線形論理型言語のJava言語による処理系の設計と実装

番原睦則[†] 姜 京順[††] 田村直之[††]

フランスの論理学者 J.-Y. Girard によって線形論理 (linear logic) [1] が発表されて以来，プログラミング言語への応用が盛んに研究されている．線形論理に基づく論理型言語については，すでにいくつかが提案されている：Lolli[2]，Lygon[3]，LO[4]，LinLog[5]，Forum[6]，HACL[7]．しかし，Java上の処理系が実現されたものはまだ存在していない．

これらの言語では，動的に生成，消費される「リソース」を論理式のレベルで表現できる．そのため効率のよいリソース処理は実装上最も重要な問題の1つである．Lolli，Lygon，Forum については，インタプリタ処理系が開発されている（Lolli は SML 及び λProlog 上，Lygon は Prolog 上，Forum は SML，Prolog 及び λProlog 上）．これらの処理系では，リソースは木構造として表現されているため，リソースを消費した場合に木構造の再構築が必要となり，効率低下の原因となっている．

本稿では，線形論理型言語 LLP の概要と，LLP から Java へのトランスレート方式について述べる．LLP 言語は Prolog のスーパーセットで，Lolli のサブセットになっている．リソースはコンパイルコードと自由変数の束縛情報から構成されるクロージャ (closure) にコンパイルされ，ただ1つのリソース表により管理される．リソースの呼び出しは通常のプログラム呼び出しに統合される．

N クイーン配置問題を解く LLP プログラムの実行速度は，標準的な Prolog ベンチマークに含まれる N クイーンプログラムを MINERVA でコンパイルしたものより，$N = 12$ で約 1.5 倍速い．また，Prolog ベンチマークの実行速度は，既存のトランスレータ jProlog と比較して，平均で 3 倍強の高速化を実現している．

Java Implementation of a Linear Logic Programming Language

Mutsunori Banbara,[†] Kyoung-sun Kang[††] and Naoyuki Tamura[††]

There have been several proposals of logic programming languages based on linear logic: Lolli[2], Lygon[3], LO[4], LinLog[5], Forum[6], HACL[7]. In these languages, it is possible to create and consume resources dynamically as logical formulas. The efficient handling of resource formulas is therefore an important issue in the implementation of these languages. Lolli, Lygon, and Forum are implemented as interpreter systems; Lolli is on SML and λProlog, Lygon is on Prolog, Forum is on SML, λProlog and Prolog. But, none of them have not been implemented in Java.

In this paper, we describe a method for translating a linear logic programming language called LLP into Java. LLP is a superset of Prolog and a subset of Lolli. In our design, resource formulas are compiled into *closures* which consist of a reference to compiled code and a set of bindings for free variables. Calling resources are integrated with the ordinary predicate invocation.

LLP program for N-Queen ($N = 12$) is about 1.5 times faster than an N-Queen program in classical Prolog benchmarks which is compiled by MINERVA. On average, the execution speed of generated code for classical Prolog benchmarks is about 3 times faster than existing Prolog-to-Java translator called jProlog.

1. はじめに

線形論理 (linear logic) [1] は，フランスの論理学者 J.-Y. Girard によって考え出された新しい論理体系であり，計算機科学のさまざまな分野への応用が期待されている．特にプログラミング言語への応用は盛んに研究されており，線形論理に基づく論理型言語については，すでにいくつかが提案されている：Lolli[2]，Lygon[3]，LO[4]，LinLog[5]，Forum[6]，HACL[7]．Lolli，Lygon，Forum については，インタプリタ処理系が開発されている（Lolli は SML 及び λProlog 上，Lygon

† 奈良工業高等専門学校
Nara National College of Technology
†† 神戸大学大学院自然科学研究科
Graduate School of Science and Technology, Kobe University

はProlog上，ForumはSML，Prolog及びλProlog上）．しかし，Java上の処理系が実現されたものはまだ存在していない．

これらの言語では，動的に生成（追加），消費される「リソース」を論理式のレベルで表現できる．そのため効率のよいリソース処理は実装上最も重要な問題の1つである．

リソースの生成は，Prologにおける節のassertに似ている（ただし，バックトラックによりキャンセルされる）．例えば，ゴール $(q \multimap p) \Rightarrow G$ におけるリソース $q \multimap p$ の生成は，節 p:-q のassertとまったく同じ効果をもち，ゴール G の実行中に呼び出すことができる．この場合，リソースは節と同じ方法でコンパイルすることができる．

しかし，自由変数を含むリソースのコンパイルは容易でない．例えば，ゴール $(q(X) \multimap p(X)) \Rightarrow G$ 中の変数 X は自由変数であり，リソースの生成後に他のゴールによって代入される可能性がある．すなわち，このリソースは実行時に自由変数の束縛情報を必要とする．そのため普通の節と同じようにコンパイルすることはできない．

本稿では，線形論理型言語LLP[☆1]の概要と，LLPからJavaへのトランスレート方式（特に，リソースコンパイル）について述べる．LLP言語はPrologのスーパーセットで，Lolli[☆2]のサブセットになっている．リソースはコンパイルコードと自由変数の束縛情報から構成されるクロージャ（closure）にコンパイルされ，ただ1つのリソース表により管理される．リソースの呼び出しは通常のプログラム呼び出しに統合される．

現在，本稿で述べる方式に基づくLLP処理系（Prolog Café）の開発を進めており，第6節で標準的なPrologベンチマーク，及び N クイーン配置の全解探索問題を解くLLPプログラムを用いて，Prolog Cafeの性能評価を行う．

2. 線形論理型言語LLP

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

本稿で述べるLLP言語[☆3]は上のような線形論理のフラグメントに基づいている．A はatomic論理式を表し，$\vec{x}$ はスコープ中のすべての自由変数を表す．

C，G，R はそれぞれ“節”，“ゴール”，“リソース”を表す．$\Rightarrow$ は直観主義論理の含意を意味し，$A \Rightarrow B$ と $!A \multimap B$ は同値である．

リソース R の定義は以下のものに置き換えることができる．

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

なぜならば，線形論理では以下の論理的同値性が成り立つからである．

$$
\begin{aligned}
G_1 \multimap (G_2 \multimap R) &\equiv (G_1 \otimes G_2) \multimap R \\
G \multimap (R_1 \,\&\, R_2) &\equiv (G \multimap R_1) \,\&\, (G \multimap R_2) \\
\forall x.(R_1 \,\&\, R_2) &\equiv (\forall x.R_1) \,\&\, (\forall x.R_2) \\
G \multimap (\forall x.R) &\equiv \forall x.(G \multimap R)^{☆4}
\end{aligned}
$$

さらに，$(R_1 \otimes R_2) \multimap G$ と $R_1 \multimap (R_2 \multimap G)$ が論理的に同値であることから，乗法的論理積 $\otimes$ で結合されたリソースも使用することができる．

以降，R' で定義される論理式をprimitiveリソースと呼び，primitiveリソース中のatomic論理式 A をそのヘッド部と呼ぶ．また，$R \multimap G$，$R \Rightarrow G$ 中に出現するリソース R をそれぞれlinearリソース，exponentialリソースと呼ぶことにする．

プログラム中では，上記のフラグメントに対応した以下のような記法を用いる．

$$
\begin{aligned}
C &::= A. \mid A\texttt{:-}G. \\
G &::= \texttt{true} \mid \texttt{top} \mid A \mid G\texttt{,}G \mid G\texttt{\&}G \mid \\
&\quad G\texttt{;}G \mid \texttt{!}G \mid R\texttt{-<>}G \mid R\texttt{=>}G \\
R &::= A \mid R\texttt{\&}R \mid G\texttt{-<>}R \mid G\texttt{=>}R \mid \\
&\quad \texttt{forall}\ x\texttt{\textbackslash}\ R
\end{aligned}
$$

