ to $\mathtt{Y}_j - 1$ to true (trailing so that they may be reset on backtracking). Set register T to $\mathtt{Y}_k \vee \mathtt{T}$. Continue execution with the following instruction.

```
⟨RES, m⟩ := Y_i;
⟨RES, n⟩ := Y_j;
for r := m to n - 1 do
   RES[r].out_of_scope := true;
TRAIL[TR] := ⟨RES, m⟩;
TR := TR + 1;
TRAIL[TR] := ⟨RES, n⟩;
TR := TR + 1;
⟨TOP, flag⟩ := Y_k;
T := flag ∨ T;
P := P + instruction_size(P);
```

WITH INSTRUCTIONS

`begin_with` $\mathtt{Y}_i$

Used when the conjunction operator is $\&$. Decrement U so that we can tell which resources are consumed in the left conjunct. Store the current value of T in a new permanent variable $\mathtt{Y}_i$ and set register T to false. Continue execution with the following instruction.

```
U := U - 1;
Y_i := ⟨TOP, T⟩;
T := false;
P := P + instruction_size(P);
```

`mid_with` $\mathtt{Y}_j$

Used between the code for the left and right conjuncts when the conjunction operator is $\&$. Perform $changepair$. 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 T is true, then perform $change$ so that resources that were available but not explicitly used, but which $\top$ can be thought of as having used, are also available for use in the second conjunct. Increment L and U. Store the current value of the T register to a new permanent variable $\mathtt{Y}_j$. Set the T register to false. Continue execution with the following instruction.

```
changepair(U, 0, L+1, L+1);
if T then
    change(L, L+1);
L := L + 1;
U := U + 1;
Y_j := ⟨TOP, T⟩;
T := false;
P := P + instruction_size(P);
```

`end_with` $\mathtt{Y}_i$, $\mathtt{Y}_j$

Used after the code for the right conjunct when the conjunction operator is $\&$. Decrement register L. If T is true, then perform $changepair$ ($\top$ 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 $consumed$ to check whether all the resources that should have been consumed, were consumed. If this fails, backtrack. Otherwise, If $\mathtt{Y}_j$ is true, then perform $change$ (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 $\mathtt{Y}_i \vee (\mathtt{Y}_j \wedge \mathtt{T})$. Continue execution with the following instruction.

```
⟨TOP, flag1⟩ := Y_i;
⟨TOP, flag2⟩ := Y_j;
L := L - 1;
if T then
    changepair(L+1, L+1, U, 0)
else if ¬ consumed(L+1) then
    backtrack;
if flag2 then
    change(L+1, L);
T := flag1 ∨ (flag2 ∧ T);
P := P + instruction_size(P);
```

BANG INSTRUCTIONS

`begin_bang` $\mathtt{Y}_i$

Increment `L`. Store the value of `T` in a new permanent variable $\mathtt{Y}_i$. Continue execution with the following instruction.

```
L := L + 1;
Y_i := ⟨TOP, T⟩;
P := P + instruction_size(P);
```

`end_bang` $\mathtt{Y}_i$

Decrement `L`. Set the value of the register `T` from the variable $\mathtt{Y}_i$. Continue execution with the following instruction.

```
⟨TOP, flag⟩ := Y_i;
L := L - 1;
T := flag;
P := P + instruction_size(P);
```

TOP INSTRUCTION

`top`

Set the register `T` to true. Continue execution with the following instruction.

```
T := true;
P := P + instruction_size(P);
```

CLOSURE INSTRUCTIONS

`put_closure` L, m, $\mathtt{A}_i$

Set register $\mathtt{A}_i$ to a new `CLO` cell pointing to the current top of the heap. Push L (code address) and m (the number of free variables) on the heap. Set mode to `write`. Continue execution with the following instruction. The `unify_value` (or `unify_variable`) instruction that follows this instruction, pushes the m references to free variables on the heap.

```
A_i := ⟨CLO, H⟩;
HEAP[H] := L;
H := H + 1;
HEAP[H] := m;
H := H + 1;
mode := write;
P := P + instruction_size(P);
```

