直観主義線形論理型言語LLPとそのコンパイラ処理系

田村，直之
番原，睦則

(Citation)
コンピュータソフトウェア,30(2):83-89

(Issue Date)
2013

(Resource Type)
journal article

(Version)
Version of Record

直観主義線形論理型言語LLPとそのコンパイラ処理系

田村 直之　番原 睦則

1 はじめに

1987 年に Girard が発表した線形論理 (Linear logic) [6] は，計算機科学への応用が期待されている比較的新しい論理体系である．

本稿では，直観主義線形論理に基づいた論理型プログラミング言語である LLP およびその処理系について紹介する [4] [5] [10] [11]．プログラミング言語としての LLP は，論理型プログラミング言語として広く知られている Prolog の拡張として設計されており，基本的には Prolog のプログラムはそのまま LLP 処理系で実行できる．さらに線形論理の「リソースを意識した論理」という特徴を利用すれば，後述の図 3 に示す N-クイーンのプログラム例のように，Prolog よりも効率の良いプログラムが作成可能である．

LLP 処理系には，抽象機械 LLPAM へのコンパイラ処理系 (LLP コンパイラ) [4][†1] と Java へのトランスレータ処理系 (Prolog Cafe) [5][†2] がある．コンパイラ処理系で用いている抽象機械 LLPAM は，Prolog コンパイラで広く用いられている WAM [18] を拡張したものであり，LLP プログラムの高速な実行が可能である．一方，Prolog Cafe は Java および Java コンパイラが動作するプラットフォームで利用可能という特徴を持ち，処理系としての完成度も高い．しかし最新版は，ISO Prolog 準拠を重視し LLP で拡張された機能には対応していないため，本稿では詳細を述べない．

以下では，線形論理と線形論理型言語の概要を述べた後，LLP の言語上の特徴について例を中心に説明し，最後に LLP コンパイラ処理系について紹介する．

2 線形論理と線形論理型言語

2.1 線形論理の概要

線形論理は計算機科学の基礎理論として適した様々な特徴を持っているが，そのうちの 1 つが **リソースを意識した論理** (resource-conscious logic) という点である．

たとえば，命題 P, Q, R がそれぞれ「100 円を持っている」,「100 円のコーヒーが飲める」,「100 円の紅茶が飲める」を表しているとする．また,「 P ならば Q 」および「 P ならば R 」を共に仮定する．すると，これまでの論理 (古典論理や直観主義論理) では，この 2 つから自動的に「 P ならば Q かつ R 」すなわち「100 円持っていればコーヒーと紅茶が飲める」が導かれる．これは，これまでの論理では一度仮定した P を何度でも使用できるためである．

一方，線形論理では仮定をリソースとみなした推論が可能になる．すなわち「 P ならば Q 」と「 P ならば R 」から「 P かつ P ならば Q かつ R 」は導かれるが「 P ならば Q かつ R 」は導かれない．これを線形論理の論理式で表した場合，$P \multimap Q$ と $P \multimap R$ から $(P \otimes P) \multimap (Q \otimes R)$ が導かれ，$P \multimap (Q \otimes R)$ は導かれないことに相当する．ここで，演算子 $\multimap$ は **線形含意** (linear implication) と呼ばれ，演算子 $\otimes$ は

Intuitionistic Linear Logic Programming Language LLP and its Compiler Systems.

Naoyuki Tamura, Mutsunori Banbara, 神戸大学情報基盤センター, Information Science and Technology Center, Kobe University.

コンピュータソフトウェア, Vol.30, No.2 (2013), pp.83–89.
2012 年 3 月 5 日受付.

†1 http://bach.istc.kobe-u.ac.jp/llp/

†2 http://kaminari.istc.kobe-u.ac.jp/PrologCafe/

乗法的連言 (multiplicative conjunction) と呼ばれる．上の例から乗法的連言は「同時的な AND」を表していると考えられる．すなわち Q および R を同時に得るためには，2 つの仮定 P が必要となる．なお，論理定数 1 が $\otimes$ の単位元である．

一方 **加法的連言** (additive conjunction) $Q \,\&\, R$ は「選択的な AND」を表す．$P \multimap Q$ および $P \multimap R$ から $P \multimap (Q \,\&\, R)$ を導くことができるが，これは 1 つの仮定 P から，Q と R どちらでも (ただしどちらか一方を) 得ることができることを表す．論理定数 $\top$ が $\&$ の単位元である．