演算子の優先度は，弱いものから順に“forall”，“\”，“:-”，“;”，“&”，“,”，“-<>”，“=>”，“!”とする．

LLPはLolliのもつ大部分の機能をサポートしているが，以下のものが記述できないという制限がある．

- 節のLinear宣言（使用回数を制限された節の記述）
- 全称記号を含んだゴール $\forall x.G$
- 高階の量化記号とλ項の単一化

☆1 http://bach.seg.kobe-u.ac.jp/llp/

☆2 http://www.cs.hmc.edu/~hodas/research/lolli/

☆3 論文8）で述べたものに $G \multimap R$，$G \Rightarrow R$，$\forall x.R$ が追加されている．

☆4 変数 x が G 中に自由な出現をもたない場合に限る．

3. リソースプログラミング

LLP の最も大きな特徴は，さまざまな「リソースプログラミング」が可能な点である．実行中に動的に追加，削除（消費）されるリソースは，ただ 1 つのリソース表によって管理される．

3.1 リソースの追加

リソースはゴール R-<>G の実行によって追加される．例えば，次の質問はリソース r(1) をリソース表に追加した後，ゴール r(X) を実行する．この質問は成功し，解 X = 1 をもつ．

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

ゴール R-<>G において，リソース R は G の実行中にすべて消費されなければならない．例えば，次の質問ではリソース r(1) が消費されないため失敗する．

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

リソース G-<>A は規則型リソースを表す．リソース中のゴール G は，ヘッド部 A が消費されると同時に実行される．例えば，次の質問は成功して 1 を表示する．

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

リソース R_1,R_2 は，R_1 と R_2 がともにリソースとして利用できることを表す．次の質問はリソース r(1) と r(2) を追加した後，その両方を消費する．この質問は成功し，2 つの解 X = 1, Y = 2 と X = 2, Y = 1 をもつ．

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

次の質問は上とまったく同じ振舞をする．

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

⇒ の左側のリソースは無限のリソースを表し，何度でも使用可能であり，また 1 度も使わなくてもよい．次の質問は成功し，解 X = 1 と X = 2 をもつ．

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

リソース R_1&R_2 はどちらか一方がリソースとして利用できることを表す．例えば，次の質問で r(1)&r(2) が追加されたとき，r(1) か r(2) のどちらか一方だけが消費可能であり，両方を消費することはできない．この質問は成功し，2 つの解 X = 1 と X = 2 をもつ．

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

全称記号の付いたリソースの ⇒ による追加は，Prolog の assert と同じ効果をもつ☆．例えば，次の質問におけるリソースの追加は，節 p(X) :- q(X) を assert するのと同じ意味をもつ．

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

☆ リソースの追加はバックトラック時にキャンセルされる．

3.2 リソースの消費

atomic ゴール A の実行はリソース消費とプログラム呼び出しの 2 つの意味をもつ．これらの選択は非決定的であり，バックトラックによりすべての可能性が実行される．例えば，次の質問は 1 と 2 を表示する．

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

ゴール G_1&G_2 は Prolog の G_1,G_2 と同様であるが，実行の前にリソースがコピーされ，G_1 と G_2 で消費されるリソースは同じものでなくてはならない．例えば，次の質問は，ゴール r(X) と r(Y) が同じリソースを消費しなければならないので，2 つの解 X = Y = 1, Z = 2 と X = Y = 2, Z = 1 をもつ．

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

ゴール !G はゴール G と同様であるが，exponential リソースしか消費できない．次の質問は成功し，解 X = 1, Y = 2 をもつ．

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

ゴール top は，消費可能なリソースのいくつかを消費する．例えば，1 つ目の質問は成功し，2 つ目の質問は，消費可能なリソースが残るため失敗する．

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

3.3 プログラム例

LLP の簡単なプログラム例を 2 つ紹介する．その他のプログラム例に関しては，9) に詳しく述べられている．Lolli のプログラム例については，Hodas と Miller の論文 [2] にいくつか紹介されている．

以下は有向グラフの経路を見つける LLP のプログラム例である．グラフの弧を規則型リソースを使って表すことにより，次のような条件を簡潔に表現している．

- それぞれの弧は高々1 度しか使えない．
- 経路は連結している弧の推移閉包である．

このプログラムは規則型リソースの消費性，及び規則としての利点を十分に生かしており，循環経路を含む場合に生じる問題点もうまく解消している．

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

次のプログラム例では，規則型 exponential リソースを追加することにより，述語 test/1 を一時的に定義している．この述語は filter 中で呼び出され，要素が条件を満たすかどうかチェックするのに使われる．

```
% choose(Xs, Y, Zs)
% Zs is a list of elements greater than Y in Xs.
choose(Xs, Y, Zs) :-
        % define test/1 and then call filter/2
        (forall X \ ( X>Y -<> test(X) )) =>
        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).
```

4. LLP から Java へのトランスレート方式

この節では，LLP，Prolog などの論理型言語を Java 上で効率良く実装するための方法について議論した後，LLP を Java にトランスレートする際に生じる問題点と，その解決法について述べる．

4.1 Java による LLP 処理系の設計

現在 Prolog の実装に関しては，the Warren Abstract Machine (WAM)[10] が事実上の標準なっており，数多くの拡張（高階，並行，制約，線形論理）が提案されている．田村と金田は，論文 11) において，LLP のための拡張 WAM である LLPAM を提案した．また LLPAM の拡張として，LLP のリソースコンパイル方式[12]，LLPAM の理論的基礎とゴール T の実装方式[8] に関する研究が行われた．

また，Prolog から C 言語へのトランスレート方式も盛んに研究されており，論文 13) にいくつかの処理系の実装方法が詳しく述べられている．これらの研究は，Java へのトランスレート方式を考える上で有用なアイデアを含んでいるが，最大の違いは Java には関数ポインタがない点である．

一方，Java 言語の発表以来，Java による論理型言語（特に Prolog）処理系の設計，開発が活発に研究されている．Java の実行速度に関する問題は，近年の JIT (Just-In-Time Compiler)，HotSpot などの開発により徐々に改良されているものの，現時点で十分満足できるものではない．しかしながら，LLP/Prolog 処理系を Java 言語で実装する長所には次のようなものがある．

(1) ネットワークプログラミングが可能な処理系を開発できる．
(2) 豊富な Java 言語のクラスライブラリをそのまま利用できる．
(3) Java のプラットフォーム独立性により，開発した処理系は Java が動作するすべての環境で利用可能となる．

特に (1) の例としては，いくつかの組み込み述語を用意することにより，指定した URL から HTML (XML) のソースファイルをダウンロードし，それらを DCG を使ってパージングすることが可能となる．このような機能は，Java 言語を使えば簡単に実装できるが，他の言語では単純な作業ではない．(2) の Java クラスライブラリに関しては，3D グラフィクス，分散処理など数多く開発されており，今後も増加していくものと予想できる．これらのクラスライブラリの機能を自由に取り込める，Java と強い連結性をもつ LLP/Prolog 処理系を開発することは有用である．例えば，Java の制約解消ライブラリを利用することにより，制約論理型言語に拡張することも可能である．

本稿で述べる処理系の設計を開始したのは 1997 年[14] であり，設計上のキーポイントは拡張性（簡潔性を含む）と効率性（実行速度の高速化）であった．

効率のよい LLP/Prolog 処理系を Java で開発するには，少なくとも以下の 3 つの方式が考えられる．

(1) LLPAM/WAM から Java バイトコードへのコンパイラ
(2) LLPAM/WAM の Java エミュレータ
(3) LLP/Prolog から Java へのトランスレータ