`execute_closure` $\mathtt{A}_i$

Save the current choice point `B` in `B0`. Set the register `S` to $c+2$ pointing to the top of the references to free variables. Set mode to `read`. Continue execution with instruction on `HEAP[`c`]`.

```
⟨CLO, c⟩ := A_i;
B0 := B;
S := c + 2;
mode := read;
P := HEAP[c];
```

RESOURCE CONTROL INSTRUCTIONS

`pickup_resource` f/n, $\mathtt{A}_i$, L

Find an index of consumable resource with predicate symbol f/n from R1 and R2. Set that index to $\mathtt{A}_i$. Continue execution with the following instruction. If there are no consumable resources, jump to the instruction labeled L.

```
found := false;
while (tag(R1) = LIS) ∧ (¬ found) do begin
    ⟨RES, r⟩ := car(R1); R1 := cdr(R1);
    found := (RES[r].pred = @(f/n))
             ∧ (¬ RES[r].out_of_scope)
             ∧ (RES[r].level = 0 ∨ RES[r].level = L)
end;
while (tag(R2) = LIS) ∧ (¬ found) do begin
    ⟨RES, r⟩ := car(R2); R2 := cdr(R2);
    found := (RES[r].pred = @(f/n))
             ∧ (¬ RES[r].out_of_scope)
             ∧ (RES[r].level = 0 ∨ RES[r].level = L)
end;
if found then Ai := ⟨RES, r⟩ else P := L;
```

`if_no_resource` L

Check whether there are any consumable resources in R1 and R2. If there are no consumable resources, jump to the instruction labeled L.

```
found := false;
while (tag(R1) = LIS) ∧ (¬ found) do begin
    ⟨RES, r⟩ := car(R1);
    found := (¬ RES[r].out_of_scope)
             ∧ (RES[r].level = 0 ∨ RES[r].level = L);
    if found then break;
    R1 := cdr(R1)
end;
while (tag(R2) = LIS) ∧ (¬ found) do begin
    ⟨RES, r⟩ := car(R2);
    found := (¬ RES[r].out_of_scope)
             ∧ (RES[r].level = 0 ∨ RES[r].level = L);
    if found then break;
    R2 := cdr(R2)
end;
if R1 = [] then
    begin R1 := R2; R2 := [] end;
if tag(R1) ≠ LIS then
    P := L
else
    P := P + instruction_size(P);
```

`consume` A_i, A_j

Mark the entry `RES[`A_i`]` as consumed (set `level` to the current value of `U`, and `deadline` to 0). Set A_j to the value of `closure` field. Continue execution with the following instruction.

```
⟨RES, r⟩ := A_i;
if RES[r].level ≠ 0 then begin
    changelevel(r, U);
    changedeadline(r, 0)
end;
A_j := RES[r].closure;
P := P + instruction_size(P);
```

CHOICE INSTRUCTIONS

`try_resource` L

Allocate a new choice point frame on the stack. Behaves the same as `try` L, except that `R1` and `R2` are also saved. Continue execution with the following instruction labeled L.

```
newB := bottom_of_stack;
STACK[newB] := num_of_args;
n := STACK[newB];
for i := 1 to n do STACK[newB+i] := A_i;
STACK[newB+n+1] := E;
STACK[newB+n+2] := CP;
STACK[newB+n+3] := B;
STACK[newB+n+4] := P + instruction_size(P);
STACK[newB+n+5] := TR;
STACK[newB+n+6] := H;
STACK[newB+n+7] := B0;
STACK[newB+n+8] := R;
STACK[newB+n+9] := L;
STACK[newB+n+10] := U;
STACK[newB+n+11] := T;
STACK[newB+n+12] := R1;
STACK[newB+n+13] := R2;
B := newB;
HB := H;
P := L;
```

`restore_resource`

Having backtracked to the current choice point, reset all the necessary information from it. Continue execution with the following instruction.

```
n := STACK[B];
for i := 1 to n do Ai := STACK[B+i];
E := STACK[B+n+1];
CP := STACK[B+n+2];
unwind_trail(STACK[B+n+5], TR);
TR := STACK[B+n+5];
H := STACK[B+n+6];
R := STACK[B+n+8];
L := STACK[B+n+9];
U := STACK[B+n+10];
T := STACK[B+n+11];
R1 := STACK[B+n+12];
R2 := STACK[B+n+13];
P := P + instruction_size(P);
```

