線形論理型言語のコンパイラ処理系のための抽象機械について

番原 睦則　姜 京順　田村 直之

本論文では，線形論理に基づいた論理型言語LLPの概要と，そのコンパイラ処理系のための抽象機械であるLLPAMについて述べている．LLPは，Prologの自然な拡張になっており，論理式をリソースとして取り扱うことができる点を特徴とする．抽象機械LLPAMは，LLPの効率的な計算モデルであるレベル付きIOモデルのリソース管理方式に基づいて設計されており，抽象機械WAMの拡張になっている．LLPAMでは，リソース自体もコンパイルされており，またリソースの検索はハッシュ表を通じて行われるため，線形論理型言語の特徴であるリソース論理式の追加や消費といった操作を非常に効率よく実現することができる．

1 はじめに

線形論理は1987年にJ.-Y. Girardが発表した新しい論理体系であり,「リソースを意識した」論理(resource-conscious logic)と呼ばれている[7][8]．古典論理や直観主義論理では仮定は何度でも利用できるが，線形論理では基本的には仮定は一度しか利用できない．すなわち，仮定は使用することによって消費され，消費された仮定は二度と使えない．

線形論理の計算機科学への応用のうち，線形論理に基づいた論理型言語の研究は，特に活発な分野の一つであり，LO [2], ACL [15], Lolli [13], Lygon [9], Forum [17], および著者らのLLP [3][4][14][20][21][24][26][27]およびTLLP [25]等の研究がある．

これらのうち，特にLolli[†1]およびLLP[†2]は以下のような特徴を持っており，線形論理型言語の一つの方向性を示している．

- Prologの自然な拡張になっており，さらに線形論理の資源性を生かしたプログラミングが可能である．たとえば，実行過程でPrologの節に相当する論理式を追加したり利用(消費)するようなプログラムを記述できる．
- Millerらのユニフォーム証明 [16]の考え方を，直観主義線形論理に適用した言語になっており，理論的なベースがしっかりしている．
- LO, Lygon, Forumなどのように，基礎となる論理を古典線形論理まで広げておらず，効率的な実装が可能である．

線形論理のリソース概念は，LolliおよびLLP以外の線形論理型言語においてもプログラミング上で重要な位置を占めている．したがって，リソースを効率良く管理するための計算モデルの設計が重要な研究テーマとなっており，いくつかの論文が発表されている[6][9][13][14][20]．特に，著者らの提案しているレベル付きIOモデル

An Abstract Machine for a Compiler System of a Linear Logic Programming Language.
Mutsunori Banbara, 奈良工業高等専門学校, Nara National College of Technology.
Kyoung-Sun Kang, 釜山外国語大学校(韓国)コンピュータ電子工学部コンピュータ工学科, Department of Computer Engineering, Division of Electronic and Computer Engineering, Pusan University of Foreign Studies.
Naoyuki Tamura, 神戸大学工学部情報知能工学科, Department of Computer and Systems Engineering, Kobe University.
コンピュータソフトウェア, Vol.18, No.1(2001), pp.39–60.
[論文]2000年1月20日受付.

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

[14] [20] [27] は，実行中はただ一つのリソース表を保持すれば良いように工夫されている点を特徴とし，コンパイラ処理系に適した計算モデルとなっている．

そこで，本論文では，レベル付きIOモデルのリソース管理方式に基づいて，LLP言語コンパイラのための抽象機械LLPAMを設計し，LLPプログラムのLLPAM命令列へのコンパイル方法について考察する．

2 線形論理型言語LLP

線形論理型言語LLPは，Prologのスーパーセットとなっており，さらに線形論理の「リソースを意識した論理」という特徴を生かしたプログラミングが可能である．すなわち，LLPではPrologの節に相当する論理式(リソース節と呼ぶ)を実行時に追加，呼出し，削除(消費)することが可能である．

たとえば，ゴール$S \multimap G$の実行は「リソース節Sを追加した後，ゴールGを実行する．ただしGを実行する際，Sをちょうど一度だけ用いなければならない」となる．これは，線形論理において「SならばG」は$S \multimap G$で表され，その意味は「仮定Sをちょうど一度だけ用いて，Gを導き出すことができる」となることに対応している．このように線形論理では，使用を制限された仮定を資源(リソース)とみなすことができる．LLPでは，この他にも線形論理のさまざまな論理演算子を用いたプログラミングが可能である．

本章では，本論文で対象としている線形論理型言語LLPの構文，およびプログラミング上の特徴とプログラム例について説明する．

2.1 LLPの構文

LLPは線形論理の以下のようなフラグメントを用いる．

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

ここで，Rはプログラム定義やリソースとして利用する論理式(**リソース論理式**と呼ぶ)，Gはゴールとして利用する論理式(**ゴール論理式**と呼ぶ)である．またAは原子論理式，論理結合子$\Rightarrow$は直観主義的含意[†3]を表す(すなわち$P \Rightarrow Q \equiv\ !P \multimap Q$である)．論文[14]と比較した場合，リソース論理式として$G \multimap R$，$G \Rightarrow R$，$\forall x.R$が追加されている．

表1 プログラム中の記法と論理式の対応

プログラム中の記法	線形論理の論理式
`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`)`	$\forall x.B$

リソースは，ゴール論理式$R \multimap G$または$R \Rightarrow G$の実行により追加されるが，$R \multimap G$の場合はリソースRはちょうど一度だけしか利用できず，$R \Rightarrow G$の場合はリソースRは0回以上何度でも利用できる．したがって，$R \multimap G$によって追加されるリソースを**有界リソース**，$R \Rightarrow G$によって追加されるリソースを**無限リソース**と呼ぶことにする．

また，上記の定義に対応し，線形論理型言語LLPのリソース論理式，ゴール論理式の記法を，Prologにならって以下のように定める(Aは原子論理式)[†4]．

$$\begin{aligned} R &::= A \mid R_1\texttt{\&}R_2 \mid G\texttt{-<>}R \mid G\texttt{=>}R \mid \texttt{forall(}X\texttt{,}R\texttt{)} \\ G &::= \texttt{true} \mid \texttt{erase} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid G_1\texttt{;}G_2 \mid \\ &\quad\ \texttt{!}G \mid R\texttt{-<>}G \mid R\texttt{=>}G \end{aligned}$$

さらに，`-<>`の逆向きとして`:-`を，`=>`の逆向きとして`<=`を使用することにする．なお，演算子の優先順位は弱いものから順に，“`:-`”，“`;`”，“`&`”，“`,`”，“`-<>`”，“`=>`”，“`!`”とし，“`:-`”以外の2項演算子は右結合的とする．プログラム中の記法と論理式の対応は，表1のようになる．

LLPプログラムは，次のように構文定義されるプログラム節Cの列である(Aは原子論理式)．

$$C \quad ::= \quad A\texttt{.} \mid A\texttt{:-}G\texttt{.}$$

†3 仮定が何回でも使えるという意味で直観主義的含意と呼んでおり，論理式$P \Rightarrow Q$自体は1回しか使えない．

†4 Lolliでは，本論文での記法とは異なり，ML風の記法を用いている．

プログラム節Cは，Prologと同様に，すべての自由変数を全称限量子で束縛した閉じた論理式を表すものとする．したがって，プログラム節Aは$\forall\vec{x}.A$を表し，A:-Gは$\forall\vec{x}.(G \multimap A)$を表している．

ゴール論理式として，AおよびG_1,G_2だけをもちいたLLPプログラムは，構文的にも操作意味論的にも純Prologと全く同一になる．さらに，リソース論理式を用いれば，線形論理のリソース消費性の特徴を生かしたプログラムを記述できる．

LLPでは，コンパイラ処理系の実現を主眼として，Lolliよりも少し狭いフラグメントを採用している．Lolliと比較して，リソース論理式中の$\top$およびゴール論理式中の$\forall$と$\exists$が除かれている．しかし，線形論理型言語のプログラミングで重要な演算子はすべて含まれており，一方，削除した演算子はプログラム中でほとんど使用することがないものである．

2.2 LLPプログラムの前処理

実は，前節で述べたLLP言語のリソース論理式は，常に以下のような形に変換できる(Aは原子論理式，$m \geq 1$).