方式 (1) 及び (2) は，当初設定したキーポイントを満足するものではなかった．方式 (1) は最も実行速度が速いと考えられるが，Java バイトコードへのコンパイルは，非常に複雑な作業であり，Java のバージョンに大きく依存する．方式 (2) は最もシンプルなアプローチであるが，巧妙な最適化をしない限り，実行速度の高速化は見込めず，単独の実行可能コードを生成することもできない．

以上のことから，(3) の LLP から Java へのトランスレート方式の研究を進めることにした．この方式は，以下のような長所をもつ．

- 方式 (1) に次いで，実行速度の高速化が可能
- エミュレータにおけるオーバーヘッドの回避
- 単独で実行できるコードの生成

Prolog に関しては，B. Demoen と P. Tarau によっ

て jProlog☆ と呼ばれるトランスレータが開発されている．jProlog はバイナリ変換（binarization transformation）[15] と呼ばれるコンパイル方式に基づいている．jProlog のトランスレート方式は非常に有用であり，実行速度に関しても，かなりの高速化が実現されているが，いくつか改良すべき点も残っている．主要な改良点は以下の通りである．

- インデキシング（indexing）の実装
- ヘッド部引数における単一化のコンパイル
- 継続ゴール（continuation）のコンパイル

4.2 Prolog のトランスレート

項は VariableTerm，IntegerTerm，SymbolTerm，StructureTerm，ListTerm のいずれかのクラスのオブジェクトにトランスレートされる．Term クラスはすべての項のスーパークラスであり，unify メソッドをもつ．

節はバイナリ変換[15]によってすべてバイナリ節に変換された後，述語ごとにクラスの集合にトランスレートされる．クラスの集合は，述語呼び出しに使われるただ1つのクラス（以降，エントリポイントと呼ぶ）と節，選択命令，インデキシング命令を表すクラスから構成される．Predicate クラスはすべての述語のスーパークラスであり，exec メソッドと engine フィールドをもつ．engine フィールドは，実行時に活性化されている LLP エンジン☆☆を格納するために使われる．また，engine.aregs，engine.cont フィールドはそれぞれ引数レジスタ，継続ゴールレジスタを表す．

以下，次の簡単なプログラムを用いてトランスレート例を示す．

```
% Source clauses
p(X, Y) :- q(X), r(Y).
p(X, X).

% Binary clauses
p(X, Y, Cont) :- q(X, r(Y, Cont)).
p(X, X, Cont) :- true(Cont).
```

最初に，各節（*Source clauses*）はバイナリ変換によりバイナリ節（*Binary clauses*）に変換される．その際，継続ゴールが最後の引数として加えられるため，述語 p/2 のアリティは 1 増える．その後，述語 $p/2$ (バイナリ述語 $p/3$) はクラスの集合にトランスレートされる．図 1 は述語 $p/2$ のエントリポイントを示す．

```
public class PRED_p_2 extends Predicate{
  ...
  // constructor of predicate p/2
  public PRED_p_2(Term _a1,
                  Term _a2,
                  Predicate _cont){
      arg1 = _a1;
      arg2 = _a2;
      cont = _cont;
  }
  public Predicate exec(){
      engine.aregs[1] = arg1;   // set X
      engine.aregs[2] = arg2;   // set Y
      engine.cont = cont;       // set Cont
      engine.setB0();           // set cut point
      return the code for choice instruction;
  }
}
```

図 1 述語 $p/2$ のエントリポイント
Fig. 1 Code for an entry point of $p/2$.

ゴールはエントリポイントのオブジェクトとして表される．例えば，ゴール p(1,Z) は図 1 のコードを使って，次のように作成される (co は p(1,Z) が成功した後で実行される継続ゴールを表す)．

```
Term t1 = new IntegerTerm(1); // create 1
Term t2 = new VariableTerm(); // create Z
// create a goal p(1,Z)
Predicate p = new PRED_p_2(t1, t2, co);
```

述語 $p/2$ が実行されると，図 1 中の exec メソッドは引数 X，Y と継続ゴール Cont の値をレジスタ aregs と cont にそれぞれ格納し，WAM の選択命令 "*try L*" に似たコードを返す．

一方，選択命令，またはインデキシング命令から，実行の制御が節に移った場合，エントリポイントでレジスタに格納した値を取り出す必要がある．そのため，述語 $p/2$ の 1 番目の節のコード（図 2）は，レジスタ aregs から引数 X，Y の値を，レジスタ cont から継続ゴール Cont の値をそれぞれ取り戻し，次に実行すべきゴールのコードを返す．

以上，述語のトランスレート例を説明してきたが，ここで継続ゴールの実装方法について議論する．バイナリ変換よる継続ゴールの実装方法には，いくつか考えられる．jProlog では，継続ゴールを項オブジェクトにトランスレートし，継続ゴールを実行する度に述語表（ハッシュ表）を通じて，項から対応するエントリポイントを検索する方法を用いている．すなわち，

☆ http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/
☆☆ 引数レジスタ，trail スタックなどを含む実行環境

```
final class Clause1 extends PRED_p_2{
  public final Predicate exec(){
      ...
      // retrieve X
      a1 = engine.aregs[1].dereference();
      // retrieve Y
      a2 = engine.aregs[2].dereference();
      // retrieve Cont
      cont = engine.cont;
      // create r(Y,Cont)
      p1 = new PRED_r_1(a2, cont);
      // create and return q(X,r(Y,Cont))
      return new PRED_q_1(a1, p1);
  }
}
```

図 2 節 p(X,Y) :- q(X),r(Y). のコード
Fig. 2 Code for the 1st clause p(X,Y) :- q(X),r(Y).

```
public final class RES_p_2 extends Predicate{
  ...
  public final Predicate exec(){
      ...
      a1 = engine.aregs[1].dereference();
      a2 = engine.aregs[2].dereference();
      // set the value of X to a3
      a3 = engine.aregs[3].dereference();
      cont = engine.cont;
      // unify the 1st argument with X
      if ( !a1.unify(a3, engine.trail) )
          return engine.fail();  // backtrack
      return new PRED_q_2(a1, a2, cont);
  }
}
```

図 3 リソース $\forall Y.(q(X,Y) \multimap p(X,Y))$ のコード
Fig. 3 Code for the resource $\forall Y.(q(X,Y) \multimap p(X,Y))$.

継続ゴールはコンパイルされておらず，順次インタープリットされ実行される．この方法のメリットは，一度生成した述語オブジェクトを述語表を使って再利用できる点である．本稿では，継続ゴールを述語オブジェクトにトランスレートするため，生成されたコードは以下の簡潔なループによって実行される．しかし，jProlog のように一度生成した述語オブジェクトを再利用することはできない．

```
Predicate code;
Set the target predicate to code;
while (code != null){
    Set code.engine to point to
    the current LLP engine;
    code = code.exec();
}
```

継続ゴールに関して，本稿の方法と jProlog の方法の処理速度はほぼ同等である．これは継続ゴールを表す項オブジェクトを生成し，ハッシュ表を使って対応する述語オブジェクトを検索，実行するコストと，述語オブジェクトを毎回生成し，実行するコストが同程度になっているためと考えられる．しかし，今後の拡張性を考えた上，継続ゴールを述語オブジェクトにトランスレートする方法を選択した．初期の実装では，継続ゴールを return することなく，順次入れ子で実行していく方法も試みた．しかし，この方法はプログラムによって Java のスタックがすぐにオーバーフローしてしまうという欠点がある．