`retry_resource_else` L

Update the next clause field to L. Update the `R1` and `R2` fields in the current choice point frame with their current values. Continue execution with the following instruction.

```
n := STACK[B];
STACK[B+n+4] := L;
STACK[B+n+12] := R1;
STACK[B+n+13] := R2;
HB := H;
P := P + instruction_size(P);
```

`trust_resource` L

Discard the current choice point frame by resetting `B` to its predecessor. Continue execution with the following instruction labeled L.

```
n := STACK[B];
B := STACK[B+n+3];
HB := STACK[B+STACK[B]+6];
P  := L;
```

CONTROL INSTRUCTIONS

`call` P, N

Save the current choice point's address `B` in `B0`. Save the value of current continuation in `CP`. If the predicate P is defined, then perform *lookup_hash*. This extracts the list of indices of the possibly consumable resource clauses in the resource table by referring to the hash and symbol tables. Set the extracted lists to `R1` and `R2` (Set [] if there are no consumable resources). Continue execution with the instruction labeled by P.

```
num_of_args := SYMBOL[@(P)].arity;
CP := P + instruction_size(P);
B0 := B;
if SYMBOL[@(P)].codeaddr = undef then
    backtrack;
if tag(SYMBOL[@(P)].res) ≠ LIS then
    begin R1 := []; R2 := [] end
else
    lookup_hash(@(P));
P := SYMBOL[@(P)].codeaddr;
```

`execute` P

Save the current choice point's address `B` in `B0`. If the predicate P is defined, then perform $lookup\_hash$. This extracts the list of indices of the possibly consumable resource clauses in the resource table by referring to the hash and symbol tables. Set the extracted lists to `R1` and `R2` (Set [] if there are no consumable resources). Continue execution with the instruction labeled by P.

```
num_of_args := SYMBOL[@(P)].arity;
B0 := B;
if SYMBOL[@(P)].codeaddr = undef then
    backtrack;
if tag(SYMBOL[@(P)].res) ≠ LIS then
    begin R1 := []; R2 := [] end
else
    lookup_hash(@(P));
P := SYMBOL[@(P)].codeaddr;
```

A.2 The LLPAM Auxiliary Procedures and Functions

We summarize auxiliary operations used in the LLPAM instructions. We use the notation $\&(x)$ to stand for the address of x.

The $deref$ function

```
function deref(a: address): address;
begin
    ⟨tag, val⟩ := STORE[a];
    if (tag = REF) ∧ (val ≠ a) then
        return deref(val)
    else
        return a
end {deref};
```

The $backtrack$ procedure

```
procedure backtrack;
begin
    if B = bottom_of_stack then
        exit_program
    else begin
        B0 := STACK[B+STACK[B]+7];
        P := STACK[B+STACK[B]+4]
    end
end {backtrack};
```

The $register\_resource$ function

Registers the current value of `R` (the index of added resource clause) to the symbol and hash tables to speed access to the resources in the resource table.

```
function register_resource(a: address): Integer;
begin
    ⟨tag, val⟩ := STORE[a];
    case tag of
        CON : idx := @(val);
        STR : idx := @(STORE[val]);
    end;
    if SYMBOL[idx].res = undef then SYMBOL[idx].res := [];
    HEAP[H] := ⟨RES, R⟩;
    HEAP[H+1] := SYMBOL[idx].res;
    SYMBOL[idx].res := ⟨LIS, H⟩;
    H := H + 2;
    TRAIL[TR] := ⟨LIS, &(SYMBOL[idx].res)⟩;
    TR := TR + 1;
    if SYMBOL[idx].res2 = undef then SYMBOL[idx].res2 := [];
    h := hash(a, hashsize);
    if h = undef then begin
        HEAP[H] := ⟨RES, R⟩;
        HEAP[H+1] := SYMBOL[idx].res2;
        SYMBOL[idx].res2 := ⟨LIS, H⟩;
        H := H + 2;
        TRAIL[TR] := ⟨LIS, &(SYMBOL[idx].res2)⟩;
        TR := TR + 1;
    end
    else begin
        HEAP[H] := ⟨RES, R⟩;
        HEAP[H+1] := HASH[h];
        HASH[h] := ⟨LIS, H⟩;
        H := H + 2;
        TRAIL[TR] := ⟨LIS, &(HASH[h])⟩;
        TR := TR + 1;
    end;
    return idx
end {register_resource};
```