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

上記の変換には，線形論理での以下の関係を利用し，左辺から右辺への置き換えを行えば良い(ただしzは，Gに自由な出現を持たない変数とする).

$$\begin{aligned} G \Rightarrow R &\equiv (!\,G) \multimap R \\ G \multimap (R_1 \,\&\, R_2) &\equiv (G \multimap R_1) \,\&\, (G \multimap R_2) \\ G_1 \multimap (G_2 \multimap R) &\equiv (G_1 \otimes G_2) \multimap R \\ G \multimap \forall x.R &\equiv \forall z.(G \multimap R[z/x]) \\ \forall x.(R_1 \,\&\, R_2) &\equiv (\forall x.R_1) \,\&\, (\forall x.R_2) \\ (R_1 \,\&\, R_2) \,\&\, R_3 &\equiv R_1 \,\&\, (R_2 \,\&\, R_3) \end{aligned}$$

このようなリソース論理式の変換は，言語処理系が自動的に行うものとすれば，言語の記述範囲はもとのままでよい．

さらに，この変換は処理系の実現をより容易にするものであり，LLPプログラムの動作を理解する上でも有用である．すなわち，ゴール$R \multimap G$あるいは$R \Rightarrow G$によって追加されるリソースは，常に$S_1 \,\&\, \cdots \,\&\, S_m$の形をしている．そしてその意味は「$S_1$から$S_m$のリソースのうち，どれか一つだけが利用可能である」となることから，$S_1 \,\&\, \cdots \,\&\, S_m$を**選択可能リソース**と呼ぶことにする．また各Sは，$\forall\vec{x}.A$または$\forall\vec{x}.(G \multimap A)$の形をしており，Prologでのプログラム節に相当する．そこで，Sを**リソース節**と呼ぶことにし，Aをそのヘッド部，Gをボディ部と呼ぶ．

また，上記の変換より前に，次の変換も行うものとする．

$$(R_1 \otimes R_2) \multimap G \quad \equiv \quad R_1 \multimap (R_2 \multimap G)$$

これは，複数のリソースの追加を簡便に記述するためのものである．

以下では，処理系が前処理として上のような変換をまず行うものとする．

2.3 LLPでのプログラミング

LLPでは，Prologの記述に加えて，実行時に動的にリソース論理式を追加したり削除(消費)したりするプログラムを記述できる．

逆に言うと，Prologに比べて多くの論理演算子が導入されており，一見するとわかりにくい印象を与える．しかし実際には，Prologにリソースを取り扱うための演算子が単純に追加されただけであり，Prologを理解しているプログラマにはそれほど理解が困難なものではない．

そこで本節では，Prologからの拡張部分を中心に，具体的な実行例を用いてLLPでのプログラミングについて説明する．

2.3.1 リソースの追加

LLPのゴールR-<>Gは，リソースRを追加した後，Gを実行する．たとえば以下の質問では，リソース`r(1)`が追加された後，ゴール`r(X)`が実行される．

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

これは，`X=1`とすることによってリソース`r(1)`を消費し，成功する．

ゴールR-<>Gの実行では，リソースRは必ず消費しなければならない．たとえば以下の質問は，リソース`r(1)`が消費されていないので失敗する．

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

リソース論理式Rが，G-<>A (あるいはA:-G)の場合は，規則タイプのリソースを表す．ゴールGはリソース消費の際に実行される．たとえば以下の質問では，`1`

が表示される．

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

リソース論理式R_1,R_2は，複数のリソースを表す．たとえば以下の質問では，リソース`r(1)`と`r(2)`が追加され，`X=1`, `Y=2`あるいは`X=2`, `Y=1`とすることによってリソースを消費し，成功する．

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

なお，この質問は以下と同一である．

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

ゴールR=>Gは，0回以上何回でも利用可能な無限リソースを追加する．たとえば以下の質問は，`X=1`あるいは`X=2`とすることによって成功する．

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

リソース論理式R_1&R_2は，選択可能なリソースを表す．たとえば`r(1) & r(2)`がリソースとして追加された場合，`r(1)`または`r(2)`のどちらか一方だけが利用可能である．以下の質問は，`X=1`あるいは`X=2`とすることによって成功する．

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

全称限量子で束縛されたリソースが，=>によって追加された場合，Prologにおける節の`assert`と同様に働く．たとえば以下の質問は，節`p(X) :- q(X)`を`assert`した後，ゴール`r`を実行するのに相当する．

```
?- forall(X, (p(X) :- q(X))) => r.
```

ただし，節が利用できるのは`r`中だけであり，さらにバックトラックにより消去される点が`assert`とは異なっている．

また，リソース中に自由変数を含む場合もある．たとえば以下の質問中で，リソース`r(X)`は何度でも利用できるが，`X`が自由変数であるという点で，事実`r(X)`を`assert`したのとは異なっている．

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

この場合，ゴール`r(1)`の実行で`X=1`となり，ゴール`r(Y)`の実行で変数`Y`も1に束縛される．

2.3.2 リソースの消費

Prologでは，原子論理式のゴールAの実行は，プログラム節の呼び出しのみを意味するが，LLPでは，プログラム節の呼び出しあるいはリソース消費を意味する．

すなわち，ある述語名がプログラム節の定義にもリソースにも利用されている場合，両方の可能性が試される．たとえば以下の質問では，`X=1`と`X=2`の両方が表示される．

```
r(2).          % プログラム中の定義
?- r(1) => r(X).
```

ゴールG_1,G_2はPrologのG_1,G_2と同様に実行される．G_1中で消費されたリソースはG_2中では消費できない．

ゴールG_1&G_2も，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一でなければならない．たとえば以下の質問は，`X=Y=1`, `Z=2`あるいは`X=Y=2`, `Z=1`とすることによって成功する．これは，`r(X)`と`r(Y)`の消費するリソースが同一でなければならないからである．

```
?- (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，実行中，無限リソースだけが利用可能である．たとえば以下の質問は`X=1`, `Y=2`とすることによって成功する．

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

ゴール`erase`は，いくつかのリソースが消費されないまま残ってもよいことを表す．すなわち`erase`は，いくつかのリソースを暗黙的に消費する．たとえば以下の質問は，`erase`が残り二つのリソースを暗黙的に消費することで成功する．

```
?- (r(1), r(2), r(3)) -<> (r(X), erase).
```

2.3.3 LLPのプログラム例

以下では，LLPのプログラム例をいくつか示す．

例 2.1 最初の例は，逆順リストを求めるプログラムである．このプログラムでは，リソース`result(Zs)`を再帰呼出しの最深部から結果を返すための「スロット」として用いている．

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

たとえば，`reverse([1,2,3], X)`を実行した場合，リソース`result(X)`が追加された後，`rev([1,2,3], [])`が実行される．そして再帰呼出しの最深部である`rev([],[3,2,1])`が実行されると，リソース`result(X)`が消費され，`X`は`[3,2,1]`と単一化される．

一方，Prologでは以下のように，結果を返すための引数(`rev`の第3引数)を用意する必要がある．

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

例 2.2 次のプログラムは，有向グラフでの経路探索の例である．各弧は，規則タイプのリソースとして表されている．一度通った弧は，消費され2度と通ることはできない．また，弧の推移的閉包としての経路の関係がエレガントに表現されている．

```
path :-
    (a -<> b) -<>      % 弧 a -> b
    (b -<> c) -<>      % 弧 b -> c
    (c -<> a) -<>      % 弧 c -> a
    (c -<> d) -<>      % 弧 c -> d
    (d -<> b) -<>      % 弧 d -> b
    a -<> (d, erase).  % aからdへの経路を探索
```

述語pathを実行すると，6個の有界リソースが追加された後，ゴール(d, erase)が実行される．dにより`c -<> d`が消費され，cが実行される．同様にして，`b -<> c`, `a -<> b`, aの順に消費される．最後にeraseにより残ったリソースが消費され成功する．

```
path :-
    ((a -<> b) & (a -<> d)) -<>
    (b -<> c) -<>
    (c -<> a) -<>
    (c -<> d) -<>
    (d -<> b) -<>
    a -<> (d, erase).
```

また，選択可能リソースを用いることにより，どちらか一方だけ選択可能な弧を表すことができる．上記のプログラムにおいて，`(a -<> b) & (a -<> d)`がリソースとして追加された場合，`a -<> b`または`a -<> d`のどちらか一方の弧だけが利用可能となる．

選択可能リソースは，どちらか一方を選択すると，他方は利用できないことから，Prologでの節の選択とは異なる．この他，選択可能リソースを用いた例題としては，論文[3]にタイルのはめ込み問題のプログラムが記載されている．

なお，上記のプログラムは次のように論理和(⊕，すなわち;)を用いても同様の探索を実行できるが，弧`a -<> b`と弧`a-<> d`のそれぞれに対して，別々に探索を行うことになり非効率的である．

```
path :-
    (a -<> b) -<>
    (b -<> c) -<>
    (c -<> a) -<>
    (c -<> d) -<>
    (d -<> b) -<>
    a -<> (d, erase)
   ;
    (a -<> d) -<>
    (b -<> c) -<>
    (c -<> a) -<>
    (c -<> d) -<>
    (d -<> b) -<>
    a -<> (d, erase).
```

例 2.3 次の例は，リストから与えられた条件を満たす要素をフィルタリングするプログラムである．

たとえば，`choose([1,2,3], 2, Z)`を実行した場合，リソース節`forall(X, (test(X) :- X>=2))`が無限リソースとして追加された後，`filter([1,2,3], Z)`が実行される．追加されたリソース節は，各要素に対する条件チェックのために使われている．

この無限リソースの追加は，節`test(X) :- X>=2`のassertと同様に働く．ただし，リソース節が利用できるのは，ゴール`filter([1,2,3], Z)`中だけであり，さらにバックトラックのより消去される点がassertとは異なっている．