4.3 リソースのコンパイル

すべてのリソースは primitive リソースに分解されることから，この節では，primitive リソースのコンパイル方式について述べる．

第 3 節で述べたように，LLP ではさまざまな「リソースプログラミング」が可能である．例えば，規則型リソース G-<>A がリソース表に追加されたとき，リソース中のゴール G は A が消費されると同時に実行される．初期の実装では，リソース G-<>A はコンパイルされず，項としてリソース表に格納されていた．このため A のリソース消費は組み込みの単一化関数，ゴール G の実行はメタコールによりそれぞれ実装され，リソースの実行速度を低下させる原因となっていた．この問題を解消し，効率のよい「リソースプログラミング」を実現するために，リソースのコンパイルは非常に重要である．

自由変数を含まないリソースのコンパイルは容易である．これらのリソースは自由変数の束縛情報を必要としないので，節とまったく同じ方法でトランスレートできる．例えば，リソース $\forall X.\forall Y.(q(X,Y) \multimap p(X,Y))$ は節 $p(X,Y)$:- $q(X,Y)$ と同様にトランスレートされる．

自由変数☆を含むリソースのコンパイルは容易でない．例えば，リソース $\forall Y.(q(X,Y) \multimap p(X,Y))$ において，変数 X は自由な出現である．このリソースを消費するためには，実行時に X がどのような値に束縛されているかを知らなければならない．

この問題を解決するために，新しいデータ構造クロージャを導入する．クロージャ構造はコンパイルコードへの参照と自由変数の束縛情報から構成される．

クロージャが実行されると，まず自由変数が使用可能な引数レジスタにセットされ，その後コンパイル

☆ ここではリソース中に出現する変数が，そのリソース中で束縛されていないとき，リソースの自由変数を呼ぶ

コードが実行される．そのため，生成されるリソースのコードは，自由変数の値を取り出す命令を含む必要がある．例えば，リソース $\forall Y.(q(X,Y) \multimap p(X,Y))$ のコード(図 3)は，第 3 引数レジスタ（aregs[3]）に格納されている自由変数 X の値を取り出し，第 1 引数と単一化している（さらに複雑なリソースのトランスレート例は第 5 節で示す）．

クロージャ構造は ClosureTerm クラスによって簡単に表現できる．例えば，図 3 のリソース $\forall Y.(q(X,Y) \multimap p(X,Y))$ のクロージャは次のように作成される．ただし，varX は自由変数 X を表すものとする．

```
Predicate res = new RES_p_2();
Term[] var = {varX};
ClosureTerm c = new ClosureTerm(res, var);
```

G. Nadathur は論文 16) において，λProlog のゴール $D \supset G$ をコンパイルするのに同様のアイデアを用いている．しかし，λProlog には LLP の $\multimap$ に相当する演算子がないことから，そのクロージャの取扱いは異なる．

本稿では，LLP から Java に変換することを意味する言葉として「トランスレート」という用語を使用しているが，リソースをクロージャに変換することを意味する言葉としては，「リソースのコンパイル」という用語を使用するものとする．

4.4 LLP のトランスレート

この節では，Prolog にはない LLP の演算子 "-<>", "=>", "&", "!" のトランスレート方法について述べる．

ゴール R-<>G，R=>G，G_1&G_2，!Gは，バイナリ変換される前に各演算子ごとに対応する命令列に展開される．この展開方法は，論文 8)，12) に述べられており，本稿で詳細は述べない．

以下にゴール R-<>Gの展開例を示す．

```
% LLP の節
p(X, Y) :- q(X) -<> r(Y).
% 展開後の節
p(X, Y) :-
        begin_imp(A),
        add_res(q(X), [q/1,0,[X]]),
        mid_imp(B),
        r(Y),
        end_imp(A, B).
```

例中の add_res/2 の第 2 引数 [q/1,0,[X]] は，追加されるリソースのクロージャを表しており，第 1 要素はリソースのファンクタ（述語記号とアリティ），第 2 要素はリソースの通し番号，第 3 要素はリソース中に出現する自由変数を示している．

ゴール R=>G，G_1&G_2，!Gは，以下のように展開される．

```
% LLP の節
p(X, Y) :- q(X) => r(Y).
% 展開後の節
p(X, Y) :-
        begin_exp_imp(A),
        add_exp_res(q(X), [q/1,0,[X]]),
        mid_exp_imp(B),
        r(Y),
        end_exp_imp(A, B).

% LLP の節
p(X, Y) :- q(X) & r(Y).
% 展開後の節
p(X, Y) :-
        begin_with,
        q(X),
        mid_with,
        r(Y),
        end_with.

% LLP の節
p(X, Y) :- !((q(X), r(Y))).
% 展開後の節
p(X, Y) :-
        begin_bang,
        q(X),
        r(Y),
        end_bang.
```

展開された節は，Prolog の節と同様にバイナリ変換された後，Java にトランスレートされる．R-<>Gのコード生成については，次節でさらに詳しく述べる．

5. Prolog Café: LLP から Java へのトランスレータシステム

この節では，前節のトランスレート方式に基づいた LLP 処理系（Prolog Café）について述べる．

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

図 4 PrimRes クラス（リソース表のレコード構造）
Fig. 4 The PrimRes class (the record of resource table).

5.1 リソース表

リソース表 RES は PrimRes クラス（図 4）の配列である．RES は $\multimap$，$\Rightarrow$ によってリソースが追加されると増加し，バックトラックにより減少する．RES の各エントリはただ 1 つの primitive リソースに対応する．

level，deadline，isOutOfScope フィールドはリソース消費のために使われる．その使用法については論文 8) に述べられおり，本論文では詳細は述べない．

rellist フィールドは & で結合されたリソース（関連リソース）の相対位置を表すものであり，その使用法については論文 11) に述べられおり，本論文では詳細は述べない．

functor，closure フィールドは格納されるリソースの情報を表す．functor フィールドには，リソースのヘッド部のファンクタ（述語記号とアリティ）が，closure フィールドには，リソースのクロージャ構造が格納される．

5.2 $R \multimap G$ のコード

リソース R はゴール $R \multimap G$（または $R \Rightarrow G$）によって追加される．ゴール $R \multimap G$ の実行概要は以下の通りである．

(1) 現在のリソース表（RES）の底を示す top の値を変数 X_i に保存しておく．

(2) 論理式 R 中のすべての primitive linear リソース（n 個とする）をリソース表に追加する．また，これらのエントリをハッシュ表に追加する（述語記号と第 1 引数をキーとする）．

(3) リソース追加後の top の値を変数 Y_i に保存しておく（$Y_i = X_i + n$）．また，追加された primitive linear リソース（すなわち RES[X_i] から RES[$Y_i - 1$]）の関連リソースの相対位置（rellist フィールドの値）を生成する．

(4) ゴール G を実行する．

(5) 論理式 R 中にあったすべての primitive linear リソース（すなわち RES[X_i] から RES[$Y_i - 1$]）が消費されているかどうかチェックする．

(6) 論理式 R 中にあったすべての primitive linear リソースの isOutOfScope フラグを true にする（これはその後の実行で消費不可能にするためである）．

primitive リソースの追加には，以下の組込み述語が使用される．

- PRED_begin_imp_1(Term a1,Predicate cont)
  ステップ (1) に対応し，リソース表（RES）の底を示す top の値を第 1 引数 a1 に保存する．
- PRED_add_res_2(Term a1,Term a2,Predicate cont)
  ステップ (2) に対応し，primitive linear リソースをリソース表 RES に追加する．第 1 引数 a1 はリソースのヘッド部の項であり，第 2 引数 a2 はリソースのクロージャである．a1 のファンクタは functor フィールドに，a2 は closure フィールドにそれぞれ格納され，RES の底を示す top の値が 1 だけ増加する．

```
PrimRes res = new PrimRes();
Set level, deadline, and
isOutOfScope fields;
res.functor = functor of a1;
res.closure = (ClosureTerm) a2;
Record this entry in the hash
and symbol tables;
RES[top++] = res;
```