The *hash* and *hash_one_level* functions

In current implementation, the entries of the resource table are hashed on the predicate symbol/arity and the first argument.

```
function hash(a:address, hashsize:Integer): Integer;
begin
    case STORE[a] of
        ⟨REF, _⟩: return undef;
        ⟨INT, _⟩, ⟨CON, _⟩: return (hash_one_level(a) mod hashsize);
        ⟨STR, addr⟩: arg1 := STORE[addr+1];
        ⟨LIS, addr⟩: arg1 := STORE[addr];
    end;
    arg1 := deref(arg1);
    ⟨tag, val⟩ := arg1;
```

```
    if tag = REF then return undef;
    h := hash_one_level(a);
    h := add_hash(h, hash_one_level(arg1));
    return (h mod hashsize)
end {hash};

function hash_one_level(a:address): Integer;
begin
    case STORE[a] of
        ⟨REF, _⟩: return undef;
        ⟨INT, i⟩: return i;
        ⟨CON, c⟩: return SYMBOL[@(c)].hash_value;
        ⟨STR, addr⟩: return SYMBOL[@(STORE[addr])].hash_value;
        ⟨LIS, _⟩: return SYMBOL[@(.)].hash_value;
    end
end {hash_one_level};
```

The *changelevel* procedure

```
procedure changelevel(i, ℓ: Integer);
begin
    if RES[i].level ≠ ℓ then begin
        TRAIL[TR] := ⟨TOP, true⟩;
        TR := TR + 1;
        TRAIL[TR] := ⟨RES, i⟩;
        TR := TR + 1;
        TRAIL[TR] := ⟨INT, RES[i].level⟩;
        TR := TR + 1;
        for k := RES[i].s1 to (RES[i].s2) - 1 do
            RES[k].level := ℓ;
    end
end {changelevel};
```

The *changedeadline* procedure

```
procedure changedeadline(i, ℓ: Integer);
begin
    if RES[i].deadline ≠ ℓ then begin
        TRAIL[TR] := ⟨TOP, false⟩;
        TR := TR + 1;
        TRAIL[TR] := ⟨RES, i⟩;
        TR := TR + 1;
        TRAIL[TR] := ⟨INT, RES[i].deadline⟩;
        TR := TR + 1;
        for k := RES[i].s1 to (RES[i].s2) - 1 do
            RES[k].deadline := ℓ;
    end
end {changedeadline};
```

The *changepair* procedure

```
procedure changepair(ℓ1, d1, ℓ2, d2: Integer);
begin
    p := RLIST;
    while tag(p) = LIS do begin
        ⟨RES, i⟩ := car(p);
        found := (¬ RES[i].out_of_scope)
                 ∧ (RES[i].level = ℓ1)
                 ∧ (RES[i].deadline = d1);
        if found then begin
            changelevel(i, ℓ2);
            changedeadline(i, d2)
        end;
        p := cdr(p)
    end
end {changepair};
```

The *change* procedure

```
procedure change(ℓ1, ℓ2: Integer);
begin
    p := RLIST;
    while tag(p) = LIS do begin
        ⟨RES, i⟩ := car(p);
        found := (¬ RES[i].out_of_scope) ∧ (RES[i].level = ℓ1);
        if found then
            changelevel(i, ℓ2);
        p := cdr(p)
    end
end {change};
```

The *consumed* function

```
function consumed(ℓ: Integer): Boolean;
begin
    p := RLIST;
    while tag(p) = LIS do begin
        ⟨RES, i⟩ := car(p);
        not_consumed := (¬ RES[i].out_of_scope) ∧ (RES[i].level = ℓ);
        if not_consumed then
            return false;
        p := cdr(p)
    end;
    return true
end {consumed};
```