```
% Zs は，Xs 中で Y 以上の要素からなるリスト
choose(Xs, Y, Zs) :-
    forall(X, (test(X) :- X>=Y)) =>
    filter(Xs, Zs).
filter([], []).
filter([X|Xs], [X|Zs]) :-
    test(X), !, filter(Xs, Zs).
filter([_|Xs], Zs) :- filter(Xs, Zs).
```

例 2.4 最後の例として，N-クイーンのプログラムを図1に示す．

このプログラムで，`queen(8,Q)`を実行した場合，リソース`result(Q)`が追加され，`place(8,8)`が実行される．`result(Q)`中の変数Qは，結果を表す変数であり，solveの中で結果の配置と単一化される．

`place(8,8)`は，リソース`c(1)`, ..., `c(8)`, `u(2)`, ..., `u(16)`, `d(-7)`, ..., `d(7)`を追加してから`solve(8,[])`を実行する．これらのリソースc, u, dは，各列，各右上がり

```
queens(N,Q) :-
    result(Q)-<> place(N,N).
place(1,N) :-
    c(1)-<>u(2)-<>d(0)-<> solve(N,[]).
place(I,N) :-
    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,N).
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]).
```

図1　Lolliで記述したNクイーンのプログラム

```
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).
```

図2　Prologで記述したNクイーンのプログラム

のライン，各右下がりのラインにそれぞれ対応している．`solve`中で`I`行目にクイーンを置くときに，`c(J)`, `u(I+J)`, `d(I-J)`を消費することによって，各列，各右上がりのライン，各右下がりのラインに高々一つしかクイーンを置けないという制約条件を表している(図3参照)．

一方，図2に示す，Prologで記述したN-クイーンのプログラム(教科書[5]中と同様のもの)の場合，LLPプログラムでリソースを利用して行っていた制約条件のチェックをリストを用いて行っている．したがって，各制約条件のチェックには，リストの長さに比例する(したがって，Nに比例する)計算量がかかる．

この外，論文[11]，[13]，[22]には，LolliおよびLLPのプログラム例として，自然言語のパーザ，直観主義命題論理の証明系，実験計画でのBIBD配置の探索プログラム，ハミルトン経路の探索，覆面算，ペントミノなどのパズルプログラムが記載されている．

3　LLPコンパイラの設計

LLPの操作的意味論は，HodasとMillerによるユニフォーム証明に基づいた体系，あるいは，ユニフォーム証明探索におけるリソース分割を遅延的に行うことで，効率を改善したIOモデルの体系[10][12][13]で与えられる．

MLやProlog上で，IOモデルに基づいた処理系を開発することは容易である．実際，Lolliのインタプリタ

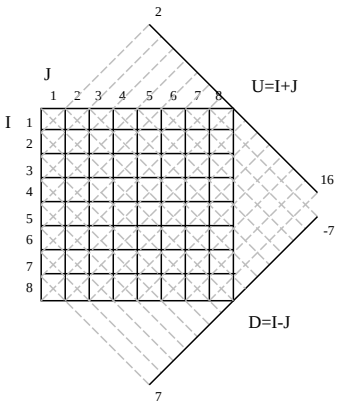

図3　8クイーンにおけるリソース

処理系は，SML上でIOモデルを実現したものになっている．そこで，LLPプログラムの動作の理解の助けのため，図4にIOモデルに基づたLLPインタプリタ処理系(Prologで記述)を示す．ただし，本インタプリタはリソース論理式中の`forall`演算子，および逆向きの含意記号(`:-`と`<=`)には未対応である．

IOモデルでは，ゴールをリソースを消費する消費者と考え，ゴールGの実行中に利用可能なリソースのリスト(**入力コンテキスト**と呼ばれる)と，Gの実行後に残ったリソースのリスト(**出力コンテキスト**と呼ばれる)を保持しながら実行が進む．図4のインタプリタの場合は，述語`prove`(G，I，O)で，Gが証明すべきゴール，Iが

```
:- 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]) )).
```

図4 IOモデルに基づいたLLPインタプリタ (Prologで記述)

入力コンテキスト，Oが出力コンテキストを表している．たとえば，ゴール$G_1 \otimes G_2$の実行は，インタプリタ中の`prove`$((G_1 \otimes G_2))$，I，O)をヘッド部にもつ節:

```
prove((G1,G2), I, O) :- !,
    prove(G1, I, M), prove(G2, M, O).
```

を見るとわかるように，まず入力コンテキストIのうちのいくつかを使ってG_1を実行し，その後，残ったリソースMを使ってG_2を実行する．残ったリソースでのG_2の実行が失敗すれば，G_1の実行にバックトラックし，別の消費方法を探す．

このインタプリタを，部分評価系などと組み合わせれば，LLPプログラムをそれと同値なPrologプログラムにトランスレートし，さらにPrologコンパイラでコンパイルすることによって，LLPコンパイラ処理系を実装することは容易である．たとえば，図2のPrologのN-クイーンプログラムは，図1のLLPプログラムを部分評価系でトランスレートしたものと同様の動作を行っている．すなわち，述語`sol(I,Js,Us,Ds,Q)`の`Js`, `Us`, `Ds`は，述語名`c`, `u`, `d`の利用可能なリソースをそれぞれリストで表現したものに対応している．

HodasらのIOモデルは，リソース分割を遅延的に行うという点で，優れたリソース管理モデルといえる．しかし，このように入出力コンテキストをリストで表現した場合，以下のような問題点が生じる．

1. **入出力コンテキストの探索の問題:**
   インタプリタ中の`pick`(I,O,A)述語では，入力コンテキストI中からヘッド部がAに単一化可能なリソース節Sを探索している．入出力コンテキストを

リスト構造で表現している場合，リストの長さに比例する時間がかかってしまう．

2. **入出力コンテキストの再構築の問題:**
さらに`pick(`I`,`O`,`A`)`では，新しい出力コンテキストOを再構築している．すなわち，有界リソースが消費された場合，入力コンテキストI中で，その有界リソースに対応する箇所を1に置き換えた出力コンテキストOを作成している．これも，リスト構造を用いている場合，入出力コンテキストのサイズに比例した時間と領域を必要とする．

これらは，実行時に動的にリソース論理式を追加したり削除(消費)したりできるというLLPの特徴を考えれば，到底看過できない問題点である．

第1の「入出力コンテキストの探索」という問題点は，入出力コンテキストをリスト構造ではなく，述語名と引数などをキーとしたハッシュ表で実現すれば解決できると考えられる．しかし，ハッシュのキーに未束縛の論理変数が現れる可能性があるなど，実装には工夫が必要である．

第2の「入出力コンテキストの再構築」という問題点は，入出力コンテキストを破壊的に書き換えれば解決できるように見える(もちろん，バックトラック時には元に戻すようにする)．しかし，この場合，ゴール$G_1 \,\&\, G_2$の実行のために，複数の入出力コンテキストを管理する必要が生じてしまう．すなわち，図4のインタプリタ中の`prove((`$G_1 \,\&\, G_2$`), `I`, `O`)`をヘッド部にもつ節:

