時相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[†††]

線形論理に基づく論理型言語については，過去 10 年間にわたり数多くの言語が提案されている．これらの言語では，リソース（使用回数を制限されたプログラム節）を動的に追加，消費（使用）することが可能である．しかしながら，「リソースを消費する際の時間的順序を正確に記述する」というようなリソースの時間に依存した性質を表現することは困難であった．本稿では，直観主義時相線形論理に基づく論理型言語 TLLP の概要と，そのコンパイラ処理系のための抽象機械について述べる．TLLP 言語は Prolog と線形論理型言語 LLP の自然な拡張になっており，線形論理型言語のリソース概念に加え，どの時刻でリソースを使用するかといった時相性を記述できる．抽象機械は，TLLP のための効率的な計算モデルであるレベル付き IOT モデルに基づいて設計されており，WAM および LLP のための抽象機械 LLPAM の拡張になっている．拡張は，主として使用時刻を制限されたリソースの効率の良い管理，特にリソースの追加，消費のためである．現在，TLLP コンパイラ処理系のプロトタイプ版が稼働中であり，TLLP から LLP へのトランスレータ処理系と比較して，1.6 倍以上の高速化を実現している．

An Abstract Machine for a Compiler System of a Temporal Linear Logic Programming Language

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

A number of logic programming languages based on linear logic have been proposed in the last ten years. Although these languages can add and consume resources (limited-use clauses) dynamically, they run into difficulties with time-dependent properties of resources, in particular, when we have to describe precisely the order of the moments when some resources are consumed. This paper describes an abstract machine for a logic programming language based on a fragment of intuitionistic temporal linear logic, called TLLP. TLLP is a superset of Prolog and LLP which is an another linear logic programming language. TLLP allows to specify the time-dependent properties of resources. TLLP Abstract Machine is based on a level-based resource management system and is an extension of WAM and LLPAM (LLP Abstract Machine). Our extension is mainly for efficient timed-resource management: especially for resource addition and consumption. Our prototype compiler produces 1.6 or more times faster code compared with a TLLP to LLP translator.

1. はじめに

線形論理（linear logic）[5]に基づく論理型言語については，過去 10 年間にわたり数多くの言語が提案されており，LO [2]，ACL [11]，Lolli [9]，Lygon [6]，Forum [12]，および LLP [3),4),10),14),16),18)~20)]などの研究がある．

特に，本研究のベースとなる Lolli[☆] と LLP[☆☆] を含むいくつかの言語では，リソース（使用回数を制限された Prolog の節に相当する論理式）を動的に追加，使用（消費）することが可能である．しかしながら，「リソースを消費する際の時間的順序を正確に記述する」といったリソースの時間に依存した性質を表現することは困難であった．これは線形論理には時間の概念が直接的には入っていないためである．

近年，著者らは直観主義時相線形論理の体系

† 奈良工業高等専門学校
Nara National College of Technology
†† 釜山外国語大学校コンピュータ電子工学部（韓国）
Division of Electronic and Computer Engineering, Pusan University of Foreign Studies, Korea
††† 神戸大学工学部
Faculty of Engineering, Kobe University

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

ITLL [7),8)]に基づく論理型言語 TLLP [17)]を提案した．TLLP は Prolog と LLP の自然な拡張になっており，LLP の演算子に加え，次の時刻に 1 回だけ使用可能なリソースを表す様相演算子 ○，現在以降の任意の時刻に 1 回だけ使用可能なリソースを表す様相演算子 □，現在以降の任意の時刻に任意の回数使用可能なリソースを表す様相演算子！を含んでいる．このため，TLLP のリソースは，使用回数および使用時刻を制限された Prolog の節に相当する論理式と考えることもできる．

しかしながら，TLLP のための初期の計算モデルである IOT モデルは，インタプリタ処理系の開発には適していたものの，コンパイラ処理系の開発に適したものではなかった．さらに，IOT モデルに基づく TLLP インタプリタ処理系では，リソースはリスト構造で表現されているため，リソースを消費した場合，リスト構造の再構築が必要となり，効率の低下の原因となっている．

本稿では，TLLP 言語の概要と，そのコンパイラ処理系のための抽象機械について述べる．この抽象機械は，TLLP のための効率的な計算モデルであるレベル付き IOT モデルに基づいて設計されており，WAM [1),15)]および LLP のための抽象機械 LLPAM の拡張になっている．拡張は，主として使用時刻を制限されたリソースの効率の良い管理，特にリソースの追加および消費のためである．

現在，本稿で述べる抽象機械に基づく TLLP コンパイラ処理系☆のプロトタイプ版が稼働中であり，TLLP から LLP へのトランスレータ処理系と比較して，1.6 倍以上の高速化を実現している．

2. 時相線形論理型言語 TLLP

2.1 TLLP の構文

TLLP は直観主義時相線形論理の体系 ITLL の以下のようなフラグメントを用いる（A は原子論理式，$m \geq 1$）．

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

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．論理演算子 $\Rightarrow$ は直観主義的含意を表す（すなわち $P \Rightarrow Q \equiv !\,P \multimap Q$ である）．

ゴール論理式 $R \multimap G$ の実行により追加されるリソース論理式 R は，$\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ または $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ の形をしている☆☆．前者は n 時刻後に一度だけ使用できるリソースを表し，後者は n 時刻以降に一度だけ使用できるリソースを表す．以降，$R \multimap G$ によって追加されるリソースを**有界リソース**と呼ぶことにする．

ゴール論理式 $S \Rightarrow G$ の実行によって追加されるリソース論理式 S は，現在時刻以降の任意の時刻に 0 回以上何度でも使用できる．したがって，$S \Rightarrow G$ 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．

$S_1 \,\&\, \cdots \,\&\, S_m$ の形をしたリソース論理式は，S_1 から S_m のリソースのうち，どれか 1 つだけが使用可能であることから，**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 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．