- PRED_mid_imp_1(Term a1, Predicate cont)
  ステップ (3) に対応し，リソース表（RES）の底を示す top の値を第 1 引数 a1 に保存し，追加された primitive linear リソースの関連リソースの相対位置（rellist フィールドの値）を生成する．
- PRED_end_imp_2(Term a1, Term a2, Predicate cont)
  ステップ (5,6) に対応し，追加された primitive linear リソース（RES[a1] から RES[a2 − 1]）が消費されているかどうかチェックする．すべて消費されていれば，それらのリソースの isOutOfScope フラグを true にする（これはその後の実行で消費不可能にするためである）．
- PRED_begin_exp_imp_1(Term a1, Predicate cont)
  PRED_begin_imp_1 と同様
- PRED_add_exp_res_2(Term a1,Term a2, Predicate cont)
  primitive exponential リソースをリソース表に追加する．level，deadline フィールドに primitive exponential リソース用の値が代入されることを除けば，PRED_add_res_2 と同じである．

```
static Predicate resP = new RES_p_2();
static SymbolTerm symP = symbol p/2;
public final Predicate exec(){
    ...
    Term[] args = {varX, new VariableTerm()};
    // create head p(X,Y)
    a3 = new StructureTerm(symP, args);
    Term[] vars = {varX};
    // create closure
    a4 = new ClosureTerm(resP, vars);
    p1 = new PRED_end_imp_2(a2, a5, cont);
    p2 =  Code for the goal G;
    p3 = new PRED_mid_imp_1(a5, p2);
    // add ∀ Y((q(X) —o r(Y)) —o p(X,Y))
    p4 = new PRED_add_res_2(a3, a4, p3);
    return new PRED_begin_imp_1(a2, p4);
}
```

図 5 ゴール $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y)) \multimap G$ のコード
Fig. 5 Code for the goal $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y)) \multimap G$.

- `PRED_mid_exp_imp_1(Term a1, Predicate cont)`
  関連リソースの相対位置（rellist フィールドの値）を生成する必要がないことを除けば，`PRED_mid_imp_1` と同様である．
- `PRED_end_exp_imp_2(Term a1,Term a2, Predicate cont)`
  追加された primitive exponential リソースの `isOutOfScope` フラグを true にする．exponential リソースは，何度でも使用可能なので消費チェックをする必要はない．

リソースへのアクセスを高速化するために，ハッシュ表を用いる．`PRED_add_res_2` と `PRED_add_exp_res_2` の第 1 引数にヘッド部のファンクタではなく項全体が使われているのは，追加されるリソースのハッシュキーを求めるためである．今のところ，述語記号と第 1 引数の値をキーとして使用しているが，今後の改良に備えて項全体を送ることにした．

リソースへのアクセスをハッシュ表だけに頼ることはできない．第 1 引数が未代入の変数であるゴールを実行する場合，そのファンクタをもつすべてのリソースにアクセスしなければならない．同様に，追加された時に第 1 引数が未代入の変数であったリソースも，そのファンクタをもつゴールが実行される度に，アクセスされなければならない．このため，シンボル表の各エントリは 2 つのリストをもっている．1 つは，そのファンクタをもつリソースすべてのインデックス値のリスト，もう 1 つは，そのファンクタをもち，追加時に第 1 引数が未代入の変数であったリソースすべてのインデックス値のリストである．

図 5 はゴール $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y)) \multimap G$ のコードであり，図 6 はこのゴールによって追加されるリソースのコードである．ただし，varX は変数 X を表すものとする．

```
public final class RES_p_2 extends Predicate{
  static Predicate resQ = new RES_q_1();
  static SymbolTerm symQ = symbol q/1;
  ...
  public final Predicate exec(){
      ...
      a1 = engine.aregs[1].dereference();
      a2 = engine.aregs[2].dereference();
      // retrieve X
      a3 = engine.aregs[3].dereference();
      this.cont = engine.cont;
      // unify the 1st argument with X
      if ( !a1.unify(a3, engine.trail) )
          return engine.fail();
      ...
      Term[] args = {a1};
      // create head q(X)
      a5 = new StructureTerm(symQ, args);
      Term[] vars = {a1};
      // create closure
      a6 = new ClosureTerm(resQ, vars);
      p1 = new PRED_end_imp_2(a4, a7, cont);
      p2 = new PRED_r_1(a2, p1);
      p3 = new PRED_mid_imp_1(a7, p2);
      // add q(X)
      p4 = new PRED_add_res_2(a5, a6, p3);
      return new PRED_begin_imp_1(a4, p4);
  }
}

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

図 6 リソース $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y))$ のコード
Fig. 6 Code for the resource $\forall Y((q(X) \multimap r(Y)) \multimap p(X,Y))$.

5.3 Atomic ゴールの実行

LLP では，atomic ゴールはリソース消費と述語呼出しを意味する．ファンクタ p/n をもつ atomic ゴール A の実行概要は以下の通りである．

(1) 消費できる可能性のある primitive リソースのインデックス値のリストをハッシュ表とシンボル表から得て（述語記号と第 1 引数をキーとして検索する），それぞれのリストをレジスタ R1

```
public class PRED_p_2 extends Predicate{
  static SymbolTerm functor = functor p/2;
  static Predicate L = ordinary code for p/2;
  static Predicate L0 = new L0();
  static Predicate L1 = new L1();
  static Predicate L2 = new L2();
  ...
  public Predicate exec(){
      ...
      engine.lookUpHash(functor);
      if (!engine.pickupResource(functor, 3))
          return L;
      return engine.tryResource(L1, L0);
  }
}

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

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

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

図 7 atomic ゴール $p/2$ のコード
Fig. 7 Code for an atomic goal $p/2$.

と R2 に格納する.

(2) R1, R2 中でファンクタ p/n をもつ各リソースエントリ R に対して, 以下を試みる.
  (a) R がスコープ外☆であるか, R が linear リソースですでに消費されている場合, 失敗する.
  (b) エントリ R に消費されたことを示すマークを付ける.
  (c) R のクロージャを実行する.

(3) すべての試みが失敗した後, ゴール A の通常のコードを呼び出す.

図 7 はファンクタ $p/2$ をもつ atomic ゴール A' のコードである. 以下, 図 7 の動作を詳しく説明する.

☆ $R \multimap G$ によって追加された R は, G 中でのみ消費可能であり, 他のゴールは消費できない.

ゴール A' の exec メソッドが実行されると, まず, 第 1 引数, 第 2 引数, 継続ゴールがそれぞれのレジスタに格納される.

lookUpHash メソッドは実行概要 (1) の手続きに対応しており, レジスタ R1, R2 にそれぞれリストをセットする. R1 はヘッド部の述語記号が同じであり, 第 1 引数に同じ値をもつリソースのインデックス値を含み, R2 はヘッド部のファンクタ $p/2$ であり, 追加時に第 1 引数が未代入の変数であったリソースのインデックス値を含む.

pickupResource メソッドは R1 (R1 が空の場合は R2) からファンクタが $p/2$ でかつ消費可能なリソースのインデックス値を 1 つ探し出し, その値を第 3 引数レジスタに格納する. さらに, R1 と R2 を未使用のリソースに更新しておく. メソッド実行時に, 消費可能なリソースがなければ, ただちにゴール A' の通常のコードを呼び出す.

tryResource メソッドは, WAM 命令の *"try L"* と同じであるが, レジスタ R1, R2, cont (継続ゴール) の値も選択点フレームに保存する. また, 選択点フレームの BP フィールド (next clause) に L0 をセットし, L1 を返す.

