論理型言語PROLOGを用いた アーキテクチャの記述と論理シミュレーション

内田俊一
（電子技術総合研究所）

樋口哲野
（慶応大学理工学部）

1. はじめに

新しいプログラミング言語として，述語論理をベースとする言語PROLOGが，注目を集めている。

PROLOGは，非手続型言語であり，非決定的処理の記述が行える等記述レベルが高いこと，計算モデルの形式が整っており，LISP等の関数型言語と同様，処理系がコンパクトにでき，マシンアーキテクチャも考えやすい等，興味ある特徴を有している。

そこで，PROLOGについてのこれらの評判を確認すべく，簡単な論理シミュレータを作ってみた。

論理回路は，並列動作し，内部状態を持つなど，記述対象としては，おもしろいテーマである。

2. シミュレータの機能とプログラムの構成

PROLOGの言語的特徴からいえば，論理回路の仕様記述や検証などの用途に，より向いていると思われるが，まずは，ゲートレベルの回路記述とシミュレーションを対象とした。

シミュレーションのモデルとしてはゲートレベルの各素子に標準の遅延時間を与える標準遅延モデルを採用し，このような素子から成る順序回路の動作を追跡し，出力信号のタイムチャートを描くことを目ざした。

しかし，PROLOGの記述レベルの高さを利用すれば，ゲートレベルより上の機能レベルの記述も容易で，双方をミックスしたシミュレーションへの拡張も容易である。

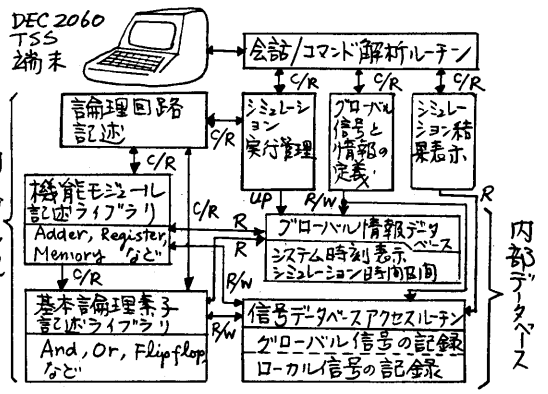
シミュレーションの実行は，基本時間単位 Ts を時刻 T の最小きざみとし，各時刻 T(m) における素子の入出力信号値をすべて求め，後から表示する必要のあるものは，記録するという手続により行われる。

PROLOGのプログラムの実行には，LISPと同様，関数性があり，一つの関数［いくつかの節（clause）で構成される］の実行が成功裏に終了すると実行途中の変数の値などの履歴［history］は，失われる。

これを保存するためには，内部データベース機能を用いればよい。内部データベースは，LISPのproperty listと同様，メモリ内にとられ，記録形式は，プログラムと同じ節の形となる。

書き込みは，asserta(X)，または，assertz(X)で，消去は，retract(X)という組込み関数（evaluable predicate）で行う。読みだしは，通常の節の呼出しと全く同じである。

本シミュレータは，簡単にLSIMと呼ばれるが，LSIMは，図－1のような構成となっており，信号値のほか，システム時刻等のグローバル変数の記



図－1. シミュレーションプログラム（LSIM）の構成

録にも，この内部データベースを用いている。

構成要素は，シミュレーション制御プログラム（プログラム本体），論理回路の記述，および，内部データベースの3つが，主なものである。

3．論理回路の記述

3.1 回路記述の2つのレベル

論理回路の記述は，andやinverter，JK-flipflop等の基本素子のレベルとこれらを組合せたadderやregister，さらに，これらを組合せたものを含む機能モジュールの2段階を設けた。

これらの違いは，基本素子の記述の中には，信号データベースの読み書き，システム時刻の読み取り等の，シミュレーション実行制御に関する記述が含まれるのに対し，機能モジュールの記述の中には，記述対象とする論理回路に関するものだけが含まれ，シミュレータの細部を知らなくとも書けるようになっていることである。