プログラム中では，上記の定義に対応して以下のような記法を用いる（A は原子論理式，$m \geq 1$）．

$$
\begin{aligned}
C &::= A. \mid A\texttt{:-}G. \\
R &::= S_1\texttt{\&}\cdots\texttt{\&}S_m \mid \texttt{\#(}S_1\texttt{\&}\cdots\texttt{\&}S_m\texttt{)} \mid \texttt{@}R \\
S &::= A \mid G\texttt{-<>}A \mid \texttt{forall}\ X\texttt{\textbackslash}S \\
G &::= \texttt{true} \mid \texttt{erase} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid \\
&\qquad G_1\texttt{;}G_2 \mid R\texttt{-<>}G \mid S\texttt{=>}G \mid \texttt{!}G \mid \texttt{@}G
\end{aligned}
$$

ここで，C はプログラム節を表し，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意味する．

ゴールとして，`true`，A，G_1`,`G_2，G_1`;`G_2 だけを用いる TLLP プログラムは，構文的にも意味的にも Prolog と完全に一致する．また，演算子`#`も`@`も使用しない TLLP プログラムは，構文的にも意味的にも LLP と完全に一致する．TLLP は Lolli の重要な演算子の大部分を含んでいるが，ゴール論理式中の $\forall$ と $\exists$ が記述できないという制限がある．

2.2 TLLP プログラミング

TLLP ではちょうど一度しか使用できないという線形論理型言語のリソース概念に加え，どの時刻でリソースを使用するかという時相性を記述できる．すなわち，TLLP ではゴールの実行される時刻によって，消費できるリソースが異なるプログラムを記述できる．

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

☆☆ $\bigcirc^n P$ は $\underbrace{\bigcirc \cdots \bigcirc}_{n\text{ 個}} P$ を表すものとする．

本節では，TLLP プログラムの操作的な意味について，直観的な説明を行う．

- 原子論理式のゴール A は，リソースの消費かプログラム節の呼び出しを意味する（これらの選択は非決定的である．すなわちバックトラックによりすべての可能性が実行される）．
- ゴール `@`G は，時刻を 1 つ進めたうえでゴール G を実行する．
- ゴール R`-<>`G は，有界リソース R を追加した後，ゴール G を実行する．R は G 中ですべて消費されなければならない．
- ゴール S`=>`G は無限リソース S を追加した後，ゴール G を実行する．S は現在時刻以降の任意の時刻に 0 回以上何回でも使用できる．
- 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 もまた 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．
- ゴール G_1`;`G_2 は，Prolog のゴール G_1`;`G_2 と同様である．
- ゴール `!`G は，ゴール G と同様であるが，有界リソースは使用できない．
- ゴール `true` は Prolog の `true` と同様である．
- ゴール `erase` も Prolog の `true` と同様だが，現在時刻以降に消費可能なリソースのうち，いくつかを暗黙的に消費する．
- リソース A は，事実タイプのリソースを表す．すなわち A とマッチするゴールの実行によって消費される．
- リソース G`-<>`A（A`:-`G と書いてもよい）は，規則タイプのリソースを表す．すなわち A とマッチするゴールの実行によって消費されるが，同時にボディ部 G が実行される．
- 選択可能リソース S_1`&`$\cdots$`&`S_m は，S_1 から S_m のリソース節のうち，どれか 1 つだけが消費可能である．
- 有界リソース $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ は，n 時刻後に S_1 から S_m のリソース節のうち，どれか 1 つだけが消費可能である．
- 有界リソース $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ は，n 時刻以降に S_1 から S_m のリソース節のうち，どれか 1 つだけが消費可能である．

ゴール S`=>`G は，Prolog での `assert` に近いが，追加されているスコープは G 中に限られており，バックトラックすれば消去される点，S が自由変数を含むことが可能な点で異なっている．たとえばゴール（`forall` $X \backslash p(X)$）`=>`G の実行は，事実 `p(X)` を `assert` したうえで G を実行するのと同様である．一方，ゴール `p(X)=>`G の実行の場合は，変数 `X` が全称限量子で束縛されていないため，G の実行中に `X` の値を決めることが可能である．

コンウェイのライフ・ゲームなどのように，ある時点での状態から次の時点での状態を生成するプログラムは，状態をリソースとして表現することにより，TLLP で簡潔に記述できる．**図 1** は TLLP で記述したライフ・ゲームのプログラムである（出力のためのプログラム部分は省略してある）．

このプログラム中，述語`life_game(glider,P)` は，20×20 の大きさの盤上でグライダーと呼ばれる図形を第 1 世代として，第 `P` 世代まで計算する．リソース `b(I,J)` は，`(I,J)` の位置に現世代で石があることを表している．世代は時刻に対応しており，述語 `loop/1` が呼ばれるたびに 1 つ進む．各世代において，述語 `next(I,J)` は次世代に石を出現すべきときに成功し，その場合はリソース`@b(I,J)` を追加する．ただし，述語 `next` の実行で周囲の石が消費されてしまうのを防ぐため，negation as failure による否定を二重に付け，消費せずにチェックだけを行っている．すなわち，`next(I,J)` が成功するとき，`\+ \+ next(I,J)` は成功し，リソースは消費されない．

もちろん，リソースをリストで表現することは可能であるが，リストは逐次的なデータ構造であり，消費可能なリソースを検索するにはリストをスキャンする必要がある．また，リソースを削除するにはリストの再構築が必要である．既存の TLLP インタプリタ処理系では，まさにこのことを行っており，実行効率の低下の原因となっている．特に，この節で述べたライフ・ゲームのプログラムように，計算が進むにつれリソースが増え続ける場合，その実行効率の低下は大きいといえる．

3. リソース管理モデル

時相線形論理型言語 TLLP を実装するうえで最も重要なのが，リソースを効率良く管理する計算モデルの設計である．ここでは，まず TLLP のベースとなる線形論理型言語 Lolli および LLP のリソース管理モデルについて説明する．

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

```
life_game(glider, P) :-
        size(20) => period(P) =>
                    b(1, 2) -<>
                                b(2, 3) -<>
        b(3, 1) -<>b(3, 2) -<> b(3, 3) -<>
        loop.

loop :- loop(1).
loop :- erase.

loop(P) :- period(Q), P =< Q, !, loop(1, 1, P).
loop(_).

loop(I, J, P) :- size(N), I > N, !, P1 is P+1, @loop(P1).
loop(I, J, P) :- size(N), J > N, !, I1 is I+1, loop(I1, 1, P).
loop(I, J, P) :- \+ \+ next(I, J), !, J1 is J+1, @b(I, J) -<> loop(I, J1, P).
loop(I, J, P) :- J1 is J+1, loop(I, J1, P).

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).
```

図 1 TLLP で記述したライフ・ゲームのプログラム
Fig. 1 A Life Game program in TLLP.