```
prove((G1&G2), I, O) :- !,
    prove(G1, I, O), prove(G2, I, O).
```

を見るとわかるように，G_1の実行とG_2の実行とでは，同一の入力コンテキストIを必要とする．よって，入出力コンテキストを破壊的に書き換えた場合，G_1の実行によってIが書き換えられてしまうため，G_1の実行の前に，Iのコピーを保存しておく必要がある．同様に，G_1の実行によって書き換えられた結果であるOも，G_2の結果と一致するかどうかを調べる必要があるから，やはりコピーを保存しておく必要がある．

しかし，見方を変えれば，ゴールG_2はゴールG_1で消費されたリソースだけを用い，かつそれらをすべて消費する(あるいは，ゴールG_1はゴールG_2で消費されたリソースだけを用い，かつそれらをすべて消費する)，と考えることもできる．したがって，各リソースに消費済みかどうかのフラグを付与し，G_1で消費されたリソースを判別できるようにする方法が考えられる．しかし，単なるフラグだけでは，ゴール中に&がネストしている場合にうまく働かない．

著者らの設計したレベル付きIOモデルでは，各リソースにフラグではなく整数値を付与することにより，この問題点を解決している．したがって，レベル付きIOモデルを採用すれば，単一の入出力コンテキストを保持するだけでよい．

そこで以下では，まずレベル付きIOモデルの体系IOLについて述べた後，IOLに基づいた具体的なリソース管理方法の実装方針について述べる．また，リソース論理式そのもののコンパイル方法についても考察する．

レベル付きIOモデルの採用は，開発する処理系がインタプリタかコンパイラかの問題とは独立である．インタプリタの場合でも，キーに未束縛の論理変数が利用可能なハッシュ表があれば，以下に述べる実装方法に基づいたインタプリタ処理系を実現できる．しかし，この場合，リソース論理式はコンパイルされず，動的に解釈および実行されることになる．

3.1 レベル付きIOモデルの採用

レベル付きIOモデル[27]は，「実行中に保持すべき入出力コンテキストをただ一つにできる」という点を特徴とする．

レベル付きIOモデルの体系IOLでは，入出力コンテキスト中の各リソース論理式に，現在使用可能かどうかを表すための整数値を割り当てている．すなわち，レベル付きリソース論理式とは，リソース論理式Rとレベルを表す整数ℓの対であり，$\langle R, \ell \rangle$で表わされる．特に，無限リソースに対しては$\ell = 0$とし，有界リソースに対しては$\ell \neq 0$とする．

また，レベル付きIOモデルでは，以下の形のシーケントを用いる．

$$\vdash_{L,U} I\,\{G\}\,O$$

ここで，Lは正整数で，消費可能なリソース論理式のレベルを表しており，**消費可能レベル**と呼ばれる．初期値は1である．Uは負整数で，消費後に割り当てられるレベルを表しており，**消費後レベル**と呼ばれる．初期値は

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

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

$$\frac{\vdash_{L,U-1} I\,\{G_1\}\,M \quad change_{U-1,L+1}(M,N) \quad \vdash_{L+1,U} N\,\{G_2\}\,O \quad thinable_{L+1}(O)}{\vdash_{L,U} I\,\{G_1 \,\&\, G_2\}\,O}\ (\&)$$

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

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

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

図5　レベル付きIOモデルの体系IOL

-1である．Gはゴール論理式である．I, Oはレベル付きリソース論理式のリストであり，入出力コンテキストを表す．

$\vdash_{L,U} I\,\{G\}\,O$の実行においては，I中でレベルがL(または0)であるリソースだけが消費可能と考え，消費された場合にはレベルはU (または0のまま)となるようにする．このようにすることで，$G_1 \,\&\, G_2$中のG_1の実行でどのリソースが消費されたかを判別できるようになる．すなわち，$\vdash_{L,U} I\,\{G_1 \,\&\, G_2\}\,O$は以下のようなステップで実行される．

1. G_1でどのリソースが消費されたかわかるように，消費後レベルを$U-1$としG_1を実行する．
2. 入出力コンテキスト中で，レベルが$U-1$となっているのはG_1で消費されたリソースであるから，それらのレベルを$L+1$に変更する．
3. 消費可能レベルを$L+1$，消費後レベルをUとしてG_2を実行する．
4. G_2は，G_1で消費したリソースすべてを消費しなければならない．すなわち，入出力コンテキスト中にレベル$L+1$のリソースが残っていてはならないので，それをチェックする．

このように考えた，レベル付きIOモデルの体系IOLを図5に示す．

たとえば，規則(&)においてI中のリソースのうちG_1で消費されたものを$r_1,\ldots,r_k$とすると，まず$\vdash_{L,U-1} I\,\{G_1\}\,M$の実行により，$M$中の$r_1,\ldots,r_k$の各レベルは$U-1$となる．次に$change_{U-1,L+1}(M,N)$の実行により，$N$中の$r_1,\ldots,r_k$のレベルは$L+1$となる．そのため，$\vdash_{L+1,U} N\,\{G_2\}\,O$において，$G_2$が消費できるリソースは$r_1,\ldots,r_k$だけであり，そのうち$G_2$によって消費されたリソースのレベルは$U$となる．最後に，$thinable_{L+1}(O)$により，$r_1,\ldots,r_k$のすべてが消費されたかどうかチェックする．したがって，規則(&)が成功した場合，G_1とG_2が消費したリソースは同一であり，それらのレベルはUになっている．

図5中の，*subcontext*, *pick*, *change*, *thinable*はそれぞれ以下のように定義される述語である．

$subcontext_{U,L}([s_1,\ldots,s_n],[r_1,\ldots,r_n]) \Longleftrightarrow$
各$i=1,2,\ldots,n$について，(1) $r_i=\langle R,L\rangle$のときは$s_i=r_i$または$s_i=\langle R,U\rangle$，(2) その他のときは$s_i=r_i$となっている．

$pick_{L,U}([r_1,\ldots,r_n],[s_1,\ldots,s_n],S) \Longleftrightarrow$
(1) ある$r_i=\langle S_1 \,\&\cdots\&\, S_m,L\rangle$について$S_k=S$であり，$s_i=\langle S_1 \,\&\cdots\&\, S_m,U\rangle$かつ$s_j=r_j$ $(j\neq i)$となっている，または，(2) ある$r_i=\langle S_1 \,\&\cdots\&\, S_m,0\rangle$について$S_k=S$であり，$s_j=r_j$ $(1\le j\le n)$となっている．

$change_{\ell,\ell'}([r_1,\ldots,r_n],[s_1,\ldots,s_n]) \Longleftrightarrow$

各$i = 1, 2, \ldots, n$について，(1) $r_i = \langle R, \ell \rangle$のときは$s_i = \langle R, \ell' \rangle$，(2) その他のときは$s_i = r_i$となっている．

$thinable_\ell([r_1, \ldots, r_n]) \Longleftrightarrow$
すべての$r_i = \langle R, \ell' \rangle$について，$\ell \neq \ell'$となっている．

3.2 入出力コンテキストの表現

レベル付きIOモデルを採用すれば，実行中に保持すべき入出力コンテキストをただ一つにできる．したがって，ハッシュ表などを用いて入出力コンテキストのアクセスを高速にするなどの実装方法を容易に取ることができる．

まず，入出力コンテキスト自体は1次元の表(リソース表と呼ぶ)で表現することにする．また，ヘッド部が単一化可能なリソースを高速に検索するためにハッシュ表を用いる．

3.2.1 リソース表

リソース表の各エントリには，以下のような情報が格納される．

```
record
  s            : 先頭のリソース節へのポインタ;
  n            : リソース節の個数(正整数);
  level        : リソースのレベル(整数);
  out_of_scope : 範囲外フラグ(ブール値);
  pred         : リソース節のヘッド部分の述語名;
  code         : リソース節のクロージャ
end
```

このエントリは，追加された各リソース節ごとに用意する．すなわち，選択可能リソース$S_1 \,\&\, \cdots \,\&\, S_m$が追加された場合，各リソース節$S_i$ごとに作成する．ただし，各$S_i$の`s`フィールドには先頭のリソース節$S_1$のエントリへのポインタをセットし，`n`フィールドには選択可能リソース$S_1 \,\&\, \cdots \,\&\, S_m$に含まれるリソース節の個数$m$をセットする．

`level`は，レベル付きIOモデルのレベル値である．$R \multimap G$で追加された有界リソースの場合は，現在の消費可能レベルLの値，$R \Rightarrow G$で追加された無限リソースの場合は0が最初にセットされる．リソース消費の際には，自分自身のレベルだけでなく，選択可能リソース$S_1 \,\&\, \cdots \,\&\, S_m$中の他の$S_i$のレベルも変更する必要があるが，これは，`s`と`n`のフィールドを参照して行う．

フラグ`out_of_scope`は，このリソース節が有効範囲に入っているかどうかを示すフラグである(有効範囲に入っていれば`false`)．すなわち，ゴール$R \multimap G$の実行が成功した場合，Rに対応するエントリを削除するのでなく，単にフラグを`true`にセットすることで，R中のリソースを利用不可にする．バックトラックによってGの実行が再開された時には，再びフラグを`false`にセットし，Rを利用可能にする．

`pred`は，リソース節のヘッド部の述語名(アリティ情報を含む)である．`code`は，ヘッド部の単一化とボディ部の実行のためのコードへのポインタである(実際には後述のクロージャ)．

3.2.2 ハッシュ表

ハッシュ表は，ヘッド部が単一化可能なリソース節を高速に検索するために用いる．

ヘッド部の述語名と引数をそのままハッシュのキーとすれば良いように考えられるが，追加時には引数が未束縛変数であり，それが検索時には具体化されている可能性がある．たとえば，`p(1)`と単一化可能なリソース節を検索する場合，追加時にはヘッド部が`p(X)`だが，その後`X`が`1`に具体化されたリソース節も探し出す必要がある．

そこで，本論文では，ヘッド部の述語名と第一引数のトップレベルをキーとして利用するが，追加時に第一引数が未束縛変数の場合はハッシュ表に登録せず，それらのリストを別に保持しておく方法を取る．

また逆に，リソースを検索するゴール側の第一引数が未束縛変数の場合もある．この場合は，同一述語名を持つすべてのリソース節が利用可能であるから，それらのリストも別に保持しておく．

具体的には，以下のように実装する．

ハッシュ表のキーは，述語名と第一引数のトップレベルとし，値は同一のハッシュ関数値を持つリソース節のリストとする．また，シンボル表の各エントリに以下のフィールドを用意しておく．