実行の制御が L1 に移ると, consume メソッドが第 3 引数レジスタに格納されたインデックス値をもつリソースに消費マークを付け, そのクロージャを取り出す. 続いて, executeClosure メソッドは抽出されたクロージャの自由変数を使用可能な引数レジスタにセットし, コンパイルコードを返す (コードは外側のループでただちに実行される).

リソース消費の 1 つの試みが失敗すると, 実行の制御は L0 に移る (tryResource において, BP が L0 にセットされているため). restoreResource メソッドによって選択点フレームの値がすべて取り戻された後, 再び pickupResource メソッドが呼び出される. retryResource メソッドは, 選択点フレーム中の R1 と R2 の値を最新の値に更新し, 再びリソース消費を行う L1 を返す. pickupResource メソッドの実行時に消費可能なリソースがなければ, 制御は L2 に移る.

L2 において, trustResource メソッドは現在の選択点フレームを削除し, 制御をゴール A' の通常のコードに渡す.

次に, atomic ゴールの最適化について述べる. 図 7 のコードでは, プログラム中にリソースがまったくない場合でも, ゴールの実行のたびにリソース消費が可能かどうか試す必要があり, これに伴うオーバーヘッドは無視できない.

この問題を解決するために，新しくリソース宣言を導入する．リソース宣言は，Prolog のダイナミック宣言（:- dynamic p/n）と同じような効果をもつ．例えば，リソース宣言:- resource p/n は他の述語がファンクタ p/n をもつリソースを追加したり，消費したりできることを意味する．すなわち，リソース宣言されていない述語に関しては，リソース消費を行うコードは生成されない．

リソース宣言がその威力を発揮するのは，通常のプログラム呼び出しのない述語に適応された場合である．この場合，atomic ゴールはリソース消費のみを意味し，本稿で述べる最適化は，この場合に限られる．最適化のアイデアは以下の通りである．

- 消費可能なリソースがただ 1 つしか存在しない場合，選択点フレームを作る必要はなく，そのリソースを消費するだけでよい．
- 最後の消費可能なリソースを消費する前に，選択点フレームを削除しても安全である．

atomic ゴールの最適化コードを生成するために，以下の新しいメソッドが使われる．

- public final boolean hasMoreResource()
  レジスタ R1，R2 の中に消費可能なリソースが存在するかどうか調べ，もし存在しなければ失敗する．

上記の最適化のアイデアは，hasMoreResource メソッドを pickupResource メソッドの直後に挿入することにより，簡単に実現できる．図 8 はファンクタ $p/2$ をもつ atomic ゴール（図 7）の最適化コードである．ただし，図 7 と重複する部分は省略している．

5.4 assert と retract の実装

Java のハッシュ表を使った assert と retract の実装について述べる．実装をより簡単にするために，以下の特別な組み込み述語を用いる．

- put_term(+Key, ?Term)
  第 1 引数の Key を第 2 引数 Term の値にマップする．Key は基底的（ground）な項でなければならず，Term 中の変数はそのまま（コピーされることなく）ハッシュ表に格納される．
- get_term(+Key, ?Term)
  Key にマップされている値を取り出し，Term と単一化する．キーにマップされている値がなければ，Term は空リストと単一化される．

ハッシュ表の各エントリは節のリストを表し，キーにはヘッド部の述語記号とアリティを使用する．

節 C の assert の実行概要は以下の通りである．

(1) ハッシュ表から，節 C のキーにマップされている節のリストを抽出する．

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

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

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

図 8 $p/2$ の最適化コード（通常の述語呼び出しがない場合に限る）
Fig. 8 Optimized code for $p/2$ when there is no ordinary predicate invocation.

(2) 抽出したリストに節 C を加えて，新しいリスト L を作成する．

(3) リスト L のコピーをハッシュ表に登録する．

上記の処理のうち，(3) においてコピーを取る必要がある．それは，節 C が assert された後で，C 中の変数への代入が起こる可能性や，また (1) – (3) での変数束縛が，バックトラックによってすべてキャンセルされる可能性があるからである．

次に節 C の retract の実行概要を示す．

(1) ハッシュ表から，節 C のキーにマップされている節のリストを抽出する．

(2) 抽出したリストのコピー L を作成する．

(3) L の各要素である節 C' に対して，以下を試みる．

(a) 節 C' と節 C が単一化可能でなければ失敗する．

(4) L の未試行の部分をハッシュ表に登録する．

retract の処理に関しても，同様の理由で (2) において抽出したリストのコピーを取る必要がある．

付録 A.2 に assert と retract のソースコードを示す．

表 1 Prolog ベンチマークの実行結果 (実行回数 5)
Table 1 Execution time of classical Prolog benchmarks (#Runs Avged.= 5).

Program	SICStus 2.1 WAM code	SWI 3.2.6 WAM code	MINERVA 2.0 JDK1.1, no-JIT	Prolog Café 0.42 JDK1.1, no-JIT	jProlog 0.1 JDK1.1, no-JIT
boyer	446 ms	2,867 ms	27,627 ms	81,401 ms	261,699 ms
	(1.0)	(6.4)	(61.9)	(182.5)	(586.8)
browse	559 ms	2,280 ms	19,469 ms	65,245 ms	390,272 ms
	(1.0)	(4.1)	(34.8)	(116.7)	(698.2)
chat_parser	30 ms	67 ms	329 ms	1,462 ms	2,148 ms
	(1.0)	(2.2)	(11.0)	(48.7)	(71.6)
nrev	20 ms	180 ms	545 ms	901 ms	4,612 ms
(300 elements)	(1.0)	(9.0)	(27.3)	(45.1)	(230.6)
poly_10	36 ms	157 ms	1,059 ms	2,425 ms	4,156 ms
	(1.0)	(4.4)	(29.4)	(67.4)	(115.4)
queens_12	10 ms	90 ms	623 ms	710 ms	1,727 ms
(first solution)	(1.0)	(9.0)	(62.3)	(71.0)	(172.7)
queens_16	545 ms	5,200 ms	36,355 ms	40,791 ms	100,592 ms
(first solution)	(1.0)	(9.5)	(66.7)	(74.8)	(184.6)
queens_20	15,009 ms	148,943 ms	1,037,970 ms	1,145,028 ms	2,850,569 ms
(first solution)	(1.0)	(9.9)	(69.2)	(76.3)	(189.9)
reducer	36 ms	97 ms	898 ms	2,591 ms	8,204 ms
	(1.0)	(2.7)	(24.9)	(72.0)	(227.9)
zebra	40 ms	60 ms	712 ms	1,223 ms	2,214 ms
	(1.0)	(1.5)	(17.8)	(30.6)	(55.4)
average ratio	(1.0)	(5.9)	(40.5)	(78.5)	(253.3)

6. 性能評価

本稿で述べた LLP 処理系 (Prolog Café) は，トランスレータ (SICStus Prolog で記述) と LLP システム (Java で記述) から構成されている．また，現在の最新版 (0.42) ではブートストラップも完了している．全ソースコードを含む Prolog Café の最新のパッケージは，以下の Web サイトから入手可能である．

http://pascal.seg.kobe-u.ac.jp/~bambara/PrologCafe/

この節では，標準的な Prolog ベンチマーク，及び N クイーン配置の全解探索問題を解く LLP プログラムを用いて，Prolog Café の性能評価を行う．時間はすべて Vine Linux (MMX Pentium 266MHz, 128MB Memory) 上で測定した．Java (no-JIT) には Linux JDK 1.1.7 を使用した．

6.1 Prolog ベンチマークを用いた Prolog Café の性能評価

表 1 は，Prolog 処理系の性能比較によく用いられる Prolog ベンチマークの実行結果である．表 1 の最下行は SICStus Prolog の実行速度を 1.0 とした場合の各プログラムの比率の平均を表している．