リソースの取扱いにおいて，まず問題となるのがゴール $G_1 \otimes G_2$ の実行である．ゴール $G_1 \otimes G_2$ を線形論理の推論規則 R$\otimes$ に従って，下から上へ適用しようとすると，リソースを Δ_1 と Δ_2 に分割する必要がある．しかし，この分割の仕方は Δ_1 および Δ_2 中のリソースの総数を n とすると 2^n になり非決定性が大きい．

Hodas らは，この問題の解決法として，リソース分割を遅延的に行う **IO モデル**[9]を提案した．

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

ここで，G はゴール，I，O はリソース論理式と 1 を要素として持つリストであり，I を**入力コンテキスト**，O を**出力コンテキスト**と呼ぶ．ここで，1 はリソースが消費されたことを示す特別な記号である．このモデルでは，ゴールをリソースを消費する消費者と考える．すなわち，$G_1 \otimes G_2$ の実行では，まず I のうちのいくつかを使って G_1 を証明し，その後，残ったリソース M を使って G_2 を証明する．

IO モデルは，リソース分割を遅延的に行う点で優れたリソース管理モデルであるが，次に述べるような問題点がある．

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

IO モデルの規則 (&) から分かるように，ゴール $G_1 \,\&\, G_2$ の実行では，G_1 と G_2 の実行において同じ入力コンテキスト I と出力コンテキスト O を必要とするため，複数の入出力コンテキストを管理しなければならない．すなわち，入出力コンテキストを破壊的に書き換えながら計算を進めることはできない．したがって，IO モデルは高級言語上でのインタプリタ処理系の開発には適しているが，コンパイラ処理系の開発に適しているとはいえない．実際，Lolli は IO モデルに基づくインタプリタ処理系（SML 上で記述）として実装されている．

著者らは，この問題の解決法として，IO モデルを拡張した**レベル付き IO モデル**[20]を提案した．このモデルでは，実行中にただ 1 つ入出力コンテキストを保持するだけでよく，それを破壊的に書き換えることにより計算を進めることができる．

レベル付き IO モデルにおける入出力コンテキスト中の各リソースは，リソース論理式 R とそのレベル

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

図 2 レベル付き IO モデルの & の規則
Fig. 2 The & rule of IO-model with level indices.

を表す整数 ℓ の対 $\langle R,\ell\rangle$ で表される．レベル ℓ は，リソースの追加時に割り当てられ，R が無限リソースの場合は $\ell=0$，有界リソースの場合は $\ell\neq 0$ である．

レベル付き IO モデルのシーケントは，以下の形をしている．

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

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（**消費可能レベル**と呼ぶ）を表しており，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（**消費後レベル**と呼ぶ）を表しており，初期値は -1 である．G はゴール論理式である．I，O は入出力コンテキスト，すなわち対 $\langle R,\ell\rangle$ のリストである．

$\vdash_{L,U} I\,\{G\}\,O$ 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．**図 2** に (&) の規則を示す．この規則に従って，ゴール $G_1\,\&\,G_2$ は以下のようなステップで実行される．

(1) $\vdash_{L,U-1} I\,\{G_1\}\,M$
G_1 で I 中のどのリソースが消費されたか分かるように，消費後レベルを $U-1$ とし G_1 を実行する．

(2) $change_{U-1,L+1}(M,N)$
入出力コンテキスト M 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) $\vdash_{L+1,U} N\,\{G_2\}\,O$
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) $thinable_{L+1}(O)$
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 O 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．

このようにレベル付き IO モデルでは，実行中にただ 1 つ入出力コンテキストを保持し，これを破壊的に書き換えることにより計算を進めることができる．したがって，入出力コンテキストを一次元の表で管理し，ハッシュ表などを用いてリソースへのアクセスを高速にするなどの実装方法を容易にとることができる．実際，LLP 言語の抽象機械 LLPAM は，このレベル付き IO モデルに基づいて設計されており，リソースの高速な処理が実現されている．

3.1 TLLP コンパイラの設計

TLLP のためのリソース管理モデルとしては，IO モデルを拡張した **IOT モデル**[17]が提案されている．この IOT モデルではリソース分割を遅延的に行うことに加え，TLLP の特徴である消費時刻に制限のあるリソースを効率良く処理することができる．

IOT モデルでは，入出力コンテキスト中の各リソースは，リソース論理式 R とその消費可能時刻を表す非負整数 t の対 $\langle R,t\rangle$ で表される．消費可能時刻 t は，リソースの追加時に割り当てられ，R が無限リソースの場合は $t=0$ であり，$\langle !\,R,0\rangle$ となる．IOT モデルのシーケントは，$I\,\{G\}_T\,O$ の形をしており，IO モデルに現在時刻を表す非負整数 T（初期値は 0）を付加したものとなっている．

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

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

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

IOT モデルでは，リソースの追加は規則 $(\multimap)$ と $(\Rightarrow)$ によって行われ，規則 $(\bigcirc)$ によって時刻が 1 つ進められる．ここで，規則 $(\multimap)$ 中のリソース R は $S_1\,\&\cdots\&\,S_m$ または $\Box(S_1\,\&\cdots\&\,S_m)$ の形のリソース論理式であり，1 は IO モデルと同様にリソースが消費されたことを示す特別な記号である．

たとえば，$I\,\{G\}_T\,O$ の実行において，I 中で消費可能なリソースは，次の 3 つの形をしている．

- $\langle S_1\,\&\ldots\&\,S_m, T\rangle$
- $\langle \Box(S_1\,\&\ldots\&\,S_m), t'\rangle$（ただし $t'\leq T$）
- $\langle !\,S,0\rangle$