従って，この違いは，実際の素子のレベルとは無関係であるから，adderやregister，さらには，RALUやメモリ等も，基本素子としてもよく，目的に応じ，任意に選択できる。

LSIMにおける回路の記述は，PROLOGの文法に沿って書くことにしており，回路や素子の記述，すなわち，仕様記述そのものが，PROLOGのプログラムとなり，インタプリタやコンパイラで，直接，処理，実行されるものとなる。

基本素子の例として，図-2に2入力andゲートの記述を示す。

この関数（＝節）が呼ばれる時は，引数として，コンテクスト番号（Cn），2つの入力信号名，1つの出力信号名が与えられる。

信号名に対する信号値の読み書きはこの関数の内部で行われ，外からは見えないようになっている。

```
%<<< 2in and gate >>>
%
:-public and_2in/4.                    } コンパイラへの指定
:-mode  and_2in(+,+,+,-).              }
%
and_2in(Cn,A,B,X):-                    { 現在のシステム時刻読み取り
    time_slot(T0),                     <
    read_sig(T0,A,Va),                 } ← 信号値読み取り
    read_sig(T0,B,Vb),                 }
    and_delay_time(Td),                ← 遅延時間読み取り
    Tn is T0+Td,                       ← 出力の決定時間の計算
%
    ( (((Va==x);(Vb==x)),Vc=x);        }
      (Va+Vb<2, Vc=0);                 } ← andの計算
      (Va+Vb)=2, Vc=1) ),              }
    write_sig(Cn,Tn,X,Vc).             ← 信号値の書込み
%
```

図-2 2入力andゲートの記述

flipflopのような内部状態を持つ素子の場合も，ほとんど同様であるが，このような素子では，出力値の算出のために，現在時刻の入力信号値のほか，一時刻前の入力，出力信号値も用いて計算を行う。（LSIMでは，一時刻前の信号値まで保存しており，それ以前は，次々と消していく）

以上のような，基本素子の記述は，システム側でライブラリとして準備しておくものと考えている。

基本素子を組合せて記述する機能モジュールの記述例として，4ビット加算器の記述を，図-3に示す。

このレベルの記述では，基本素子の記述に含まれていないような信号データベースの読み書き等は含まれず，実際の論理回路図と，ほぼ完全に対応する。

従って，容易に記述できるとともに，回路規模が大きくなっても対応でき，さらに，ドキュメントとしても使用可能であろう。

3.2 回路記述と実際のハードウェアの対応

LSIMでは，回路中のすべての素子間，機能モジュール間の配線について信号値を求め，一般には，信号データベースへ記録する。

この時，その信号値の定義される時刻と，ユニークな信号名が必要となる。

```
%<<< 1 bit adder >>>>>>
%
:-public adder_1bit/6.
:-mode   adder_1bit(+,+,+,+,-,-).
%
adder_1bit(Cn,XI,YI,CII,ZI,COI):-
    inverter([Cn|1],XI,NXI),
    inverter([Cn|2],YI,NYI),
    inverter([Cn|3],CII,NCI),
%
    and_3in([Cn|4],XI,YI,CII,OR0),
    and_3in([Cn|5],NXI,NYI,CII,OR11),
    and_3in([Cn|6],NXI,YI,NCI,OR12),
    and_3in([Cn|7],XI,NYI,NCI,OR13),
    and_3in([Cn|8],XI,YI,NCI,OR21),
    and_3in([Cn|9],XI,NYI,CII,OR22),
    and_3in([Cn|10],NXI,YI,CII,OR23),
%
    or_4in([Cn|11],OR0,OR11,OR12,OR13,ZI),
    or_4in([Cn|12],OR0,OR21,OR22,OR23,COI).
%
```