オフコース様相演算 (of-course modality) $!P$ を用いれば，これまでの論理と同様に 0 回以上何度でも利用可能な仮定を表現できる．$P \multimap Q$ および $P \multimap R$ から $!P \multimap (Q \otimes R)$ を導くことができるが，これは仮定 P を 0 回以上何度でも利用できるためである．

このように線形論理には豊富な演算子が用意されており，高い表現力を持っている．ただし線形論理の詳細については本稿の範囲外である．文献[6] などを参照されたい．

2.2 線形論理型言語

線形論理に基づいた論理型言語の設計や処理系の開発は，線形論理の応用として活発な分野の 1 つであり LO [2], LinLog [1], ACL [12], Lolli [8], Lygon [7], Forum [13], 筆者らによる LLP [4][10][11], TLLP [3] 等の研究がある．また，線形論理型言語の良いサーベイとしては Miller による解説 [14] がある．

これらの言語の多くの計算モデルは，Miller らのユニフォーム証明 [15] の考えに基づいている．直観主義シーケント計算におけるユニフォーム証明とは，右辺が原子論理式でないシーケントは必ず右辺への導入規則の結果となっているようなカット無しの証明をいう．したがって，与えられたシーケントに対するユニフォーム証明の探索は，シーケントの右辺の論理式 (ゴール論理式) の形により適用すべき上の規則が決まるという意味でゴール指向となる．このとき，ユニフォーム証明が必ず存在するような論理式の集合が論理型言語を定め，ユニフォーム証明の探索が計算に対応する．たとえば，1 階論理のホーン節は必ずユニフォーム証明を持ち，ユニフォーム証明の探索は Prolog プログラムの実行過程 (すなわち SLD 導出) に対応する．

上記の線形論理型言語のうち Lolli と LLP は直観主義線形論理のユニフォーム証明に基づいている．LLP を拡張した TLLP は直観主義線形論理と時相論理を融合した直観主義時相線形論理のユニフォーム証明に基づいており，時刻毎のリソースを表現可能である．古典線形論理に基づいた Forum, LinLog, Lygon はより豊富な表現力を持っているが，非決定性も大きく，より複雑な証明探索を必要とする．LO および ACL も古典線形論理を用いているが，それぞれオブジェクト指向計算，並行計算を実現するという目的で設計されている．

これらの言語のうち Lolli と LLP および LLP を拡張した TLLP が Prolog と最も近い言語設計となっている．また，LLP および TLLP のみが WAM に基づいたコンパイラ処理系を有している．

3 LLP の言語

LLP は直観主義線形論理に基づいた論理型言語であり，基本的には Prolog の拡張になっている [10][11]. LLP 言語は，Hodas らによる Lolli [8] を元にしているが，コンパイラ処理系の実現を主眼として，Lolli よりも少し狭いサブセットを採用している．しかし，線形論理型言語のプログラミングに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．

以下では LLP 言語の構文について説明する．まず **ゴール論理式** G および **リソース論理式** R の構文を以下のように定める (ただし A は原子論理式).

$$
\begin{array}{lcl}
G & ::= & \texttt{true} \mid \texttt{erase} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid \\
 & & G_1\texttt{;}G_2 \mid \texttt{!}G \mid R\texttt{-<>}G \mid R\texttt{=>}G \\
R & ::= & A \mid R_1\texttt{\&}R_2 \mid G\texttt{-<>}R \mid \texttt{forall [}X\texttt{]\textbackslash}R
\end{array}
$$

プログラム中の記法と線形論理の論理式との対応は図 1 の通りである．また，演算子の優先順位は弱いものから順に，“`forall`”, “`\`”, “`;`”, “`&`”, “`,`”, “`-<>`”, “`=>`”, “`!`” であり，2 項演算子は右結合的とする．

ここで，ゴール論理式は Prolog の節のボディ部に現れるゴールに対応する論理式であり，リソース論理

LLP での記法	論理式
`true`	1
`erase`	$\top$
B, C	$B \otimes C$
B `&` C	$B \,\&\, C$
B; C	$B \oplus C$
B `-<>` C	$B \multimap C$
B `=>` C	$!B \multimap C$
`!`B	$!B$
`forall [`X`]\`B	$\forall x.B$

図 1 LLP での記法と論理式の対応

式は Prolog のプログラム (事実や規則) に対応する論理式である．ただし Prolog のプログラムとは異なり，消費により削除されることがある．シーケント計算として定式化した場合，ゴール論理式はシーケントの右辺のトップレベルに，リソース論理式は左辺のトップレベルに現れる論理式となっている．Prolog と比較したとき，最も大きな違いはゴール論理式としての R`-<>`G と R`=>`G である．これらはいずれもリソース論理式 R を仮定として追加した上で，ゴール G を実行することを意味している．