The *lookup_hash* procedure

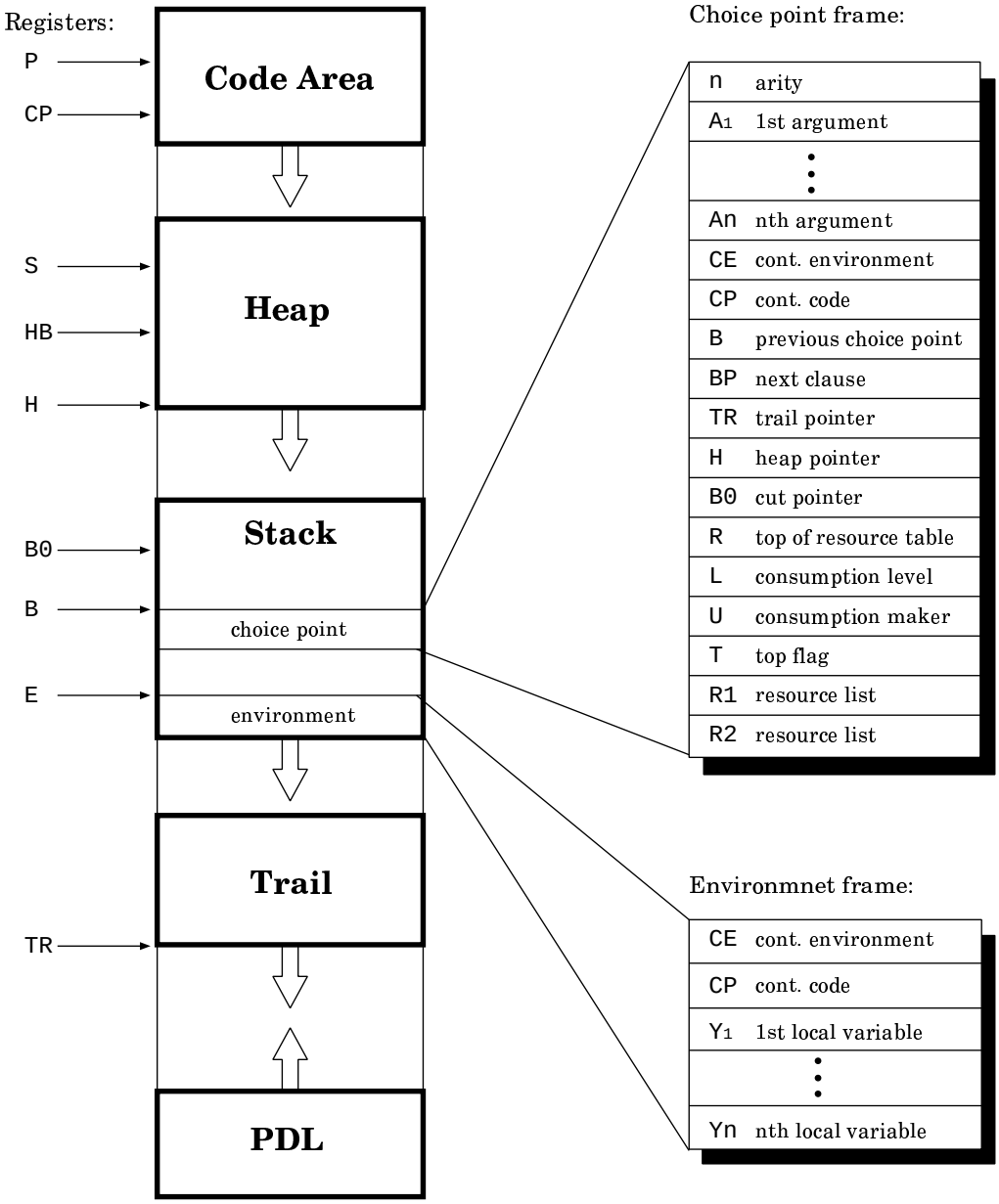
```
procedure lookup_hash(s: Integer);
begin
    if num_of_args = 0 then begin
        h := (SYMBOL[s].hash_value mod hashsize);
        R1 := HASH[h];
        R2 := []
    end
    else begin
        addr := deref(A1);
        ⟨tag, _⟩ := STORE[addr];
        if tag = REF then begin
            R1 := SYMBOL[s].res;
            R2 := []
        end
        else begin
            h := SYMBOL[s].hash_value;
            h := (add_hash(h, hash_one_level(addr)) mod hashsize);
            R1 := HASH[h];
            R2 := SYMBOL[s].res2
        end
    end
end {lookup_hash};
```