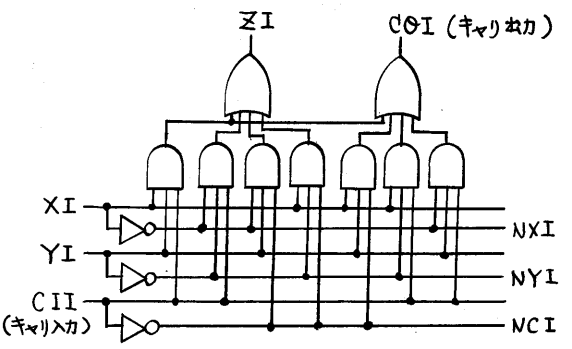


図-3 1ビットadderの記述と対応する論理回路

回路の記述中には，その信号名が，陽に与えられないローカル変数に対応するものが存在し，回路記述の階層が増えると，ほとんどがこのようなローカル変数となる．

また，実際の論理回路のハードウェアにおいては，図-3にもあるように，必要な個数だけの素子が用いられる．しかし，プログラム上では，一つの記述が，くり返し呼ばれて使われるだけである．

LSIMにおける信号値の読み書きは，この最もレベルの低い基本素子で行われるから，実際のハードウェアと対応するユニークな番号づけが必要である．

このために，図-3にも見られるようなコンテクスト番号を用いることとした．この番号は，各モジュールの記述の中で，そのモジュールの呼ばれ時に与えられる番号Cnと，各モジュール内でのユニークな素子の番号1，～nを組合せて，[Cn|3]のようなリスト形式で作る．

記述の階層が深くなると，これは，その階層（関数のネスティング）の数だけの桁を持つ．

このユニークなコンテクスト番号により陽に信号名の与えられない配線の信号の識別が可能となり，信号データベースの読み書きが行われる．

このような無名の信号を，LSIMでは，ローカル信号と呼び，陽に名前が指定される信号をグローバル信号と呼んでいる．

これらは，次のように扱われる．

①グローバル信号：

- トップレベルの回路記述で，陽に信号名が与えられる．
- シミュレーション終了後，そのタイムチャートの表示が求められるため，全時間区間にわたりその値を保存する．
- PROLOGでは，atomとして扱われ，信号名は，小文字で書く．
- データベース中の記録形式

```
clear_reg1 ( 5 , 1 ):-!.
信号名      時刻  信号値
                (1, 0, x)
```

②ローカル信号：

- 信号名が陽に与えられない配線に対応．識別は，コンテクスト番号で行われる．
- シミュレーション終了後に表示の必要がないから，シミュレーションの現在時刻より一時刻前までを保存し，それ以前は消去．
- データベース中の記録形式

```
local_sig_system ( [1,1,3],
   5, 1 ):-!.
統一名   時刻 信号値   コンテクスト番号
                カットシンボル
```

実際の回路記述では，ローカル信号が，ほとんどを占めている．

4. シミュレーションの実行

4.1 並列動作の追跡メカニズム

論理回路の動作のモデル化の考え方としては，各配線を伝わる信号の到着によって，各素子の動作が起動されるというイベント駆動型モデルが，まず，考えられる。

これは，GPSSやSIMULA等のシミュレーション言語でおわれ，コルーチン構造をベースとする便利な機能が準備されている。

PROLOGにも，このような機能の組込みが試みられているが，現在，使用しているエジンバラ大学版のPROLOGには，このような機能は無い。

そこで，LSIMでは，内部データベースを用いた信号値の受け渡しを前提とし，論理回路の記述においては，回路中の信号伝搬順序を考慮して，モジュール内の記述をを行うことで，並列動作をシミュレートすることとした。

すなわち，図－4のような回路を記述する場合を考えると，プログラムの記述において制御のわたる順序①～⑤は，実際の回路において，信号の伝搬順序に反していないようにする。

たとえば，①～⑤の素子の任意の2つ ⓐ，ⓑをとり出した時，aくbならその2点間で，信号が伝わるのは，ⓐ→ⓑの方向となるようにする。

これは，信号値が決定されていく順序でもある。

但し，出口から入口の方へ戻る信号については，この条件に従わなくともよい。

4.2 PROLOG実行メカニズムとの対応

シミュレーションにおける時刻の進め方とPROLOGの実行メカニズムとの対応は，次のようになっている。

PROLOGの実行は，一つの質問形式の節の真偽を証明する手続きとして実行され，その間で作られる実行環境は内部データベースへ記録（assert）されたもの以外は，クリアされる。