```
r1    : リソース・リスト1;
r2    : リソース・リスト2;
```

`r1`は，このシンボルをヘッド部の述語名として持つリソース節のリストであり，`r2`は，このシンボルをヘッド部の述語名として持つリソース節のうち追加時に第一引

数が未束縛変数だったもののリストである．

リソース追加時の処理は以下のようになる．追加すべきリソース節のヘッド部を$p(X_1,\ldots,X_n)$とする．X_1が未束縛変数の場合は，シンボルp/nのフィールド`r1`と`r2`にリソース節を追加する．X_1が未束縛変数でない場合は，ハッシュ表およびシンボルp/nの`r1`に追加する．

リソース検索時の処理は以下のようになる．今，$p(X_1,\ldots,X_n)$と単一化可能なヘッド部を持つリソース節を検索するとする．X_1が未束縛変数の場合は，シンボルp/nのフィールド`r1`に追加されているリソース節に対して処理を行う．X_1が未束縛変数でない場合は，同一のハッシュ関数値を持つリソースおよびシンボルp/nのフィールド`r2`に追加されているリソース節に対して処理を行う．

3.3 クロージャの導入

LLPのための抽象機械LLPAMは，Prologコンパイラで広く用いられているWAM [1][23]の自然な拡張になっている．ここでは，LLPAMの詳細を述べる前に，リソース節のコンパイルについて考察する．なお，WAM命令の説明には，教科書[1]の記法を用いる．

自由変数を含んでいないリソース節のコンパイルは容易である．たとえば，リソース節$\forall X.\forall Y.(q(X,Y) \multimap p(f(X),Y))$は，プログラム節$p(f(X),Y)$:-$q(X,Y)$と全く同様にコンパイルすれば良い．

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

ここで，WAM命令`get_structure` $f/1$, `A1`は，引数レジスタ`A1`が未束縛変数の場合，ファンクタ$f/1$を指す新しい`STR`セルをヒープ上に作成し，`mode`フラグを`write`にセットする．`A1`がファンクタ$f/1$を指す`STR`セルの場合は，その引数を取り出すために，WAMレジスタ`S`をファンクタセルに続くヒープ上のアドレスにセットし，`mode`フラグを`read`にセットする．また，WAM命令`unify_variable A1`は，`read`モードの場合，レジスタ`S`の指す内容を`A1`にセットし，`write`モードの場合は，新しく未束縛の変数をヒープ上作成し，その内容を`A1`にコピーする．

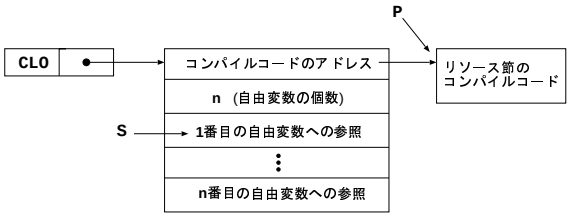

図6 クロージャ構造

しかし，自由変数を含んでいるリソース節の場合は，リソースの実行時にそれらの自由変数の値を知る必要がある．たとえば，リソース節$\forall Y.(q(X,Y) \multimap p(f(X),Y))$の場合，このリソース節が呼び出される時点での自由変数Xの値を知る必要がある．

この処理の実現のために，**クロージャ**を導入する．図6に示すように，クロージャはコンパイルされたコードと変数環境(自由変数の個数と自由変数への参照)からなる構造体であり，`CLO`タグによってタグ付けされる．

クロージャの呼び出し時には，WAMの`S`レジスタをクロージャ構造の3番目のセルを指すようにセットし，命令ポインタ`P`をコンパイルされたコードを指すようにセットする．また，`mode`フラグは`read`にセットする．したがって，クロージャのコードの最初の部分で，`unify_variable`命令を実行すれば，自由変数の値を取り出すことができる．

以下は，リソース節$\forall Y.(q(X,Y) \multimap p(f(X),Y))$のコード例である．最初の`unify_variable A3`の命令で，自由変数Xの値をレジスタ`A3`に取り出している．

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

ここで用いたクロージャのアイデアは，関数型プログラミング言語の実装で広く用いられている．

すでに含意を導入している言語には，MillerとNadathurの提案した直観主義論理に基づく論理型言語

λProlog [18][19]がある．λPrologのゴール$D \supset G$によって追加される節Dのコンパイルにも同様のアイデアが使われているが，$D \supset G$はLLPの$D \Rightarrow G$に対応するものであり，LLPの$D \multimap G$に対応するゴールはλPrologには存在しない．そのため，LLPとλPrologでは，クロージャの管理方法が異なる．

4 抽象機械LLPAM

本章では，抽象機械LLPAMの詳細について述べる．

LLPAMは，LLPの効率的な計算モデルであるレベル付きIOモデルのリソース管理方式に基づいて設計されており，Prologの抽象機械WAMの拡張になっている．

4.1 データ領域とレジスタ

WAMで用いられているCODE, HEAP, TRAIL, PDLに加えて，LLPAMではSYMBOL, RES, HASHと呼ばれるデータ領域を用いる．

SYMBOL領域の各エントリは，各記号に対して用意され，以下のような情報が格納される．

```
record
  pname : 印字名を表す文字列;
  arity : アリティを表す整数;
  code  : コンパイル・コード;
  r1    : リソース・リスト1;
  r2    : リソース・リスト2
end
```

codeフィールドには，この述語名でプログラムが定義されていれば，それをコンパイルしたLLPAMコードへのポインタが格納される．

前述のように，r1フィールドは，ヘッド部に同一述語名を持つリソースのリスト，r2フィールドは，ヘッド部に同一述語名を持ち，追加時に第一引数が未束縛だったリソースのリストである．

RES領域，HASH領域は，前述のリソース表とハッシュ表である．

レジスタについては，WAMで用いられてるP, CP, S, H, HB, E, B, B0, TRに加えて，以下のレジスタを新たに導入する．

- Rレジスタ: リソース・トップ
- Lレジスタ: 消費可能レベル (整数，初期値は1)
- Uレジスタ: 消費後レベル (整数，初期値は-1)
- R1レジスタ: 消費可能リソース・リスト1
- R2レジスタ: 消費可能リソース・リスト2

Rレジスタは，RES領域の未使用領域の先頭を指す．Lレジスタと Uレジスタは，レベル付きIOモデルの消費可能レベルと消費後レベルである．R1レジスタとR2レジスタには，述語呼び出し時に，前述したシンボル表とハッシュ表から取り出したリソース節のリスト(消費可能なリソース節の候補のリスト)がセットされる．

4.2 ゴール$G_1 \otimes G_2$のコード

レベル付きIOモデルの体系IOLでは，$G_1 \otimes G_2$は単にゴールG_1の実行後にゴールG_2の実行を行えばよい．

すなわちリソース表RESは，$\vdash_{L,U} I\,\{G_1\}\,M$の実行によって$I$から$M$に変更され，$\vdash_{L,U} M\,\{G_2\}\,O$の実行によって$M$から$O$に変更される．したがって，ゴール$G_1 \otimes G_2$のコードは以下のようにゴール$G_1$のコードと$G_2$のコードを並べたものになる．

ゴールG_1のコード
ゴールG_2のコード

4.3 ゴール$G_1 \,\&\, G_2$のコード

ゴール$G_1 \,\&\, G_2$のコードは，以下のようになる．

```
begin_with
ゴールG1のコード
mid_with
ゴールG2のコード
end_with
```

ここで，各命令の処理は以下の通りである

- begin_with
  レジスタUをデクリメントする．
- mid_with
  リソース表RES中で，レベルがUに一致しておりout_of_scopeでないリソース節のレベルをL+1に変更する(*change*の処理)．さらに，LとUをインクリメントする．またバックトラックに備えて，レベルを変更したリソースの表中の位置と元のレベル値をトレイルに積んでおく．

- `end_with`
  リソース表RES中で，レベルがLに一致しておりout_of_scopeでないリソース節が残っていればバックトラックする(*thinable*の処理)．そうでなければLをデクリメントする．

上の各命令は，レベル付きIOモデルの体系IOLの規則(&) の処理に対応している．すなわち，begin_with命令は，前提部の一番目の条件$\vdash_{L,U-1} I\{G_1\}M$で消費後レベルをUから$U-1$ に変更することに対応する．mid_with命令は，二番目の条件$change_{U-1,L+1}(M,N)$と，三番目の条件$\vdash_{L+1,U} N\{G_2\}O$で消費可能レベルをLから$L+1$に，消費後レベルを$U-1$からUに変更することに対応している．end_with命令は，四番目の条件$thinable_{L+1}(O)$に対応している．

4.4 ゴール$!G$のコード

ゴール$!G$のコードは，以下のようになる．

```
begin_bang
ゴールGのコード
end_bang
```

各命令の処理は以下の通りである

- `begin_bang`
  レジスタLをインクリメントする．
- `end_bang`
  レジスタLをデクリメントする．

4.5 ゴール$\top$のコード

ゴール$\top$に対しては，一命令のコードを生成する．