TLLP の処理系を実装するには，少なくとも以下の 3 つの方法が考えられる．

(1) インタプリタ処理系
(2) TLLP から LLP へのトランスレータ処理系
(3) TLLP コンパイラ処理系

(1)，(2) に関しては，文献 17) においてその実装方

法が述べられている．インタプリタ処理系は，IOT モデルの規則を Prolog で記述することにより実装されている．また，トランスレータ処理系では，TLLP の各述語に時刻を表す引数を追加することで，LLP の述語に変換している．しかし，これらの実装方法には以下のような問題点がある．

- IOT モデルに基づくインタプリタ処理系は，リソース分割を遅延的に行い，TLLP の特徴である消費時刻に制限のあるリソースを効率良く処理することができる点では優れているといえる．しかし，先に述べた IO モデルの場合と同様に，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ある．また，入出力コンテキストはリスト構造で表されるため，消費可能なリソースを検索するにはリストをスキャンする必要があり，リソースを削除するにはリストの再構築が必要である．そして，これらのことが実行効率低下の原因となっている．
- 現在のトランスレート方式は，TLLP の時刻を表す引数を各述語に追加することで LLP に変換するというシンプルな方法であり，変換された LLP プログラムは抽象機械 LLPAM の命令列にコンパイルされるため，リソースは一次元の表により効率良く管理される．しかし，この変換は TLLP のゴール $\top$ をそのまま LLP のゴール $\top$ に変換しているため，TLLP のゴール $\top$ を正しく実現できていない．つまり，TLLP の $\top$ は現在時刻以降に消費可能なリソースしか消費できないが，LLP の $\top$ はそれ以外のリソースも消費してしまう可能性がある．
- 現在のトランスレータ処理系では，$\top$ を含まない TLLP プログラムに関しては，効率良く実行することができる．しかし，$\top$ の問題以外にも細かな問題点が残っている．変換された LLP プログラムにおいて，リソースの消費可能性のチェックは，リソースを呼び出したうえで時刻を表す引数を取り出し，組込み述語により比較演算することによって行われる．つまり消費できる可能性のあるリソースを取り出す際に，時刻に関するチェックを行っていない．これは無駄な選択点フレームを作成することになり，実行効率低下につながる．

これらの点を改善するために，IOT モデルをレベル付き IO モデルによって拡張したレベル付き IOT モデルを設計し，(3) のコンパイラ処理系のための抽象機械の設計を行った．

3.2 レベル付き IOT モデル

レベル付き IOT モデルは，IOT モデルとレベル付き IO モデルの両方の特徴を持っており，TLLP の消費時刻に制限のあるリソースを効率良く処理できることに加えて，実行中にただ 1 つ入出力コンテキストを保持し，これを破壊的に書き換えることにより計算を進めることができる．そのため，コンパイラ処理系の開発に適したリソース管理モデルであるといえる．

レベル付き IOT モデルでは，入出力コンテキスト中の各リソースは，リソース論理式 R，消費可能時刻を表す非負整数 t，レベルを表す整数 ℓ との組 $\langle R, t, \ell \rangle$ で表される．消費可能時刻 t とレベル ℓ は，リソースの追加時に割り当てられ，R が無限リソースの場合は $t = \ell = 0$，有界リソースの場合は $t \neq 0$，$\ell \neq 0$ である．

レベル付き IOT モデルのシーケントは，以下の形をしている．

$$\vdash^{T}_{L,U} I \,\{G\}\, O$$

ここで，T は正整数で現在時刻を表しており，初期値は 1 である．L は正整数で消費可能なリソース論理式のレベル（**消費可能レベル**と呼ぶ）を表しており，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（**消費後レベル**と呼ぶ）を表しており，初期値は -1 である．G はゴール論理式である．I，O は入出力コンテキスト，すなわち組 $\langle R, t, \ell \rangle$ のリストである．

$\vdash^{T}_{L,U} I \,\{G\}\, O$ の実行においては，I 中で

- $\langle S_1 \,\&\, \ldots \,\&\, S_m, T, L \rangle$
- $\langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t', L \rangle$（ただし $t' \leq T$）
- $\langle S, 0, 0 \rangle$

を満たす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（無限リソースのときは 0 のまま）となるようにする．**図 3** にレベル付き IOT モデルの体系 IOTL を示す．

$pick^{T}_{L,U}(I, M, S)$ は，入出力コンテキスト I から，消費可能なリソース節 S を 1 つ取り出す．入出力コンテキスト M は，選択された S のレベルが U に変更されることを除けば I と同じである．ただし，S が無限リソースの場合はレベルは 0 のまま変更されない．

$change_{\ell,\ell'}(M, N)$ は，入出力コンテキスト M 中のレベルが ℓ であるリソースに対して，そのレベルを ℓ' に変更したものが N である．

$thinable_{\ell}(O)$ は，入出力コンテキスト O 中のレベルが ℓ であるリソースがすべて消費されたかどうかチェックする．

$subcontext^{T}_{U,L}(O, I)$ は，入出力コンテキスト I 中

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

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

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

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

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

（ただし，R は $S_1 \,\&\, \cdots \,\&\, S_m$ または $\Box(S_1 \,\&\, \cdots \,\&\, S_m)$ の形をしたリソース論理式）

$$\frac{\vdash^T_{L,U} [\langle S,0,0\rangle \mid I]\,\{G\}\,[\langle S,0,0\rangle \mid O]}{\vdash^T_{L,U} I\,\{S \Rightarrow G\}\,O}\ (\Rightarrow) \qquad \frac{\vdash^{T+1}_{L,U} I\,\{G\}\,O}{\vdash^T_{L,U} I\,\{\bigcirc G\}\,O}\ (\bigcirc)$$

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

図 3 体系 IOTL：命題 TLLP のレベル付き IOT モデル
Fig. 3 IOTL: IOT-model with level indices for propositional TLLP.