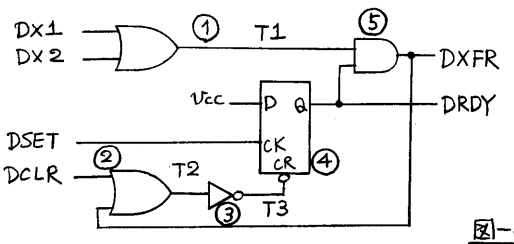
LSIMでは，各時刻ごとに，論理回路を記述したプログラムのトップレベルを呼出し実行することを，一つの証明の単位と考えている。

従って，時刻が更新されるごとに，この証明手続が，くり返し呼ばれ，証明手続き間の信号値の受け渡しは，信号データベースを用いて行っている。

各基本素子の記述の中では，そこに制御がわたってくるたびに，現在時刻における信号値の読出し，出力値を計算する。そして，出力値は，その素子の遅延時間だけ，未来の値として，信号データベースへ書き込む。

遅延時間が長ければ，長い程，その出力値は，現在より先の時刻において決まるものであり，将来使われる値として，信号データベースへ書き込まれる。

この方法によれば，flipflopやregisterのような内部状態を含む素子も，回路の出口から入口の方へ戻るフィードバック信号を含む回路も，andのような素子と，同様に扱え，記述がきわめて簡単なものとできる。




```
qfunc( Cn, DX1, DX2, DSET, DCLR, DXFR, DRDY ):-
    or_2in( [Cn|1], DX1, DX2, T1 ),          ①
    or_2in( [Cn|2], DCLR, DXFR, T2 ),        ②
    inverter( [Cn|3], T2, T3 ),              ③
    dff( [Cn|4], vcc, DSET, DRDY, T3 ),      ④
    and_2in( [Cn|5], T1, DRDY, DXFR ).       ⑤
```

図－4　論理回路要素の記述順序と規則

信号の値は，真（１），偽（０），未確定および未定義（Ｘ）の３値である。信号データベース中に，陽に未定義として記録されていない場合も，未定義（Ｘ）と見なしている。

もちろん，入力信号の値は，全時間区間にわたり，前もって定義（assert）されている必要がある。

5. LSIMの評価とPROLOGの使い勝手

ここでは，LSIM作成における経験をまとめてみる。LSIM自身の評価というよりは，PROLOG処理系の評価となっている。

ここで使用したK処理系は，エジンバラ大学で開発されたPROLOGで，インタプリタとコンパイラの双方がそろっている。コンパイルすると処理速度が，４～５倍程度向上し，プログラムのメモリ占有量も，２～３割小さくできる。[1], [2]

使用した計算機は，DEC社のSystem 2060で，OSは，TOPS-20である。このPROLOG処理系では，ユーザプログラムの使用可能メモリサイズは，約160 KWである。

5.1 記述の容易さ

書きやすさ，見やすさは，すでにいくつかの図で示したとうり，なかなかのものである。FORTRANなどよりは，はるかに良い。

変数のパタンマッチング機能は，プログラムの読み易さの改善に大きく貢献している。LISPと比較すると，カッコの数が少くてすむこと，car, cdrのようなセレクタを書かなくてすむことは，ありがたいことである。

また，LISPと同様，プログラムの部分的実行が容易であり，虫のとれにくいプログラムが作りやすい。デバッガも準備されており，複雑な虫の発見に効果的である。これが無いと，まず，プログラムを作る気がしないと思われる。（マルセイユ大学版PROLOGもあるが，これは使う気がおこらない）

バックトラッキングを用いた非決定的処理は，うまく用いれば，大変おもしろいが，思いがけない所で，バックトラッキングがおこることがある。

特に，予想外のケースが発生して，一種の虫によるバックトラックがおこる場合があり，このため，バックトラックが無意味な個所は，カットシンボル（！）を入れて，バックトラックを禁止しておくよう心がける必要がある。