```
top
```

ゴール$\top$は，消費可能なリソースのいくつかを消費することを意味する．どのリソースを消費するかは非決定的である．

しかし，実際のプログラムではほとんどの場合,「消費可能なリソースのすべてを消費する」ことを意図している．したがって，本論文では命令topの処理を以下のように定める．

- `top`
  リソース表RES中で，レベルがLに一致しておりout_of_scopeでないすべてのリソース節のレベルをUに変更する．

ゴール$\top$の完全な処理方法については，Hodasらと著者らの共同論文[14]で論じられている．

4.6 ゴール$R \multimap G$のコード

ゴール$R \multimap G$は，リソースRを追加しゴールGを実行する．リソースRは常に$S_1 \,\&\, \cdots \,\&\, S_m$の形をしている．

ゴール$R \multimap G$の実行の概要は以下のようになる．

1. 現在のレジスタR (リソース・トップ)の値をパーマネントレジスタ$\mathtt{Y}_i$に保存しておく．
2. リソースR中のすべてのリソース節S_iをリソース表RESに追加する(Rはm増加する)．また，それぞれをハッシュ表とシンボル表にも登録する．
3. ゴールGを実行する．
4. リソースRが消費されているかどうかをチェックする．消費されていなければ失敗する．
5. リソースR中のすべてのリソース節のout_of_scopeをtrueにセットする．ゴール$R \multimap G$の実行において，ゴールGが成功すれば，リソースRは必要なくなる．しかしG中に選択点が残っておりバックトラックが起きた場合，Rを再度利用可能にする必要がある．したがって，リソースRのエントリをリソース表RESから完全に削除するのではなく，out_of_scopeフラグだけをセットする．

ゴール$R \multimap G$全体のコードは以下の通りである．(有界リソースS_i追加のコードについては後述)．

```
begin_imp Y_i
有界リソースS_1追加のコード
.....
有界リソースS_m追加のコード
mid_imp m
ゴールGのコード
end_imp Y_i
```

各命令の処理は以下の通りである．

- begin_imp Y_i
  レジスタRの値をパーマネントレジスタY_iに代入する．
- mid_imp m
  リソース表RES中のR $-$ mからR $-$ 1の位置の各リソース節について，フィールドsにR$-m$を，フィールドnにmを代入する．
- end_imp Y_i
  追加されたリソースが消費されないまま残っているならば，すなわち，リソース表RES中のY_iの位置のリソース節のレベルがレジスタLと同じ値であればバックトラックする．違う値であれば，Y_iの位置のリソース節のフィールドnの値をmとして，Y_iから$Y_i + m - 1$の位置の各リソース節(同じ選択可能リソースに含まれる各リソース節)のout_of_scopeフラグをtrueにする．また，バックトラック時にフラグをfalseに戻せるように，Y_iの値をトレイルに積んでおく．

4.7 ゴール$R \Rightarrow G$のコード

ゴール$R \Rightarrow G$のコードは，以下の通りである．(無限リソースS_i追加のコードについては後述)．

begin_exp_imp Y_i
無限リソースS_1追加のコード
.....
無限リソースS_m追加のコード
mid_exp_imp m
ゴールGのコード
end_exp_imp Y_i

各命令の処理は以下の通りである．

- begin_exp_imp Y_i
  begin_imp Y_iと同一．
- mid_exp_imp m
  mid_imp mと同一．
- end_exp_imp Y_i
  Y_iの位置のリソース節のフィールドnの値をmとして，Y_iから$Y_i + m - 1$の位置の各リソース節のout_of_scopeフラグをtrueにする．また，バックトラック時にフラグをfalseに戻せるように，Y_iの値をトレイルに積んでおく．

4.8 リソース節Sのコード

リソース節Sのヘッド部がAであり，ボディ部がGであるとする．また，Aの述語名をp/nとし，S中にX_1, ..., X_mが自由変数として現れているとする．

このとき，リソース節Sの消費時に実行されるコードは以下のようになる．

L: unify_variable A_{n+1}
unify_variable A_{n+2}
.....
unify_variable A_{n+m}
ヘッド部Aのコード
ボディ部Gのコード

通常のプログラム節の呼出しの場合と同様に，実引数がレジスタA_1からA_nにセットされた状態で，このコードが実行される．したがって，自由変数の値はunify_variable命令で空いているA_{n+1}以降のレジスタに取り込む．その後に続いているヘッド部Aとボディ部Gに対するコードは，プログラム節A:-Gのコンパイルの場合と同様に生成すれば良い．

リソース節Sの追加の実行概要は以下のようになる．

1. put命令でヘッド部Aの構造体を作成し，A_iレジスタにセットする．この構造体は，ハッシュ値を計算するために利用される．
2. クロージャを作成し，A_jレジスタにセットする．上記のラベルLがコードのアドレス，X_1, ..., X_mが自由変数である．
3. リソース節をリソース表に追加する．同時にハッシュ表とシンボル表にも登録する．

以上の処理のために，以下の命令を利用する．

- put_closure L, n, A_i
  クロージャ構造の最初の二つのセル(コードアドレスLと自由変数の個数n)の部分をHEAP上に作成し，A_iがそれを指すようにセットする．またmodeフラグをwriteにセットする．すなわち，以下の動作を

行う.

```
Ai := ⟨CLO, H⟩;
HEAP[H] := L;
H := H + 1;
HEAP[H] := n;
H := H + 1;
mode := write;
```

- add_res A_i, A_j

  ヘッド部がA_iでクロージャがA_jの有界リソース節をリソース表RESに追加する．すなわち以下の動作を行う．

```
RES[R].level := L;
RES[R].out_of_scope := false;
RES[R].pred := Aiのファンクタ名;
RES[R].code := Aj;
Aiのファンクタ名と第一引数をキーとして
Rの値をハッシュ表とシンボル表に登録する;
R := R + 1;
```

  なお，sとnのフィールドはmid_imp命令でセットされる．

- add_exp_res A_i, A_j

  ヘッド部がA_iでクロージャがA_jの無限リソース節をリソース表RESに追加する．すなわち，以下の動作を行う．

```
RES[R].level := 0;
RES[R].out_of_scope := false;
RES[R].pred := Aiのファンクタ名;
RES[R].code := Aj;
Aiのファンクタ名と第一引数をキーとして
Rの値をハッシュ表とシンボル表に登録する;
R := R + 1;
```

  なお，sとnのフィールドはmid_exp_imp命令でセットされる．

4.9 述語に対するコード

LLPでは，述語呼び出しは，通常のプログラム節の呼び出しに加えて，リソース節の呼び出しも意味する．すなわち，プログラム節およびリソース節について，すべての実行を試みる必要がある．

したがって，述語p/nの呼び出しの実行概要は以下のようになる．

1. ヘッド部がマッチする可能性のあるリソース節のリストをシンボル表とハッシュ表から得る．また，これらのリストを2つのレジスタR1とR2に格納する．
2. R1とR2中のリソース節で，述語名p/nを持つリソース節Sに対して，以下を試みる．
   (a) Sがout_of_scope，またはすでに消費されているならば，他のリソース節を調べる．
   (b) S (および同一の選択可能リソース中の他のリソース節) のレベルを更新する．
   (c) Sのクロージャを実行する．
3. すべての試みが失敗すれば，通常のプログラムp/nを呼び出す．

上記の処理のうち，ステップ1は命令call p/n (またはexecute p/n) 中で行う．ステップ2と3については，以下で述べる新しい命令を使用し，それらで記述した命令列を述語p/nのコードの先頭に挿入する．

- try_resource L

  WAMの命令try Lと同じであるが，レジスタR1とR2の値も選択点フレームに保存する．

- restore_resource

  現在の選択点フレームから，R1, R2を含め各レジスタの値を取り出す．

- retry_resource_else L

  現在の選択点フレーム中のR1, R2フィールドおよびBPフィールドに，現在のR1, R2の値およびLを代入する．

- trust_resource L

  選択点フレームを一つ前に戻し，ラベルLにジャンプする．すなわち， 現在の選択点フレーム中に保存されているBのフィールドの値をレジスタBにセットし，LのアドレスをレジスタPにセットする．

- pickup_resource p/n, A_i, L

  レジスタR1 (R1が空リストならばR2) で指されるリストから，ファンクタp/nを持つ消費可能なリソースを検索し，それをA_iにセットする (実際には，リソース表RESのインデックス値をA_iにセットする)．レジスタR1 (あるいはR2) の値は，見つかったリソースの次の要素を指すように更新しておく．消費可能なリソースが見つからなかった場合は，ラベルLにジャンプする．

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

L':  p/nのプログラム節のコード
```

図7 述語p/nのLLPAMコード

```
found := false;
while R1≠nil ∧ ¬found do begin
  r := car(R1);  R1 := cdr(R1);
  found := RES[r].pred = p/n
         ∧ ¬RES[r].out_of_scope
         ∧ RES[r].level = 0またはL;
end;
while R2≠nil ∧ ¬found do begin
  r := car(R2);  R2 := cdr(R2);
  found := RES[r].pred = p/n
         ∧ ¬RES[r].out_of_scope
         ∧ RES[r].level = 0またはL;
end;
if found then
  A_i := r
else
  P := L;
```

- consume A_i, A_j
  リソースA_iを消費する．すなわち，インデックス値A_iで指されるリソース節が有界リソースなら，自分自身のレベルと，同一の選択可能リソースに含まれていた他のリソース節のレベルをレジスタUの値に更新する．バックトラックに備えて，レベルを変更したリソースの表中の位置と元のレベル値をトレイルに積んでおく．また，リソースA_iのクロージャをA_jにセットする．
- execute_closure A_i
  クロージャA_iを実行する．すなわち，A_i=⟨CLO,c⟩とすると，レジスタSを$c+2$にセットし，modeフラグをreadにセットした後，HEAP[c]のアドレスにジャンプする．