次に，LLP プログラムを次のように構文定義されるプログラム節 D の列として定める．

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

ここでプログラム節 D 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であり，仮定として何度も利用可能である．すなわち，プログラム節 A は $!\forall\vec{x}.A$ を表し，A`:-`G は $!\forall\vec{x}.(G \multimap A)$ を表す ($\vec{x}$ は A あるいは $G \multimap A$ に現れる全自由変数)．

以上が LLP の基本的な構文であるが，プログラムの記述性を高めるためにいくつかのシンタックスシュガーを用意している．たとえば $R_1 \multimap R_2 \multimap G$ は R_1`-<>`R_2`-<>`G と記述するが，この式は $(R_1 \otimes R_2) \multimap G$ と同値なので (R_1`,`R_2)`-<>`G と記述しても良い．

ゴール論理式として R`-<>`G も R`=>`G も現れないプログラム，すなわちリソース論理式が現れないプログラムの場合，呼び出し可能な述語はプログラム節で定義されたものだけであり，それらは何度でも利用できる．このようなプログラムでは，LLP のゴール論理式 `true` および `erase` は Prolog の `true` と同じ動作になり，同様に G_1`,`G_2 および G_1`&`G_2 は G_1`,`G_2 と，G_1`;`G_2 は G_1`;`G_2 と，`!`G は G と同じ動作になる．したがってゴール論理式として A および G_1`,`G_2 だけを用いた LLP プログラムは，構文的にも操作意味論的にも純 Prolog プログラムと全く同一になる．

4 LLP のプログラミング

LLP プログラムの動作を理解するには，リソースを実行時に追加/消費可能なリソース・プールが存在すると考えるとわかりやすい．リソース・プールに追加されているリソースは，プログラム節と同様にゴール中からの述語呼び出しが可能である．ただし，プログラム節では全変数が全称束縛されているが，リソース論理式は自由変数を含んでも良い点が異なる．

4.1 リソースの追加

リソース R を追加するには，ゴール R`-<>`G あるいは R`=>`G を実行する．

ゴール R`-<>`G の場合，R をちょうど一度だけ利用可能なリソースとしてリソース・プールに追加し，ゴール G の実行に進む．G 中では R をプログラム節と同様に呼び出すことが可能であり，呼び出されるとリソース R はリソース・プールから削除される．これをリソースの **消費** と呼ぶことにする．たとえば以下の質問では，リソース `r(1)` を追加した後，ゴール `r(X)` を実行するが，これにより，リソース `r(1)` が呼び出され (消費され)，`X=1` となって成功する．

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

また，ゴール R`-<>`G で追加されたリソース R は必ず G 中で消費されなければならない．たとえば以下の質問は `r(1)` が消費されないため失敗する．

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

ゴール R`=>`G (すなわち $!R \multimap G$) の場合，R を 0 回以上何度でも利用可能なリソースとしてリソース・プールに追加し，ゴール G の実行に進む．たとえば以下の質問の解は，`X=1` および `X=2` である．

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

リソース R_1`&`R_2 は選択可能なリソースを表す．R_i のどちらか 1 つだけを利用できる．たとえば以下の

質問の解は，X=1 および X=2 である．

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

リソース G-<>R は規則型のリソースを表す．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，X=1 となり同時に 1 が表示される．

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

全称限量子で束縛されたリソース R を R=>G によって追加した場合，Prolog の assert と同様の効果を持つ．たとえば以下の質問は，節 p(X) :- q(X) を assert した後，r を実行することに相当する．

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

ただし，追加したリソースを利用できるのは r の実行中だけであり，バックトラックにより節が消去される点が assert とは異なる．

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) の assert とは異なる．

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

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

4.2 リソースの消費

ゴールの実行はプログラム節の呼び出しあるいは追加されているリソース論理式の呼び出しを意味する．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．