また，まだ生まれて間もない処理系なので，組込み関数（evaluable predicate）が少かったり，専用のエディタ等が無い等，LISPと比べると見劣りがする。特に，LSIMの場合，処理の規模が，すぐ大きくなるので，メモリの仮想化等をするためのファイル入出力等のサポートが欲しいところである。

このような点は，今後，急速に改善されていくと思われる。

ちなみに，LSIMのプログラムサイズは，プリンタ用紙の枚数でいうと，シミュレータ本体が８枚，MSI, SSIレベルの記述等20種程度で８枚くらいである。

5.2 処理速度について

処理速度は，予想していたよりも，早く感じられた。LSIMでは，信号データベースのサイズが大きくなることから，その探索に時間がかかると考えられた。

しかし，節の呼出しにおいて，そのヘッド（principal functorと第１引数）をkeyとするハッシングが行われることもあり，きわめて迅速に探索される。

特に，データベース中に定義が無い

ような場合の探索と，迅速に行えることは，LSIMでは，きわめて都合が良い。

この結果，LSIMによって，基本素子数50個，時間区間50単位時間くらいのシミュレーションを行うと，約60秒で終了する。

LSIMの回路記述の実行部分では，非決定的処理は，ほとんど無いから，処理時間は，素子数，および，時間区間に，ほぼ比例すると仮定すると，500素子×500時間単位で，6000秒となり，小規模の回路なら実用価値も出そうである。

今後，回路規模を増やし，どのようになるか試みる予定である。

5.3 メモリの使用量について

PROLOGによるプログラムでは，雑にプログラムを書いても，あまり難しい虫は発生しないという利点があるが，その場合のメモリ消費量は，かなり大きくなる傾向がある。

LSIMの場合，各時刻ごとの証明手続きを単位とし，その手続き間のデータの受け渡しは，データベースを介して行っている。

従って，スタック等の実行環境は，1回ごとにガーベージとして，自由領域へ戻して良い。また，信号データベース中のローカル信号値については，不要なものは，次々と消去し，空いた領域は，自由領域へ戻して良い。

しかし，このような操作は，プログラム中で陽に指定するか，そのようにプログラムを作成しないと，処理系はやってくれない。メモリを節約したい時は，それなりの手間をかけねばならない。

PROLOGのインタプリタは，メモリ領域を次のように分けている。

① heap：プログラム本体とデータベースの格納領域。

② global stack，local stack：変数に結合されるatomやデータ構造（スケルトン）に関する情報等の実行環境が入る。

③ 上の②と同様だが，バックトラックの時に回復する変数に関する情報が入る。

これらのメモリ量は，statisticsという組込み関数で，観察できる。

A) スタックの解放

PROLOGの実行では，節の実行が，次々と行われていくに従いスタックが伸びていく。（PROLOGでは，一つの関数は，いくつかの同名のヘッド（functor）を持った節より成る。）一つの節の実行が成功裏に終了しても，その節の後に，まだ同名の節が残っている場合，インタプリタは，バックトラック時に，残りの節を実行できるように，スタック中の情報を保存する。従って，スタックが縮まない。（この辺がLISPと異る。）

LSIMのような非決定的処理をあまり用いないプログラムでは，このようなバックトラックが意味を持たないことが多い。これは，プログラム作成時に容易にわかるから，それを陽に示すことで，スタックが解放される。

これは，カットシンボル（!マーク）で行う。

これと同等の効果は，一つの関数の実行が終了するときは，その最後の節が実行されてから，戻るようにすれば得られる。

PROLOGでは，LISPと同様，プログラムのループは，再帰（recursion）で書くから，上記のような考慮をしないと，簡単にメモリが使い尽されることになる。但し，tail recursionの最適化は，コンパイラがやってくれる。