述語p/nのコードの先頭に挿入されるリソース節の呼出しの命令列は図7のようになる．

述語p/nが呼び出されると，まずpickup_resourceが実行され，消費可能なリソース節がなければ直ちにプログラム節のコード(ラベルL')にジャンプする．消費可能なリソース節があれば，try_resourceで選択点フレームを作成しレジスタの値を保存した後(R1, R2はすでに次のリソースを指している)，consume命令でリソースのレベルを更新し，execute_closureでリソース節のコードを実行する．

リソース節のコードの実行が失敗するなどして，バックトラックしてきた場合は，restore_resource命令に実行が戻り，レジスタの値が復帰された後，pickup_resourceが実行される．消費可能なリソース節がなければ，trust_resourceによって選択点フレームが削除され，プログラム節のコードにジャンプする．消費可能なリソース節があれば，retry_resource_elseで選択点フレーム上のR1, R2を更新した後，consumeとexecute_closureを実行する．

再度バックトラックしてきた場合も，バックトラックポイントはラベルL_0のままなので，restore_resource命令に実行が戻ることになる．

以上のコードは，すべての述語に対して付け加えられるが，コード量の増加はごくわずか(述語ごとに8命令)である．また，実行上のオーバーヘッドについても，リソースの追加されていない述語に対しては，レジスタR1, R2が空リストにセットされているから，直ちにプログラム節のコードにジャンプすることになり，最小限だといえる．

ただし，後述のような最適化は可能である．

4.10 バックトラック

バックトラックの処理についても基本的にはWAMと同一になる．ただし，LLPAMでは，レジスタR, L, Uの値も各選択点フレームに保存する必要がある．

また，バックトラックの処理のために，変数への代入以外に以下の情報をトレイルに積む必要がある．

- add_res命令, add_exp_res命令において，シンボル表あるいはハッシュ表へのリソース節の追加時に，シンボル表あるいはハッシュ表のエントリと変更前のリソースリストの値
- consume命令, mid_with命令において，リソース

節のレベル変更時に，リソース節のインデックス値と変更前のレベル値

- end_imp, end_exp_imp 命令において，リソース節の out_of_scope フラグの変更時に，リソース節のインデックス値

4.11 最適化

WAMで行われているレジスタ割当て，末尾呼出し，インデクシングなどの最適化はLLPAMでもそのまま適用可能である．ただし，末尾呼出しについては，最後のゴールが$G_1 \,\&\, G_2$, $!G$, $R \multimap G$, $R \Rightarrow G$の場合は最適化できない．これは，end_with, end_bang などの命令が最終の call 命令の後に必要なためである．

LLPAM特有の最適化については，以下に述べる方法が考えられる．

4.11.1 リソース追加の最適化

LLPプログラムでは，$(R_1 \otimes R_2 \otimes \cdots R_n) \multimap G$ (あるいは$R_1 \multimap (R_2 \multimap \cdots (R_n \multimap G) \cdots)$) といった形で複数のリソースを一度に加える記述がしばしば現れる．これをそのままコンパイルしたコードは，begin_imp, end_imp の組がn重にネストし，Y レジスタもn個必要となる．

そこで，“end_imp $\mathtt{Y}_i$”をn回行う命令“end_imp $\mathtt{Y}_i$,n” (ただし，$\mathtt{Y}_i$の値を次のリソースを指すように増やしながら繰り返す) を導入し，以下のようなコード生成を行う．

begin_imp $\mathtt{Y}_i$
R_1中の有界リソース節追加のコード
mid_imp m_1
R_2中の有界リソース節追加のコード
mid_imp m_2
.....
R_n中の有界リソース節追加のコード
mid_imp m_n
ゴールGのコード
end_imp $\mathtt{Y}_i$, n

ただし，m_iはR_i中の有界リソース節の個数である．

4.11.2 リソースの決定的な選択

LLPでは，同一の述語名をプログラム節にもリソース節にも利用することが可能である．しかし，ほとんどのLLPプログラムでは，どちらか一方だけに利用していることが多い．

```
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
```

図8 述語p/nの最適化されたLLPAMコード (プログラム節なしの場合)

リソースとして利用していない述語名については，前述のリソース呼出しの命令列を省くことができる．

逆に，プログラム節が存在していない述語名については，リソース呼出しの命令列は必要となるが，以下のような最適化を考えることができる．

- 利用可能なリソースがただ一つしかない場合は，選択点フレームを作成せずに直ちに消費すればよい．
- 選択点フレームを作成した場合でも，最後に利用可能なリソースが一つだけ残った時点で選択点フレームを削除できる．

このために以下の命令を新たに導入する．

- if_no_resource L
  レジスタR1とR2中に，消費可能なリソースが残っているかどうか調べる．もしなければ，ラベルLにジャンプする．

述語p/nに対する最適化されたコードは図8のようになる．通常の述語p/nに対するコード(図7)と比較すると，二つのpickup_resource命令の直後にif_no_resource命令が追加され，また最後にtrust_resource命令が一つ追加されている．

4.12 LLPAMのコード例

例2.1中のreverse述語について，LLPAMのコード例を図9に示す．なお，rev述語についてはWAMの場合と全く同一のコードになる．

別のコード生成例として，例2.3で示したプログラム中のchooseおよびtest述語に対するコードを図10に

表2 Nクイーンの実行結果

N-Queens	SICStus (msec)	LLP (msec)	速度向上比
8 (全解)	170	52	3.27
9 (全解)	780	228	3.42
10 (全解)	3828	1004	3.81
11 (全解)	20202	4900	4.12
12 (全解)	114268	26100	4.38
13 (全解)	681656	145988	4.67
14 (全解)	4326485	883344	4.90
15 (第一解)	290	52	5.58
20 (第一解)	63488	8826	7.19
25 (第一解)	22532	2556	8.82

表3 ナイト・ツアー(5×5)の実行結果

	SICStus (msec)	LLP (msec)	速度向上比
Knight Tour	241404	89042	2.71

表4 Prologベンチマークの実行結果

プログラム名	SICStus (msec)	LLP (msec)	SWI (msec)
`boyer`	616	13432	2785
`browse`	770	1510	2352
`cal`	142	311	645
`chat_parser`	29	40	63
`ham`	580	1492	1286
`poly_10`	43	60	145
`queens_8` (全解)	56	100	288
`queens_10` (全解)	1271	2320	6978
`queens_16` (第一解)	777	1460	4567
`zebra`	39	68	66
実行時間比の平均	1.00	3.85	3.81
`boyer`を除いた実行時間比の平均	1.00	1.86	3.73

示す.

5 評価

LLPプログラムを抽象機械LLPAMへコンパイルするコンパイラと，LLPAMのエミュレータ(C言語で記述)を実装し，評価を行った(最適化については，ほとんど未実装である).

表2は，二つのN-クイーンプログラムの実行結果である．一つ目は，図2のPrologで記述したN-クイーンのプログラムについて，SICStus Prologコンパイラ処理系上(バージョン3.7.1, コンパクトコード)での実行速度を計測した．この場合，入出力コンテキストはリストで表現されており，リソース管理のための拡張を行っていないWAM抽象機械上での実装に対応するとみなせる．二つ目は，図1のLolliで記述したN-クイーンのプログラムについて，LLPコンパイラ処理系上(バージョ

```
% reverse(Xs,Zs) :- result(Zs) -<> rev(Xs,[]).
reverse/2:
    allocate 1                  % Y1の領域確保
    begin_imp Y1
    put_structure result/1, A3 % A3 = result(Zs)
    unify_value A2
    put_closure L, 1, A4        % クロージャ作成
    unify_value A2              % 自由変数Zs
    add_res A3, A4
    mid_imp 1
    put_constant '[]'/0, A2
    call rev/2                  % rev(Xs,[])
    end_imp Y1
    deallocate
    proceed

% result(Zs)のクロージャのコード
L:  unify_value A1              % ZsをA1と単一化
    proceed
```

図9 述語reverseのLLPAMコード

ン0.43)での実行速度を計測した．この場合，リソースはLLPAMのリソース表に登録され，ハッシュ表で検索される．

表2は，Nが8以上14以下については全解探索の実行時間(単位はミリ秒)，Nが15, 20, 25については第一解探索までの実行時間(単位はミリ秒)を示している．Nが8の場合ですでに3倍以上の速度であり，Nが大きくなるにつれて速度向上比も大きくなっている．

また，5×5のチェス盤上でのナイト・ツアー問題(ハミルトン経路探索の一種で，ナイトの駒がすべてのマスをちょうど一度ずつ訪れる経路を探索する問題)についても実行時間を計測した．表3は，5×5のチェス盤上での全解探索の実行時間(単位はミリ秒)を示しており，LLPの実行速度は，SICStus Prologと比較して，2.71倍速くなっている．

リソース節をクロージャを用いてコンパイルした場合と，コンパイルせずにインタプリタを介して実行する場合との比較も行った．リソース節のコンパイルすることで，リソース節にボディ部を含まないLLPプログラムでは10%程度の速度向上，リソース節にボディ部を含むプログラムでは30%程度の速度向上を得た．