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，G_1 でのリソース消費をキャンセルした後，G_2 を実行する．また，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の解は，X=Y=1, Z=2 および X=Y=2, Z=1 である．つまり，r(X) と r(Y) が共に r(1) を消費するか，共に r(2) を消費するかの場合にのみ成功する．

```
?- 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，=>で追加されたリソースあるいはプログラム節だけが利用可能である．たとえば以下の質問の解は X=1, Y=2 となる．

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

ゴール erase (線形論理の $\top$) 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 は残りのリソースを消費する．たとえば以下の質問は，残り 2 つのリソースが erase により消費され成功する．解は X=1, X=2, および X=3 である．

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

4.3 LLP の応用プログラム

図 2 に有向グラフでの経路探索のプログラム例を示す．各弧は規則型のリソースとして表され，1 度通った弧は消費され 2 度と通ることはできない．したがってこのプログラムにより，頂点 a から頂点 d について，同じ弧を通らない経路を探索することができる．なお，通らなかった弧は erase で消費される．

一方 erase を削除した場合，すなわち最後の行を a -<> d に変更した場合，すべての弧を通るオイラー経路の探索になる．その他，線形論理における各種の論理演算子の特徴を生かした探索の実現例を以下に示す．

- 最後の行を a -<> ((c; d), erase) に変更した場合，頂点 a から頂点 c または頂点 d への経路探索になる．
- 最後の行を a -<> ((c, erase) & (d, erase) に変更した場合，頂点 a から頂点 c への経路および頂点 a から頂点 d への経路の両方が探索される．頂点 a から c への経路探索で消費された弧は，頂点 d への経路探索の前に消費がキャンセルされるため，再び利用可能となる．
- 最後の行を (a & c) -<> (d, erase) に変更し

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

図 2 有向グラフでの経路探索のプログラム例

```
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 3 *N*-クイーンのプログラム例

た場合，頂点 a あるいは頂点 c から頂点 d への経路探索になる．

- 2 行目を ((a -<> b) & (b -<> a)) -<>に変更した場合，a から b への弧と逆向きの弧が選択可能なリソースとなり，どちらか一方の弧しか消費できない．したがって，他の弧についても同様に変更すれば，無向グラフでの経路探索となる．

同様のプログラムを Prolog でリストを用いて記述する場合と比較し，線形論理の豊富な論理演算子を活用した簡潔な記述が実現できている．

次に *N*-クイーンのプログラムを図 3 に示す．queen(8,Q) の実行により，まず結果の配置を取り出すためのリソース result(Q) が追加され，place(8,8) が実行される．place(8,8) では，リソース c(1), . . . , c(8), u(2), . . . , u(16), d(-7), . . . , d(7) が追加され，solve(8,[]) が実行される．これらのリソース c, u, d は，各列，各右上がりのライン，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1 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．

同じ問題を Prolog で記述した場合，リストによって配置を記憶するが，リストは逐次的なデータ構造であり高速なアクセスができない．一方，LLP コンパイラ処理系ではリソース・プールをハッシュ表で実現しており，8-クイーンの全解探索で 3.2 倍以上，13-クイーンの全解探索で 4.7 倍以上の速度向上結果を得ている [4]．

また，論文 [9] では一階述語論理の節形式に対する定理証明系 lolliCoP について報告している．lolliCoP は LLP で記述された数十行程度のプログラムであるにもかかわらず，他の定理証明系と比肩できる性能を示した．他の応用プログラムとしては，一階述語論理の節形式を LLP プログラムに変換し証明探索を行う LLPTTP [16]，命題線形論理の論理式を LLP プログラムに変換し証明探索を行う LL2LLP [17] がある．

5 LLP コンパイラ処理系

LLP コンパイラでは，Prolog コンパイラで広く用いられている WAM (Warren's Abstract Machine) [18] をベースにし，リソース論理式のための命令を追加した抽象機械 LLPAM [4] を用いている．

LLP コンパイラは LLP プログラムを入力とし，それを LLPAM の命令列にコンパイルするプログラムであり Prolog で記述されている．リソース論理式も Prolog のプログラムと同様に抽象機械の命令列にコンパイルされるが，それらの命令列へのポインタ (クロージャ) を LLPAM 内のリソース管理表に登録し，どのリソース論理式が利用可能かを管理している．また，リソース論理式の第一引数をキーとしリソース管理表へのエントリを値とするハッシュ表を導入し，ゴールがどのリソース論理式と単一化できるかを高速に判定できるようにしている．

LLPAM エミュレータは C 言語で記述されている．基本的な組込み述語も C 言語で記述されているが，LLP インタプリタを含めた他の多くの組込み述語は LLP 自体で記述されており，通常の Prolog 処理系として利用可能である．ただし，Prolog プログラムでしばしば用いられる assert, retract 等の組込み述語が用意されておらず，LLP のリソース論理式を用いたプログラムに修正する必要がある．また，現在の実装ではガーベジコレクタが実装されていないため，大規模なプログラムの実行には注意が必要である．ただ，LLPAM におけるリソースの消費は，リソース・プール上で消費可能性を表すレベルの数値を変更する方法で実装されており，利用可能なデータをリストで管理し消費時にリストの再構築を行うのに比較して，