Prolog ベンチマークには表 1 に挙げたもの以外にもたくさんある．今回は代表的な 28 種類の Prolog プログラムを使用したが，SICStus Prolog での実行速度が微量 (実行回数 5 回の内，少なくとも 1 回の実行結果が 0 ms) であったものは載せていない．Prolog Café は recorda/3, recorded/3, keysort/2 などの組み込み述語をまだサポートしていないため，nand と simple_analyzer の 2 つが実行できなかった．jProlog も組み込み述語不足のため，いくつかのベンチマークが実行できなかった．また，jProlog は (IF->THEN;ELSE)，DCG を直接処理できないため，ベンチマークには多少コードを書き換えたものを使用した．

以下に比較に用いた Prolog 処理系を挙げる．

- SICStus Prolog 2.1 (commercial, Swedish Institute of Computer Science) ☆
  現在最もポピュラーな商用コンパイラであり，その実行速度は非常に高速である．Prolog プログラムは，WAM コードにコンパイルされ，エミュレータにより実行される．Sparc (Sun-4)，MC68020 (Sun-3, NeXT) プラットホームでは，ネイティブコードコンパイルをサポートしている．本稿で使用したバージョン 2.1 は最新版 (3.7.1) ではないが，その実行速度に大きな差はない．
- SWI-Prolog 3.2.6 (free, J. Wielemaker) ☆☆
  SWI の大きな特徴はコンパイルの速さと，充実した組み込み述語である．実行速度では SICStus Prolog，BinProlog に劣るものの，非常に知名度

☆ http://www.sics.se/isl/sicstus.html
☆☆ http://www.swi.psy.uva.nl/projects/SWI-Prolog/

表 2 N クイーン配置の全解探索問題実行結果 (実行回数 5)
Table 2 Execution time of N-Queen puzzle (all solutions, #Runs Avged.= 5).

N	LLP 0.43 (LLP) LLPAM code	Prolog Café 0.42 (LLP) JDK 1.1, no-JIT	MINERVA 2.0 (Prolog) JDK1.1 no-JIT
8	50 ms	2,396 ms	2,276 ms
	(1.0)	(47.9)	(45.5)
9	226 ms	9,952 ms	10,868 ms
	(1.0)	(44.0)	(48.1)
10	1,006 ms	44,336 ms	54,437 ms
	(1.0)	(44.1)	(54.1)
11	4,912 ms	217,474 ms	294,408 ms
	(1.0)	(44.3)	(59.9)
12	25,754 ms	1,150,125 ms	1,711,187 ms
	(1.0)	(44.7)	(66.4)

の高いアカデミック Prolog コンパイラである.

- MINERVA 2.0 (commercial, IF Computer) ☆ Java 言語による商用の Prolog コンパイラであり, Prolog プログラムを独自の抽象機械コードにコンパイルし, それを Java で実行する方式をとっている. 現時点で最も高速な Java による Prolog 処理系の 1 つである.
- jProlog 0.1 (free, B. Demoen and P. Tarau) ☆☆

  1996 年に開発された Prolog から Java への最初のトランスレータであり, Prolog Café を開発する上でベースとなったトランスレータである.

Prolog Café の実行速度は, jProlog と比較して, 平均で約 3.2 倍速い. MINERVA と比較した場合, ベンチマークにより同程度のものから 4.4 倍遅いものまであるが, 平均で約 2 倍遅くなっている. 表 1 の boyer の実行速度が他のベンチマークに比べて大きく遅いのは, WAM 命令の *switch_on_constant* に相当するものが実装されていないためと考えられる.

また, Prolog Café は, SICStus, SWI と比較して, それぞれ 78.5 倍, 13.3 倍遅くなっている. Prolog Café の JIT を使った実行速度は, ベンチマークによって多少の差はあるが, no-JIT より平均で 2.2 倍速くなる. よって JIT の効果を考慮すれば, SICStus, SWI より, それぞれ 35 倍, 6 倍遅い程度に抑えられる.

6.2 LLP プログラムを用いた Prolog Café の性能評価

第 3 節で述べたように LLP は, Prolog と比較して, さまざまな「リソースプログラミング」が可能である. ここでは, N クイーン配置の全解探索問題を解く LLP プログラム (付録 A.1 参照) をベンチマークとして, Prolog Café を用いて本稿のトランスレート方式 (特に, リソース処理) の評価を行った.

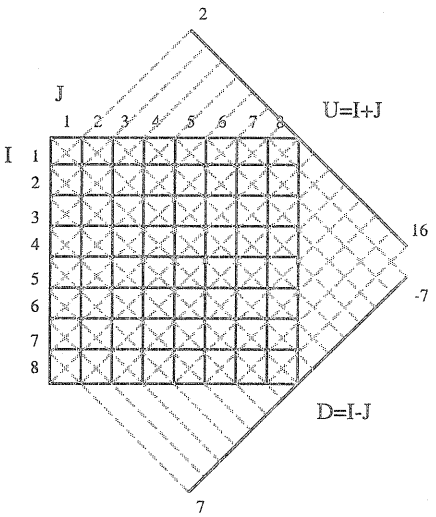

図 9 8-queen program におけるリソース
Fig. 9 Resources in 8-queen program.

使用した LLP プログラムは, リソースの消費性の利点 (一度消費されたリソースは二度と消費できないという特徴) をうまく生かしている. このプログラムは, 各列, 各右上がりのライン, 各右下がりのラインがリソース c/1, u/1, d/1 に対応しており, クイーンを配置する時に対応するリソースを消費する. 例えば, i 行 j 列に配置する場合, リソース c(j), u($i+j$), d($i-j$) を消費する. よって, クイーンがお互いに攻撃し合わないかどうかというチェックは, リソースが消費可能かどうかのチェックにより自動的に行われる (図 9 参照).

比較対象としては以下の 2 つを用いた.

(1) Prolog ベンチマーク中の N クイーン問題を解く Prolog プログラム (付録 A.1 参照) を MIN-

☆ http://www.ifcomputer.com/MINERVA/
☆☆ http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/

ERVA でコンパイルしたもの.

(2) 同じ LLP プログラムを線形論理型言語で初のコンパイラ処理系である LLP でコンパイルしたもの.

- LLP 0.43 (free, N. Tamura et al.) ☆
  現在最も高速な線形論理型言語のコンパイラ処理系である. LLP は, LLP プログラムを拡張 WAM である LLPAM にコンパイルし, それをエミュレータで実行するという方式をとっている. Prolog プログラムもそのまま実行可能であり, その実行速度は, SICStus Prolog と比較して 2〜3 倍遅い程度である.

(1) に関して, MINERVA は表 1 の Prolog ベンチマークにおいて, Java 言語で実装された処理系の中で最も速く, Prolog Café より平均で約 2 倍の高速化を実現している. 使用した Prolog プログラムは「Prolog の技芸」にある N クイーンプログラムとほぼ同様である. このプログラムでは, クイーンは, `select/3` によって 1 つ 1 つ配置され, クイーンがお互いに攻撃し合わないかどうかというチェックは, `not_attack/2` によって行われる. すなわち, これらはすべてリスト構造への操作である.

(2) に関して, その他の線形論理型言語の処理系は, 大部分がインタプリタであり, 高速な処理が可能なものはまだ少ないと言える.

MINERVA と比較した場合, Prolog Café による LLP プログラムの実行速度は, $N = 8$ でほぼ同程度, N が大きくなるにつれて速くなり, $N = 12$ で約 1.5 倍速くなる (表 2 参照). また, $N > 12$ ではこの差がさらに大きくなる. これはハッシュを使ったリソースの処理速度と逐次的なリスト構造の処理速度の差によるものであり, Prolog Café は, リソースの性質を十分生かしたプログラムに対して, 効率のよい実行が可能であると言える.