また，LLPAMの拡張によるオーバーヘッドを計測するために，標準的なPrologプログラムのベンチマークも実行し，既存のPrologコンパイラ処理系との性能

```
% choose(Xs,Y,Zs)
choose/3:
     allocate 1                 % Y1の領域確保
     begin_exp_imp Y1
     put_structure test/1, A4   % A4 = test(_)
     unify_variable A6
     put_closure L, 1, A5       % クロージャ作成
     unify_value A2             % 自由変数Y
     add_exp_res A4, A5         % リソース追加
     mid_exp_imp 1
     put_value A3, A2
     call filter/2              % filter(Xs,Zs)
     end_exp_imp Y1
     deallocate
     proceed

% forall(X, (test(X) :- X>=Y))
L:   unify_variable A2          % Y
     execute '>='/2             % 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
```

図10 述語chooseおよびtestのLLPAMコード

比較も行った(表4参照)．商用であるSICStus Prolog(バージョン3.7.1, コンパクトコード)と比較すると，3.85倍遅くなっている．しかしながら，LLPのboyerベンチマークの実行時間が，他のコンパイラ処理系と比較して大きく遅いのは，インデキシング最適化(switch_on_structure命令とswitch_on_constant命令)が実装されていないためである．このboyerベンチマークを除いた場合，LLPの実行速度は，SICStus Prologより1.86倍遅い程度に抑えられており，また広く利用されているフリーのコンパイラ処理系のSWI-Prolog(バージョン3.2.6)と比較して，約2倍速くなっている．これらの結果から，LLPAMは過大なオーバーヘッドなしに拡張が行われていることを確認できた．

なお，表2，表3，表4中の実行時間(単位はミリ秒)の計測はすべてMMX Pentium 266MHz，メモリ128M

バイト，Linux OSの計算機上で行った．

6 おわりに

本論文では，直観主義線形論理に基づく論理型言語LLPのコンパイル方式について述べた．

線形論理型言語LLPは，Prologの自然な拡張になっており，論理式をリソースとして取り扱うことができる点を特徴とする．

LLPプログラムは，本論文で述べた抽象機械LLPAMの命令列にコンパイルされ実行される．

抽象機械LLPAMは，Prologコンパイラで広く用いられている抽象機械WAM (Warren Abstract Machine)の拡張になっており，リソース管理のための命令が追加されている．LLPAMのリソース管理方式は，LLPの計算モデルであるレベル付きIOモデルに基づいて設計されており，実行中にただ一つのリソース表を保持すれば良いように工夫されている．

また，リソース自体もコンパイルされており，リソースの検索はハッシュ表を通じて行われるため，線形論理型言語の特徴であるリソース論理式の追加や消費といった操作を非常に効率よく実現することができる．

N-クイーン問題のプログラムをPrologおよびLLPで記述し，それぞれをSICStus Prologコンパイラ処理系とLLPコンパイラ処理系で実行したところ，Nが8の場合ですでに3倍以上の速度であり，Nが大きくなるにつれて速度向上比も大きくなっている．これにより，LLPでのリソースを用いたプログラミングの有用性と，LLPAMの設計の妥当性を示すことができた．

本論文とHodasらとの共同論文[14]との最も大きな違いは，リソース論理式として，Aと$R_1 \,\&\, R_2$以外に，$G \multimap R$，$G \Rightarrow R$，$\forall x.R$が使用でき，事実だけでなく規則もリソースとして利用可能になっている点である．さらに，クロージャを導入することにより，リソース節自体のコンパイルが可能である点も異なっている．

一方，論文[14]では，ゴール$\top$の完全な取り扱いについて議論している．しかし，実際のプログラムではほとんどの場合,「消費可能なリソースのすべてを消費する」ことを意味しており，本論文ではこの意味に沿った実装方法を採用した．

本論文で述べたコンパイル方式は，線形論理型言語の特徴であるリソースに対して，その効率の良い処理方法を実現したものであり，Lolliを含め他の線形論理型言語のコンパイルにもそのまま適用可能と考える．ただし，Lolliの量化記号を含むゴール論理式のコンパイルには対応していない．特に，全称記号を含むゴール$\forall x.G$のコンパイルについては，λPrologでの実装方法(論文[18])の導入を検討する必要があると考える．また，LolliはλPrologの線形論理による拡張であり，高階の機能(higher-order quantification, unification of λ-term)を備えているが，この点に関しても本論文の対象範囲外としている．

謝辞

本稿執筆にあたり，有益なコメントを頂いた査読者の方に感謝いたします．

参考文献

[1] Aït-Kaci, H.: *Warren's Abstract Machine*, The MIT Press, 1991. (`http://www.isg.sfu.ca/~hak/documents/wam.html`).

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

[3] 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*, June 1998, pp. 32–45.

[4] Banbara, M. and Tamura, N.: Translating a Linear Logic Programming Language into Java, *Proceedings of ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, November 1999, pp. 19–39.

[5] Bratko, I.: *Prolog programming for artificial intelligence*, Addison-Wesley, 1986.

[6] Cervesato, I., Hodas, J. S., and Pfenning, F.: Efficient resource management for linear logic proof search, *Proceedings of the 1996 International Workshop on Extensions of Logic Programming*, Springer-Verlag LNAI 1050, March 1996, pp. 67–81.

[7] Girard, J.-Y.: Linear Logic, *Theoretical Computer Science*, Vol. 50(1987), pp. 1–102.

[8] Girard, J.-Y.: Linear Logic: Its Syntax and Semantics, *Advances in Linear Logic*(Girard, J.-Y., Lafont, Y., and Regnier, L.(eds.)), Cambridge University Press, 1995, pp. 1–42. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[9] Harland, J. and Winikoff, M.: Implementing the Linear Logic Programming Language Lygon, *Proceedings of the 1995 International Logic Programming Symposium*(Lloyd, J.(ed.)), Portland, Oregon, 1995, pp. 66–80.

[10] Hodas, J. S.: Lolli: An Extension of λProlog with Linear Context Management, *Workshop on the λProlog Programming Language*(Miller, D.(ed.)), Philadelphia, Pennsylvania, August 1992, pp. 159–168.

[11] Hodas, J. S.: Specifying Filler-Gap Dependency Parsers in a Linear-Logic Programming Language, *Proceedings of the Joint International Conference and Symposium on Logic Programming*(Apt, K.(ed.)), Washington, DC, November 1992, pp. 622–636.

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

[13] Hodas, J. S. and Miller, D.: Logic Programming in a Fragment of Intuitionistic Linear Logic, *Information and Computation*, Vol. 110,No. 2(1994), pp. 327–365. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[14] Hodas, J. S., Watkins, K., Tamura, N., and Kang, K.-S.: Efficient Implementation of a Linear Logic Programming Language, *Proceedings of the 1998 International Conference and Symposium on Logic Programming*, June 1998, pp. 145–159.

[15] Kobayashi, N. and Yonezawa, A.: ACL — A Concurrent Linear Logic Programming Paradigm, *Proceedings of the 1993 International Logic Programming Symposium*(Miller, D.(ed.)), Vancouver, Canada, MIT Press, October 1993, pp. 279–294.

[16] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A.: Uniform proofs as a foundation for logic programming, *Annals of Pure and Applied Logic*, Vol. 51(1991), pp. 125–157.

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

[18] Nadathur, G., Jayaraman, B., and Kwon, K.: Scoping Constructs in Logic Programming: Implementation Problems and their Solution, *Journal of Logic Programming*, Vol. 25(2)(1995), pp. 119–161.

[19] Nadathur, G. and Miller, D.: An Overview of λProlog, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*(Kowalski, R. A. and Bowen, K. A.(eds.)), Cambridge, Massachusetts, MIT Press, August 1988, pp. 810–827.

[20] 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, Nov. 1996, pp. 33–50.

[21] Tamura, N. and Kaneda, Y.: A WAM model for a linear logic programming language, *Proceedings of the 9th Symposium on Industrial Applications of Prolog (INAP'96)*, Oct. 1996, pp. 63–70. (in Japanese).

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

[23] Warren, D. H. D.: An abstract Prolog instruction set, Technical Report Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.

[24] 田村直之, 池田雄一: 線形論理型言語のコンパイラ処理系でのリソース管理方式について, **情報処理学会 プログラミング研究会報告** *No. 7*, 5月 1996, pp. 25–30.

[25] 田村直之, 平井崇晴, 吉川英男, 姜京順, 番原睦則: 直観主義時相線形論理における論理プログラミングについて, **情報処理学会論文誌: プログラミング**, Vol. 41,No. SIG 4 (PRO 7)(2000), pp. 11–23.

[26] 番原睦則, 姜京順, 田村直之: 線形論理型言語の Java 言語による処理系の設計と実装, **情報処理学会論文誌: プログラミング**, Vol. 40,No. SIG 10 (PRO 5)(1999), pp. 1–16.

[27] 姜京順, 番原睦則, 田村直之: 線形論理型言語の効率的なリソース管理モデル, **コンピュータソフトウェア**, Vol. 18,No. 0(2001), pp. 138–154.