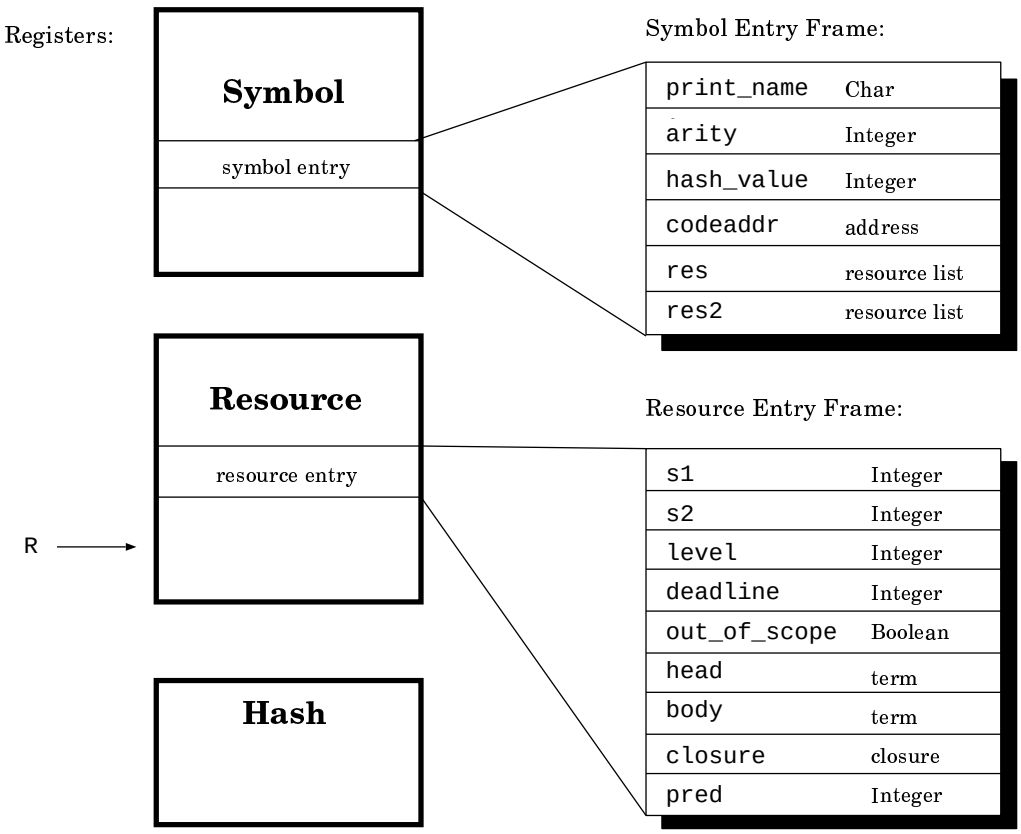
The *unwind_trail* procedure

```
procedure unwind_trail(a1, a2: address);
begin
    p := a2 - 1;
    while p >= a1 do begin
        case TRAIL[p] of
            ⟨REF, _⟩: begin
                STORE[TRAIL[p]] := ⟨REF, TRAIL[p]⟩;
                p := p - 1 end;
            ⟨CON, _⟩: begin
                RLIST := cdr(RLIST);
                p := p - 1 end;
            ⟨INT, n⟩: begin {undo changelevel and changedeadline}
                p := p - 1;
                ⟨RES, m⟩ := TRAIL[p];
                p := p - 1;
                ⟨TOP, flag⟩ := TRAIL[p];
                if flag then
                    for k := RES[m].s1 to RES[m].s2 - 1 do
                        RES[k].level := n
                else
                    for k := RES[m].s1 to RES[m].s2 - 1 do
                        RES[k].deadline := n;
                p := p - 1
                end;
            ⟨LIS, a⟩: begin
```

```
                STORE[a] := cdr(STORE[a]);
                p := p - 1
                end;
            ⟨RES, m⟩: begin {undo end_imp and end_exp_imp}
                p := p - 1;
                ⟨RES, k⟩ := TRAIL[p];
                for ℓ := k to m - 1 do
                    RES[ℓ].out_of_scope := false;
                p := p - 1
                end;
        end
    end
end {unwind_trail};
```