必要となるメモリ使用量は少なくなることが期待できる．たとえば N-クイーンについて，Prolog の場合は空いている列番号のリストを使用し，クイーンが置かれるたびにそのリストを再構築する必要があるが，LLP でリソースを利用した場合，リソースの消費可能性を表すレベルの数値の変更だけで良いため，ほとんどメモリを消費しない．

Prolog コンパイラ処理系としての LLP 処理系の性能は，高性能で知られている SICStus Prolog の 1.5 倍程度遅く，広く利用されている SWI-Prolog よりは 1.7 倍程度速い．ただし，前述の N-クイーンのプログラム例のように，LLP 言語の機能を活用すれば SICStus Prolog よりも速いプログラムを実現できる可能性がある．

6 おわりに

本稿では，直観主義線形論理に基づいた論理型プログラミング言語 LLP とその処理系について紹介した．

残念ながら近年においては，Prolog を代表とする論理型プログラミング言語が用いられる機会は少なくなっているように思われる．しかし，2011 年にクイズ番組で優勝した IBM Watson の自然言語処理プログラムは Prolog で記述されているとのことだ．計算機の性能向上を背景として，自然言語処理や推論といった高度な処理を計算機に行わせたいという要求は今後増大するものと思われる．そのような状況で，論理型プログラミング言語およびその処理系の実装で培われてきた技術が再び注目される可能性を期待して，この稿を閉じる．

参 考 文 献

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

[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., Kang, K.-S., Hirai, T. and Tamura, N.: Logic Programming in a Fragment of Intuitionistic Temporal Linear Logic, in *Proceedings of the 2001 International Conference on Logic Programming* (*ICLP 2001*), LNCS 2237, Springer, 2001, pp. 315–330.

[4] 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(2001), pp. 39–60.

[5] 番原睦則, 田村直之, 井上克巳: Prolog から Java へのトランスレータ処理系とその応用, コンピュータソフトウェア, Vol. 24, No. 3(2007), pp. 75–86.

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

[7] Harland, J., Pym, D. and Winikoff, M.: Programming in Lygon: An Overview, *Algebraic Methodology and Software Technology*, LNCS 1101, Springer, 1996, pp. 391–405.

[8] 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.

[9] Hodas, J. S. and Tamura, N.: LolliCoP – a Linear Logic Implementation of a Lean Connection-method Theorem Prover for First-order Classical Logic, in *Proceedings of the International Joint Conference on Automated Reasoning 2001* (*IJCAR 2001*), LNCS 2083, Springer, 2001, pp. 670–684.

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

[11] 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.

[12] Kobayashi, N. and Yonezawa, A.: ACL — A Concurrent Linear Logic Programming Paradigm, in *Proceedings of the 1993 International Logic Programming Symposium*, 1993, pp. 279–294.

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

[14] Miller, D.: Overview of Linear Logic Programming, *Linear Logic in Computer Science*, London Mathematical Society Lecture Note Series, No. 316, Cambridge University Press, 2004.

[15] 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.

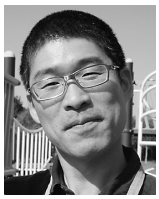
[16] 田村直之, 番原睦則: LLPTTP: 線形論理型言語コンパイラ処理系を用いた定理証明システム, コンピュータソフトウェア, Vol. 20, No. 5(2003), pp. 90–96.

[17] 田村直之, 番原睦則: 線形論理型言語コンパイラ処理系を用いた古典命題線形論理の定理証明システム, コンピュータソフトウェア, Vol. 22, No. 1(2005), pp. 98–103.

[18] Warren, D. H. D.: An Abstract Prolog Instruction Set, Technical Report Technical Note 309, SRI International, Menlo Park, CA, 1983.

田村直之

1980年神戸大学理学部物理学科卒業．1985年同大学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年神戸大学工学部勤務．2003年より神戸大学情報基盤センター教授．論理プログラミング，制約プログラミング，SAT技術等に興味をもつ．日本ソフトウェア科学会，情報処理学会会員．



番原睦則

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．1996年国立奈良工業高等専門学校助手．1998年同校講師．2003年より神戸大学情報基盤センター講師．2007年同校准教授．博士(工学)．論理プログラミング，制約プログラミング，SAT技術等に興味をもつ．日本ソフトウェア科学会，情報処理学会会員．