の時刻 T（すなわち，現在時刻）以降に消費可能なリソースのうち，いくつかを消費したものが O である．

以下は，*pick*，*change*，*thinable*，*subcontext* の各述語の詳細な定義である．

$pick^T_{L,U}([r_1,\ldots,r_n],[s_1,\ldots,s_n],S) \Longleftrightarrow$
次の (1) または (2) または (3) になる．
(1) ある i について，
$r_i = \langle S_1 \,\&\, \ldots \,\&\, S_m, T, L\rangle$，
$s_i = \langle S_1 \,\&\, \ldots \,\&\, S_m, T, U\rangle$，
$S_k = S$ となる．
その他の j $(j \neq i)$ は，$s_j = r_j$ となる．
(2) ある i について（ただし $t' \leq T$），
$r_i = \langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t', L\rangle$，
$s_i = \langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t', U\rangle$，
$S_k = S$ となる．
その他の j $(j \neq i)$ は，$s_j = r_j$ となる．
(3) ある i について，
$r_i = \langle S_1 \,\&\, \cdots \,\&\, S_m, 0, 0\rangle$，
$S_k = S$，
$s_j = r_j$ $(1 \leq j \leq n)$ となる．

$change_{\ell,\ell'}([r_1,\ldots,r_n],[s_1,\ldots,s_n]) \Longleftrightarrow$
各 $i = 1,2,\ldots,n$ について，
(1) $r_i = \langle R,t,\ell\rangle$ のとき，$s_i = \langle R,t,\ell'\rangle$，
(2) その他のとき，$s_i = r_i$

$thinable_\ell([r_1,\ldots,r_n]) \Longleftrightarrow$
すべての $r_i = \langle R,t,\ell'\rangle$ について，$\ell \neq \ell'$ となる．

$subcontext^T_{U,L}([s_1,\ldots,s_n],[r_1,\ldots,r_n]) \Longleftrightarrow$
各 $i = 1,2,\ldots,n$ について，
(1) $r_i = \langle S_1 \,\&\, \ldots \,\&\, S_m, t', L\rangle$（ただし $t' \geq T$）のとき，$s_i = \langle S_1 \,\&\, \ldots \,\&\, S_m, t', U\rangle$ または $s_i = r_i$
(2) $r_i = \langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t', L\rangle$（$t'$ は任意）のとき，$r_i = \langle \Box(S_1 \,\&\, \ldots \,\&\, S_m), t', U\rangle$ または $s_i = r_i$
(3) その他のとき，$s_i = r_i$

4. TLLP 抽象機械

TLLP のための抽象機械は，先に述べたレベル付き IOT モデルに基づいて設計されており，WAM[1),15)] および線形論理型言語 LLP の抽象機械 LLPAM[19)] の拡張になっている．

LLPAM はレベル付き IO モデル[20)] に基づいて設計された LLP 言語のための拡張 WAM であり，主に以下のような点が拡張されている．

- 新しく 3 つのデータ領域 `RES`（リソース表），`SYMBOL`（シンボル表），`HASH`（ハッシュ表）が追加されている．`HASH` と `SYMBOL` は主に `RES` に格納されたリソースへ高速にアクセスするために用いられる．`RES` の各エントリは，述語記号と第 1 引数をハッシュキーとして検索される．
- 新しく 5 つのレジスタ `R`，`L`，`U`，`R1`，`R2` が追加されている．`R` は `RES` の未使用領域の先頭を指す．

L と U 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 と R2 には，述語呼び出し時に HASH と SYMBOL から取り出した消費できる可能性のあるリソース節のリストが格納される．

- LLP 言語の特徴であるリソースを効率良く管理するための新しい命令が追加されている．

本章では，LLPAM からの拡張部分を中心に TLLP 抽象機械の詳細を述べる．

4.1 リソース表

RES 領域の各エントリには，以下のような情報が格納される．LLPAM と比較して，新たに time と box の 2 つのフィールドが追加されている．

```
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);
  time         : リソースの消費可能時刻 (整数);
  box          : □の有無 (ブール値)
  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，有界リソース $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ あるいは $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ 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．

time はレベル付き IOT モデルにおいて，各リソースに割り当てられた消費可能時刻である．すなわち，$\bigcirc^n R \multimap G$（ただし，R は $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ または $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$）で追加された有界リソースの場合は，現在時刻 T に n を加えた値 $T+n$，$S \Rightarrow G$ で追加された無限リソースの場合は 0 がセットされる．

box フラグは □ の有無を表すブール値であり，$\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ が追加された場合は false，$\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ の場合は true がセットされる．

level は，レベル付き IOT モデルにおいて，各リソースに割り当てられたレベル値である．すなわち，$\bigcirc^n R \multimap G$（ただし，R は $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ または $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$）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，$S \Rightarrow G$ 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ブール値である（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ときには，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，リソース節のコンパイルコードへのポインタ，詳しくはコンパイルコードと変数環境（自由変数の個数と自由変数への参照）から構成されるクロージャへのポインタである．これは，リソース節は自由変数を含む可能性があるため，通常のプログラム節と同様にコンパイルできないためである．クロージャに関しては文献 3)，19) に述べられており，本稿で詳細は述べない．

4.2 レジスタ

LLPAM のレジスタに加えて，新しいレジスタ TI を追加する．TI はレベル付き IOT モデルの現在時刻 T である．TI の値は上述の time フィールドをセットする際に使われる．また，この値はリソース検索のためのハッシュキー，リソースの消費可能性のチェックにも使われる（後述の pickup_timed_resource 命令を参照）．

4.3 ゴール $\bigcirc G$ のコード

ゴール $\bigcirc G$ のコードは，以下のようになる．

```
begin_next
ゴール G のコード
end_next
```

各命令の処理は以下のとおりである

- begin_next
  レジスタ TI をインクリメントする．
- end_next
  レジスタ TI をデクリメントする．

4.4 ゴール $\top$ のコード

ゴール $\top$ に対しては，1 命令のコードを生成する．