A.3 The LLPAM Memory Layout and Registers




Registers:
Symbol
symbol entry
Resource
resource entry
R
Hash
Symbol Entry Frame:
print_name Char
arity Integer
hash_value Integer
codeaddr address
res resource list
res2 resource list
Resource Entry Frame:
s1 Integer
s2 Integer
level Integer
deadline Integer
out_of_scope Boolean
head term
body term
closure closure
pred Integer

Bibliography

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

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

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

[4] Mutsunori Banbara, Kyoung-Sun Kang, Takaharu Hirai, and Naoyuki Tamura. Logic programming in a fragment of intuitionistic temporal linear logic. In Philippe Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming (ICLP'01)*, pages 315–330. Springer-Verlag LNCS 2237, November 2001.

[5] Mutsunori Banbara, Kyoung-Sun Kang, and Naoyuki Tamura. Java implementation of a linear logic programming language. *Information Processing Society of Japan Transactions on Programming*, 40(SIG 10 (PRO 5)):1–16, December 1999. (in Japanese).

[6] Mutsunori Banbara, Kyoung-Sun Kang, and Naoyuki Tamura. An abstract machine for a compiler system of a linear logic programming language. *Computer Software, Japan Society for Software Science and Technology*, 18(1):39–60, 2001. (in Japanese).

[7] Mutsunori Banbara, Kyoung-Sun Kang, and Naoyuki Tamura. An abstract machine for a compiler system of a temporal linear logic programming language. *Information Processing Society of Japan Transactions on Programming*, 42(SIG 11 (PRO 12)):52–66, November 2001. (in Japanese).

[8] Mutsunori Banbara and Naoyuki 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, October 1997.

[9] Mutsunori Banbara and Naoyuki Tamura. Compiling resources in a linear logic programming language. In Konstantinos Sagonas, editor, *Proceedings of the JICSLP'98 Post Conference Workshop 7 on Implementation Technologies for Programming Languages based on Logic*, pages 32–45, June 1998.

[10] Mutsunori Banbara and Naoyuki Tamura. Translating a linear logic programming language into Java. In M. Carro, I .Dutra, et al., editors, *Proceedings of the ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 19–39, December 1999.

[11] Ivan Bratko. *Prolog programming for artificial intelligence*. Addison-Wesley, 1986.

[12] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW'99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[13] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the Fifth International Workshop on Extensions of Logic Programming — ELP'96*, pages 67–81, Leipzig, Germany, 28–30 March 1996. Springer-Verlag LNAI 1050.

[14] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science — LICS'96*, pages 264–275, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press. This work also appeared as Preprint 1834 of the Department of Mathematics of Technical University of Darmstadt, Germany.

[15] Jawahar Chirimar. *Proof Theoretic approach to specification language*. PhD thesis, University of Pennsylvania, February 1995.

[16] Philippe Codognet and Daniel Diaz. `WAMCC`: Compiling Prolog to C. In Leon Sterling, editor, *Proceedings of International Conference on Logic Programming*, pages 317–331. The MIT Press, Jun 1995.

[17] Jon Cook. P#: Using prolog within the .net framework. Technical report, University of Edinburgh, to appear.

[18] Bart Demoen and Paul Tarau. jProlog home page. `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`.

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

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

[21] James Harland, David Pym, and Michael Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, pages 391–405, Munich, Germany, July 1996. Springer-Verlag LNCS 1101.

[22] James Harland and Michael Winikoff. Implementing the linear logic programming language Lygon. In J. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 66–80, Portland, Oregon, 1995.

[23] Takaharu Hirai. An application of temporal linear logic to Timed Petri Nets. In *Proceedings of the Petri Nets'99 Workshop on Applications of Petri Nets to Intelligent System Development*, pages 2–13, June 1999.

[24] Joshua S. Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 622–636, Washington, DC, November 1992.

[25] 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.

[26] 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, Amsterdam, July 15–18, 1991.

[27] Joshua S. Hodas and Jeffrey Polakow. Forum as a logic programming language: Preliminary results and observations. In M. Okada, editor, *Proceedings of the Linear Logic '96 Meeting*, volume 3, Tokyo, Japan, 1996. Elsevier Electronic Notes in Theoretical Computer Science.