LLP と比較した場合, Prolog Café の実行速度は約 45 倍遅くなっている (表 2 参照). この LLP プログラムに関して, Prolog Café の JIT を使った実行速度は, no-JIT より約 3.6 倍速くなる. よって JIT の効果を考慮すれば, LLP より約 12 倍遅い程度に抑えられる.

7. おわりに

本稿で述べた処理系 Prolog Café, 比較に用いた jProlog, MINERVA を含め, 近年 Java 上の Prolog 処理系が数多く開発されている.

Prolog インタプリタには, BirdLand's Prolog in Java, CKI Prolog, DGKS Prolog, JavaLog, Jinni, JP, LL などがある. BinNet Corp. が開発した Jinni☆☆ は, 軽量でマルチスレッドにも対応した Prolog インタプリタであり, ネットワークプログラミングが可能である. M. Winikoff が開発した W-Prolog☆☆☆ は GUI を備え, WWW ブラウザ上でアプレットとしても使うことのできる Prolog インタプリタである.

本稿では線形論理型言語 LLP から Java へのトランスレート方式を提案し, その方式に基づいた LLP 処理系 Prolog Café の性能評価を行った. Prolog ベンチマークの実行速度は, 平均で jProlog より約 3 倍速く, MINERVA より約 2 倍遅いという結果を得た. また, N クイーン配置問題を解く LLP プログラムの実行速度は, $N = 12$ で, 標準的な Prolog ベンチマークに含まれる N クイーンプログラムを MINERVA でコンパイルしたものより約 1.5 倍速いという結果を得た. これらの結果から, 本稿で提案したトランスレート方式は「リソースプログラミング」の特徴をうまく使ったプログラムに対して高速な実行が可能であると言える.

今後の課題としては以下のものを考えている.

- 本稿で述べた方式では, 生成されるクラスファイルの数が非常に大きくなるという問題がある. 述語及びリソースをメソッドにコンパイルし, Java のリフレクション機能を使って実行する方式についても検討する必要がある.
- 全称記号を含んだゴール $\forall x.G$ の実装
- HTML パーザの作成
- Java の制約解消系 (現在開発中) を用いて, Prolog Café で制約プログラミングを可能にする.
- ネットワーク上でオブジェクトの交換を可能にする JavaSpaces との融合
- コンパイルスピードの向上
- 浮動小数点演算の追加

謝辞 最後に, 査読者の方々, 並びにプログラミング研究会において貴重な御意見を下さった皆様に, 感謝いたします.

参考文献

1) Girard, J.-Y.: Linear Logic, *Theoretical Computer Science*, Vol. 50, pp. 1–102 (1987).
2) Hodas, J. S. and Miller, D.: Logic Program-

☆ http://bach.seg.kobe-u.ac.jp/llp/

☆☆ http://www.binnetcorp.com/Jinni/

☆☆☆ http://luke.wvhtf.org/~winikoff/wp/

ming in a Fragment of Intuitionistic Linear Logic, *Information and Computation*, Vol. 110, No. 2, pp. 327–365 (1994). Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

3) Harland, J. and Pym, D.: The Uniform Proof-Theoretic Foundation of Linear Logic Programming, *Proceedings of the International Logic Programming Symposium* (Saraswat, V. and Ueda, K.(eds.)), San Diego, California, pp. 304–318 (1991).

4) Andreoli, J.-M. and Pareschi, R.: Linear Objects: Logical Processes with Built-In Inheritance, *New Generation Computing*, Vol. 9, pp. 445–473 (1991).

5) Andreoli, J.-M.: Logic Programming with Focusing Proofs in Linear Logic, *Journal of Logic and Computation*, Vol. 2, No. 3, pp. 297–347 (1992).

6) Miller, D.: A Multiple-Conclusion Specification Logic, *Theoretical Computer Science*, Vol. 165, No. 1, pp. 201–232 (1996).

7) Kobayashi, N. and Yonezawa, A.: Typed Higher-Order concurrent linear logic programming, Technical Report 94-12, University of Tokyo (1994).

8) Hodas, J. S., Watkins, K., Tamura, N. and Kang, K.-S.: Efficient Implementation of a Linear Logic Programming Language, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming* (1998).

9) Tamura, N. and Kaneda, Y.: A Compiler System of a Linear Logic Programming Language, *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing*, pp. 180–183 (1997).

10) Warren, D. H. D.: *An abstract Prolog instruction set*, Technical Note 309, SRI International (1983).

11) Tamura, N. and Kaneda, Y.: Extension of WAM for a linear logic programming language, *Second Fuji International Workshop on Functional and Logic Programming* (Ida, T., Ohori, A. and Takeichi, M.(eds.)), World Scientific, pp. 33–50 (1996).

12) Banbara, M. and Tamura, N.: Compiling Resources in a Linear Logic Programming Language, *Proceedings of Post-JICSLP'98 Workshop on Parallelism and Implementation Technology for Logic Programming Languages* (1998).

13) Codognet, P. and Diaz, D.: WAMCC: Compiling Prolog to C, *Proceedings of International Conference on Logic Programming* (Sterling, L.(ed.)), The MIT Press, pp. 317–331 (1995).

14) Banbara, M. and Tamura, N.: Java Implementation of a Linear Logic Programming Language, *Proceedings of the 10th Exhibition and Symposium on Industrial Applications of Prolog*, pp. 56–63 (1997).

15) Tarau, P. and Boyer, M.: Elementary Logic Programs, *Proceedings of Programming Language Implementation and Logic Programming* (Deransart, P. and Maluszyński, J.(eds.)), Lecture Notes in Computer Science, No. 456, Springer, pp. 159–173 (1990).

16) Nadathur, G., Jayaraman, B. and Kwon, K.: Scoping Constructs in Logic Programming: Implementation Problems and their Solution, *Journal of Logic Programming*, Vol. 25, No. 2, pp. 119–161 (1995).

付　　録

A.1 比較に用いたプログラム

Prolog プログラム: *N*-Queen

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

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

not_attack(Xs,X) :-
        not_attack(Xs,X,1).

not_attack([],_,_) :- !.
not_attack([Y|Ys],X,N) :-
        X =\= Y+N, X =\= Y-N,
        N1 is N+1,
        not_attack(Ys,X,N1).

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

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

LLP プログラム: *N*-Queen

```
:- resource n/1, result/1, c/1, u/1, d/1.

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

A.2 assert と retract のソースコード

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

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

(平成 11 年 5 月 28 日受付)
(平成 11 年 10 月 13 日採録)

番原 睦則

1971 年生．1994 年神戸大学理学部数学科卒業．1996 年同大学大学院自然科学研究科博士課程前期数学専攻修了．同年国立奈良工業高等専門学校助手．1998 年より同校講師．線形論理（linear logic），論理プログラミングなどの研究に従事．日本ソフトウェア科学会会員．

姜 京順

1970 年生．1993 年済州大学（韓国）理学部数学科卒業．同年より 1 年間神戸大学大学院工学研究科研究生．1996 年同大学大学院自然科学研究科博士課程前期情報知能専攻修了．1996 年より同大学院自然科学研究科博士課程後期知能科学専攻．線形論理（linear logic），論理プログラミングなどの研究に従事．日本ソフトウェア科学会会員．

田村 直之（正会員）

1957 年生．1980 年神戸大学理学部物理学科卒業．1982 年同大学大学院工学研究科修士課程システム工学専攻修了．1985 年同大学院自然科学研究科博士課程システム科学専攻修了．学術博士．1985 年日本 IBM 東京基礎研究所入社．1988 年より神戸大学工学部勤務．線形論理（linear logic），論理プログラミングなどの研究に従事．