```
top
```

ゴール $\top$ は，現在時刻以降に消費可能なリソースのいくつかを暗黙的に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,「現在時刻以降に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，本稿では命令 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 でない現在時刻以降に消費可能なすべてのリソース節のレベルを U に変更する．

4.5 リソース節の追加のためのコード

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 はつねに $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ または $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ 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．追加の際，フィールド time の値を TI + n にセットし，R が $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ のときは box フラグを false に，R が $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ のときは true にセットする．
(3) 追加された各リソース節 S_i に対して，フィールド s と n の値をセットする．
(4) ゴール G を実行する．
(5) 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．
(6) $\mathtt{Y}_i$ の位置のリソース節のフィールド n の値を m として，$\mathtt{Y}_i$ から $\mathtt{Y}_i + m - 1$ 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，$\mathtt{Y}_i$ の値をトレイルに積んでおく．

ゴール $S \Rightarrow G$ の実行の概要は，ステップ (2) で無限リソース節 S が 1 つだけ追加されること（time は 0，box は true とする），無限リソースであるためステップ (5) の処理が必要ないことを除けば，$R \multimap G$ と同じである．

ステップ (2) のリソース節 S の追加のためには，以下のような命令が使われる．

- add_exact_timed_res $\mathtt{A}_i$, $\mathtt{A}_j$, n
  有界リソース $\bigcirc^n(S_1 \,\&\, \cdots \,\&\, S_m)$ 中のリソース節 S_i をリソース表 RES に追加する．ただし，$\mathtt{A}_i$ と $\mathtt{A}_j$ は各々 S_i のヘッド部とクロージャであり，n は ○ の個数である．

```
RES[R].time := TI + n;
RES[R].box := false;
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 1 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;
```

- add_timed_res $\mathtt{A}_i$, $\mathtt{A}_j$, n
  有界リソース $\bigcirc^n \Box(S_1 \,\&\, \cdots \,\&\, S_m)$ 中のリソース節 S_i をリソース表 RES に追加する．ただし，$\mathtt{A}_i$ と $\mathtt{A}_j$ は各々 S_i のヘッド部とクロージャであり，n は ○ の個数である．

```
RES[R].time := TI + n;
RES[R].box := true;
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 1 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;
```

- add_exp_res $\mathtt{A}_i$, $\mathtt{A}_j$
  ゴール $S \Rightarrow G$ の実行によって追加される無限リソース節 S をリソース表 RES に追加する．ただし，$\mathtt{A}_i$ は S のヘッド部，$\mathtt{A}_j$ はクロージャである．

```
RES[R].time := 0;
RES[R].box := true;
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 1 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;
```

ステップ (1), (3), (5), (6) のために使われる命令は，LLPAM の命令をそのまま利用できるので，ここでは省略した．

4.6 述語に対するコード

TLLP の述語呼び出しは，LLP 同様，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，プログラム節のリソース節の両方について，すべての実行を試みる必要がある．

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．

```
p/n: pickup_timed_resource p/n, A_{n+1}, L'
     try_timed_resource L_1
L_0: restore_timed_resource
     pickup_timed_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 4 述語 p/n のためのコード
Fig. 4 Code for p/n.

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 を消費したことを示すため，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，以下の新しい命令（一部の命令は LLPAM 命令をそのまま利用できる）を使用し，それらで記述した命令列（**図 4**）を述語 p/n の通常のコードの先頭に挿入する．

- try_timed_resource L
  WAM 命令 try Lと同じであるが，レジスタ TI，R，L，U，R1，R2 の値も選択点フレームに保存する．
- restore_timed_resource
  現在の選択点フレームから，TI，R，L，U，R1，R2 を含め各レジスタの値を取り出す．
- retry_resource_else L
  同名の LLPAM 命令と同じであり，現在の選択点フレーム中の R1，R2 フィールドおよび BP（バックトラックポイント）に，現在の R1，R2 の値および L を代入する．
- trust_resource L
  同名の LLPAM 命令と同じであり，選択点フレームを 1 つ前に戻し，ラベル L にジャンプする．
- pickup_timed_resource p/n, A_i, L

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

/* RES[i] が消費可能かどうかチェックする関数 */
function consumable(i) : boolean
  begin
     if RES[i].out_of_scope then
         return false;
     l := RES[i].level;
     t := RES[i].time;
     box := RES[i].box;
     return l = 0 or
            (l = L ∧ t = TI) or
            (l = L ∧ box ∧ t <= TI)
  end;
```

図 5 pickup_timed_resource 命令
Fig. 5 The pickup_timed_resource instruction.

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（**図 5** 参照）．

- consume A_i, A_j
  同名の LLPAM 命令と同じであり，リソース A_i を消費する．すなわち，RES[A_i] に格納されているリソース節が有界リソースなら，自分を含めた同じ選択可能リソースに含まれるすべてのリソース節のレベルをレジスタ U の値に更新する．また，リソース RES[A_i] のクロージャを A_j にセットする．
- execute_closure A_i
  同名の LLPAM 命令と同じであり，クロージャ A_i を実行する．

述語 p/n を呼び出すと，pickup_timed_resource が最初に実行され，消費可能なリソース節がなければただ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，

```
:- op(1060, xfy, (&)).
:- op( 900,  fy, [!,#]).

life_game(glider,P,T,R0,R) :-
    loop(T,
         [(!size(20),0),(!period(P),0),(b(1,2),T),(b(2,3),T),(b(3,1),T),(b(3,2),T),(b(3,3),T)|R0],
         [(!size(20),0),(!period(P),0),         1,         1,         1,         1,         1|R]).

loop(T,R0,R) :- loop(1,T,R0,R).
loop(T,R0,R) :- subcontext(T,R,R0).

loop(P,T,R0,R) :- proveA(period(Q),T,R0,R1), P =< Q, !, loop(1,1,P,T,R1,R)
loop(_,_T,R,R).

loop(I,_J,P,T,R0,R) :- proveA(size(N),T,R0,R1), I > N, !, P1 is P+1, T1 is T+1, loop(P1,T1,R1,R).
loop(I, J,P,T,R0,R) :- proveA(size(N),T,R0,R1), J > N, !, I1 is I+1, loop(I1,1,P,T,R1,R).
loop(I, J,P,T,R0,R) :- \+ \+ next(I,J,T,R0, _), !, J1 is J+1, T1 is T+1,
                       loop(I,J1,P,T,[(b(I,J),T1)|R0],[1|R]).
loop(I, J,P,T,R0,R) :- J1 is J+1, loop(I,J1,P,T,R0,R).

next(I,J,T,R0,R) :- proveA(b(I,J),T,R0,R1), !, count(I,J,C,T,R1,R), 2 =< C, C =< 3.
next(I,J,T,R0,R) :- count(I,J,C,T,R0,R), C = 3.

count(I1,J1,C,T,R0,R) :- 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,T,R0,R).

count_b([],           0,_T,R, R) :- !.
count_b([(I,J)|IJs],  C, T,R0,R) :- proveA(b(I,J),T,R0,R1), !, count_b(IJs,C1,T,R1,R), C is C1+1.
count_b([(_I,_J)|IJs],C, T,R0,R) :- count_b(IJs,C,T,R0,R).

proveA(A,T,I,O) :- pick(T,I,O,A).