[28] Joshua S. Hodas and Naoyuki Tamura. Lollicop - a linear logic implementation of a lean connection-method theorem prover for first-order classical logic. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of First International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 670–684. Springer-Verlag LNCS 2083, 2001.

[29] Joshua S. Hodas, Kevin Watkins, Naoyuki Tamura, and Kyoung-Sun Kang. Efficient implementation of a linear logic programming language. In Joxan Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 145–159. MIT Press, June 1998.

[30] Kyoung-Sun Kang, Mutsunori Banbara, and Naoyuki Tamura. Efficient resource management model for linear logic programming languages. *Computer Software, Japan Society for Software Science and Technology*, 18(0):138–154, 2001. (in Japanese).

[31] Max I. Kanovich and Takayasu Ito. Temporal linear logic specifications for concurrent processes (extended abstract). In *Proceedings of 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 48–57, 1997.

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

[33] Naoki Kobayashi and Akinori Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3:279–294, 1994. Short version appeared in Joint International Conference and Symposium on Logic Programming, Washington, DC, November 1992, Workshop on Linear Logic and Logic Programming.

[34] Argonne National Laboratory. Otter and MACE on TPTP v2.3.0. Web page at `http://www-unix.msc.anl.gov/AR/otter/tptp230.html`, May 2000.

[35] Pablo López and Ernesto Pimentel. A lazy splitting system for forum. In M.Falaschi, M.Navarro, and A.Policriti, editors, *Proceedings of the Joint Conference on Declarative Programming*, pages 247–258, 1997.

[36] W. MacCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[37] M. Okada M.I. Kanovich and A. Scedrov. Specifying real-time finite-state systems in linear logic. In Frank S. de Boer and Maurizio Gabbrielli, editors, *Electronic Notes in Theoretical Computer Science*, volume 16. Elsevier Science Publishers, 2000.

[38] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[39] Dale Miller. An overview of linear logic programming. In Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott, editors, *Submitted as a chapter for a book on linear logic*. Cambridge University Press.

[40] Dale Miller. A survey of linear logic programming. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 2(2):63–67, December 1995.

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

[42] Gopalan Nadathur. The metalanguage λprolog and its implementation. In Herbert Kuchen and Kazunori Ueda, editors, *Proceedings of the Fifth International Symposium on Functional and Logic Programming (FLOPS'01)*, pages 1–20. Springer-Verlag LNCS 2024, March 2001.

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

[44] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[45] Gopalan Nadathur and Guanshan Tong. Realizing modularity in λprolog. *Journal of Functional and Logic Programming*, 9, April 1999.

[46] J. Otten and W. Bibel. leanCoP: lean connection-based theorem proving. In *Proceedings of the Third International Workshop on First-Order Theorem Proving*, pages 152–157. University of Koblenz, 2000. Electronically available, along with submitted journal-length version, at `http://www.intellektik.informatik.tu-darmstadt.de/~jeotten/leanCoP/`.

[47] Mutsunori Banbara Eiji Sugiyama, Kyoung-Sun Kang, and Naoyuki Tamura. Towards a logic programming based on linear logic. In *Proceedings of the Symposium on Industrial Applications of Prolog 1995*, pages 65–72, October 1995. (in Japanese).

[48] G. Sutcliffe and C. Suttner. The TPTP problem library—CNF release v1.2.1. *Journal of Automated Reasoning*, 21:177–203, 1998.

[49] Naoyuki Tamura and Yukio 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, November 1996.

[50] Makoto Tanabe. Timed petri nets and temporal linear logic. In *Lecture Notes in Computer Science 1248: Proceedings of Application and Theory of Petri Nets*, pages 156–174, June 1997.

[51] 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.

[52] Paul 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`.

[53] Paul Tarau. Jinni: a Lightweight Java-based Logic Engine for Internet Programming. In Kostis Sagonas, editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, Manchester, U.K., jun 1998. invited talk.

[54] Paul Tarau and Michel 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, August 1990.

[55] Anne S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

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

[57] Michael Winikoff. W-Prolog home page.
`http://goanna.cs.rmit.edu.au/~winikoff/wp/`.

[58] Eric Wohlstadter, Stoney Jackson, and Premkumar T. Devanbu. Generating wrappers for command line programs: The cal-aggie wrap-o-matic project. In *International Conference on Software Engineering*, pages 243–252, 2001.