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

図 6 比較に用いた Prolog で記述したライフ・ゲームのプログラム
Fig. 6 A Life Game program in Prolog.

try_timed_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_timed_resource 命令に実行が戻り，レジスタの値が復帰された後，pickup_timed_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

```
life_game(glider,A,B) :-
        (forall (C\size(20,C)))=>
        (forall (D\period(A,D)))=>
         b(1,2,B)-<>b(2,3,B)-<>b(3,1,B)-<>b(3,2,B)-<>b(3,3,B)-<>
         loop(B).

loop(A) :- loop(1,A).
loop(A) :- erase.

loop(A,B) :- period(C,B),A=<C, !, loop(1,1,A,B).
loop(A,B).

loop(A,B,C,D) :- size(E,D),A>E,!,F is C+1,G is D+1,loop(F,G).
loop(A,B,C,D) :- size(E,D),B>E,!,F is A+1,loop(F,1,C,D).
loop(A,B,C,D) :- \+ \+next(A,B,D),!,E is B+1,F is D+1,b(A,B,F)-<>loop(A,E,C,D).
loop(A,B,C,D) :- E is B+1,loop(A,E,C,D).

next(A,B,C) :- b(A,B,C), !, count(A,B,D,C), 2=<D, D=<3.
next(A,B,C) :- count(A,B,D,C), D=3.

count(A,B,C,D) :- E is A-1,F is A+1,G is B-1,H is B+1,
        count_b([(E,G),(E,B),(E,H),(A,G),(A,H),(F,G),(F,B),(F,H)],C,D).

count_b([],0,A) :- (!).
count_b([(A,B)|C],D,E) :- b(A,B,E),!,count_b(C,F,E),D is F+1.
count_b([(A,B)|C],D,E) :- count_b(C,D,E).
```

図 7 比較に用いた LLP で記述したライフ・ゲームのプログラム
Fig. 7 A Life Game program in Prolog.

する．

再度バックトラックしてきた場合も，BP は L_0 のままなので，restore_timed_resource 命令に実行が戻ることになる．

5. 性能評価

現在，本稿で述べた TLLP 抽象機械に基づくコンパイラ処理系を開発中であり，コンパイラ（SICStus Prolog で記述）およびエミュレータ（C 言語で記述）のプロトタイプ版が稼働している．プロトタイプ版はインデキシングの最適化は行っているが，その他の最適化は未実装である．

前述のライフ・ゲームのプログラム（図 1）をベンチマークとして，プロトタイプ版を用いてリソース管理方式の評価を行った．

比較に用いたのは，**図 6** の Prolog プログラムを SICStus Prolog（バージョン 3.7.1，コンパクトコード）でコンパイルしたもの（以下，Prolog 版と呼ぶ）と，**図 7** の LLP プログラムを LLP コンパイラ処理系（バージョン 0.45，LLPAM コード）でコンパイルしたもの（以下，トランスレート版と呼ぶ）の 2 つである．

Prolog プログラムは，IOT モデルに基づく TLLP インタプリタを部分評価系で処理したものとほぼ同等であり，リソースはリストで表現している．LLP プログラムは，TLLP から LLP へのトランスレート方式[17]に基づいた処理系を作成し，図 1 の TLLP プログラムを LLP プログラムへ変換したものであり，リソースはコンパイルされ，LLPAM のリソース表により効率良く管理される．

第 10 世代までの実行時間は，プロトタイプ版が 380 ミリ秒，トランスレート版が 613 ミリ秒，Prolog 版が 9403 ミリ秒であり，トランスレート版より 1.6 倍，Prolog 版より 24 倍の高速化が実現できた（**表 1** 参照）．また，世代が大きくなるにつれて速度向上比も大きくなっている．

次に，拡張によるオーバ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比較も行った（**表 2** 参照）．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，2 倍速くなっている．また，商用である SICStus Prolog（バージョン 3.7.1，コンパクトコード）と比較しても，1.9 倍遅い程度に抑えられている．

さらに，LLP プログラムのベンチマーク（5 種類）も実行し，LLP コンパイラ処理系（バージョン 0.45）

表 1 ライフゲームの実行結果（20 × 20 盤上のグライダー）
Table 1 Results of Life Game benchmark (glider on 20 × 20 board).

グライダー	TLLP		LLP		Prolog	
世代数	ミリ秒	（比率）	ミリ秒	（比率）	ミリ秒	（比率）
10	380	(1.0)	613	(1.6)	9,403	(24.7)
20	773	(1.0)	1,640	(2.1)	30,570	(39.6)
30	1,183	(1.0)	3,070	(2.6)	63,190	(53.4)
40	1,603	(1.0)	4,953	(3.1)	107,503	(67.1)
50	2,050	(1.0)	7,233	(3.5)	163,293	(79.7)

表 2 Prolog ベンチマークの実行結果
Table 2 Results of Prolog benchmarks.

プログラム名	SICStus（ミリ秒）	TLLP（ミリ秒）	SWI（ミリ秒）
browse	1,310	2,266	3,498
cal	197	517	916
chat_parser	40	60	93
ham	970	2,490	2,218
poly_10	68	89	212
queens_8 (全解)	84	154	442
queens_10 (全解)	1,882	3,578	10,686
queens_18 (第 1 解)	5,622	11,328	35,543
zebra	57	96	104
実行時間比の平均	1.0	1.9	3.8

との性能比較も行ったところ，各ベンチマークの実行速度比の平均は 1%未満であった．

これらの結果から，TLLP 抽象機械は過大なオーバ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 1，表 2 中の実行時間（単位はミリ秒）の計測はすべて Pentium 200 MHz，メモリ 98 M バイト，Linux OS の計算機上で行った．

6. おわりに

本稿では，直観主義時相線形論理の体系 ITLL に基づく論理型言語 TLLP のコンパイル方式について述べた．

TLLP 言語は Prolog と線形論理型言語 LLP の自然な拡張になっており，線形論理型言語のリソース概念に加え，どの時刻でリソースを使用するかといった時相性を記述できる．

TLLP プログラムは，本稿で述べた TLLP 抽象機械の命令列にコンパイルされ実行される．この抽象機械は，TLLP のための効率の良いリソース管理モデルであるレベル付き IOT モデルに基づいて設計されており，Prolog のための抽象機械 WAM および LLP のための抽象機械 LLPAM の拡張になっている．

ライフ・ゲームをベンチマークとして，TLLP で記述したプログラムを TLLP コンパイラ処理系で実行し，その結果を LLP プログラム（TLLP から LLP へのトランスレータ処理系によって変換したもの）を LLP コンパイラ処理系で実行した結果，および Prolog で記述したプログラム（TLLP インタプリタを部分評価系で処理したものとほぼ同等）を SICStus Prolog コンパイラ処理系で実行した結果と比較すると，第 10 世代までの実行時間は，トランスレート版より 1.6 倍速く，Prolog 版より 24 倍速い．また，世代が大きくなるにつれて速度向上比も大きくなっていく．

本稿では，LLPAM からの拡張部分を中心に TLLP 抽象機械の詳細を述べたが，その他のゴール $G_1 \,\&\, G_2$，$!G$ のコンパイル，述語のコードに対する最適化などは，LLPAM と同様に行うことができる．

現在，ゴール $\top$ の操作的意味を「現在時刻以降に消費可能なリソースのすべてを消費する」としているが，完全な取扱いのためには，文献 10) にある手法の拡張を考える必要がある．

また，TLLP は Lolli の重要な演算子の大部分を含んでいるが，量化記号を含むゴール論理式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[13]の導入を検討する必要があると考える．

TLLP のベースとなる直観主義時相線形論理の体系 ITLL は，直観主義論理，直観主義線形論理，直観主義時相論理を含んでおり，その表現力は大きい．また，時間ペトリネットを ITLL でエンコーディングできることも示されている．よって，ITLL の自動証明系を TLLP 上で開発することが今後の課題である．

謝辞 最後に，査読者の方々，ならびにプログラミング研究会において貴重なご意見をくださった皆様に

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, pp.445–473 (1991).
3) Banbara, M. and Tamura, N.: Compiling Resources in a Linear Logic Programming Language, *Proc. JICSLP'98 Post Conference Workshop 7 on Implementation Technologies for Programming Languages based on Logic*, Sagonas, K. (Ed.), pp.32–45 (1998).
4) Banbara, M. and Tamura, N.: Translating a Linear Logic Programming Language into Java, *Proc. ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Carro, M., Dutra, I., et al. (Eds.), pp.19–39 (1999).
5) Girard, J.-Y.: Linear Logic, *Theoretical Computer Science*, Vol.50, pp.1–102 (1987).
6) Harland, J. and Winikoff, M.: Implementing the Linear Logic Programming Language Lygon, *Proc. 1995 International Logic Programming Symposium*, Portland, Oregon, Lloyd, J. (Ed.), pp.66–80 (1995).
7) Hirai, T.: An Application of Temporal Linear Logic to Timed Petri Nets, *Proc. Petri Nets'99 Workshop on Applications of Petri Nets to Intelligent System Development*, pp.2–13 (1999).
8) Hirai, T.: Propositional Temporal Linear Logic and Its Application to Concurrent Systems, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E83-A, No.11, pp.2219–2227 (2000).
9) Hodas, J.S. and Miller, D.: Logic Programming in a Fragment of Intuitionistic Linear Logic, *Information and Computation*, Vol.110, No.2, pp.327–365 (1994). Extended abstract in the Proc. 6th Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.
10) Hodas, J.S., Watkins, K., Tamura, N. and Kang, K.-S.: Efficient Implementation of a Linear Logic Programming Language, *Proc. 1998 Joint International Conference and Symposium on Logic Programming*, Jaffar, J. (Ed.), pp.145–159, The MIT Press (1998).
11) Kobayashi, N. and Yonezawa, A.: ACL—A Concurrent Linear Logic Programming Paradigm, *Proc. 1993 International Logic Programming Symposium*, Vancouver, Canada, Miller, D. (Ed.), pp.279–294, MIT Press (1993).
12) Miller, D.: A Multiple-Conclusion Specification Logic, *Theoretical Computer Science*, Vol.165, No.1, pp.201–232 (1996).
13) Nadathur, G., Jayaraman, B. and Kwon, K.: Scoping Constructs in Logic Programming: Implementation Problems and Their Solution, *Journal of Logic Programming*, Vol.25(2), pp.119–161 (1995).
14) Tamura, N. and Kaneda, Y.: Extension of WAM for a linear logic programming language, *2nd Fuji International Workshop on Functional and Logic Programming*, Ida, T., Ohori, A. and Takeichi, M. (Eds.), pp.33–50, World Scientific (1996).
15) Warren, D.H.D.: An abstract Prolog instruction set, Technical Report Technical Note 309, SRI International, Menlo Park, CA (1983).
16) 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, pp.25–30 (1996).
17) 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), pp.11–23 (2000).
18) 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), pp.1–16 (1999).
19) 番原睦則，姜 京順，田村直之：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，コンピュータソフトウェア，Vol.18, No.1, pp.39–60 (2001).
20) 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, pp.138–154 (2001).

(平成 13 年 3 月 12 日受付)
(平成 13 年 6 月 21 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等の研究に従事．

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 年同大学院自然科学研究科博士課程前期情報知能専攻修了．2000 年同大学院自然科学研究科博士課程後期知能科学専攻修了．博士（工学）．2000 年より釜山外国語大学校（韓国）理工大学コンピュータ電子工学部講師．線形論理（linear logic），論理プログラミング等の研究に従事．

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，制約プログラミング等に興味を持っている．