この工夫は，一にんメカニズムをのみ込めば，容易にできる。

B) ヒープ(heap)の解放

LSIMでは，信号データベース中に信号値を書き込んだり，消去したりしている．

これは，asserta(X)とretract(X)で行っているが，書き込んだ節Xが，プログラム中から参照されていると，retractによって，論理的に消去しても，Xは，ガーベージとはならず，従って，ヒープ中にとられたメモリ領域は解放されない．

この参照が，わかりにくい所で行われているのが，困る理由である．

PROLOGでは，プログラムのどこで発生するかわからないバックトラックの可能性に備えて，実行に成功した節を結んだ経路(And-Or木の探索ルートと考えてもよい．)を保存している．(但し，この経路中には，実行に失敗した節は含まれない．)

通常，asserta(X)やretract(X)を行う節は，この成功した節を結ぶ経路に含まれ，従って，節Xも，この経路から参照されることとなる．(スタック中に，Xを指すポインタがある．)

従って，Xを消去し，さらに，そのメモリ領域を解放するためには，この経路から，assertやretract，さらには，Xを参照する節を除けばよく，これは，それらの節を強制的に失敗(fail)させればよい．(図-5)

```
write_DB(X):- asserta(X).

erase_DB(X):- retract(X).

        ⬇ 上記を，下のように変更する．

write_DB(X):- asserta(X), fail.
write_DB(X).

erase_DB(X):- retract(X), fail.
erase_DB(X).
```

図-5 ヒープを解放するための工夫

以上のような工夫を行うことで，メモリ使用量は，大幅に節約できる．特に，スタックの解放をきめ細かくやると効果は大で，その方法も提案されている．[3]

6. おわりに

PROLOGの使い勝手を調べることに重点を置いて論理シミュレータLSIMを作ってみた．論理シミュレータは，プログラムサイズも，実行時間も，きわめて大きくなり，処理系には，かなりきびしいテストプログラムである．

現在までの感想としては，その記述レベルの高さが，論理回路の仕様記述として使えることが，きわめて魅力的であること，また，実行速度も，今後の改善の可能性を考えれば，実用に供するプログラムも実現可能と思われることなど，かなり良い印象を受けた．

また，未検討だが，論理回路の検証や，VLSIにおける回路要素の配置問題など，非決定的処理が入るものでは，PROLOGの判定が，さらに生かせるものと予想される．[4]

また，処理系自身が，LISPと同様かなりコンパクトであるから，専用のPROLOGマシンを作れば，その処理速度が，かなり向上すると思われる．

しかし，現在の処理系は，まだ，未成熟なこともあり，現在のLISPの処理系が持つような強力な組み込み述語群は，作られていない．

LSIMに関していえば，ディスク等のファイルを仮想メモリとして扱えるようにするための組み込み述語や，端末に描画するための組み込み述語群の欲しいところである．

しかし，これらは，ひんぱんに使われるようでなければ，系統的に整備されていくものと思われる．

LSIMは，まだ，toy programの域を出ないが，今後，基本素子のライブ

ラリを充実するとともに，記述回路の高レベル化を試みる予定である．参考までに，少し複雑な回路のシミュレーション実行例を図－8に示す．

最後に，本研究に有益な御助言をいただいた古川康一主任研究官をはじめとする情報システム研究室の諸氏に深謝する．

参考文献

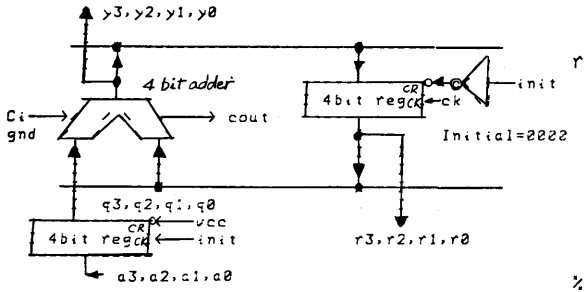
1) L. M. Pereira, F. C. N. Pereira and D. H. D. Warren: User's Guide to DECsystem-10 PROLOG, Sept. 1978

2) D. H. D. Warren: Implementing PROLOG - compiling predicate logic programs, vol. 1 および vol. 2, DAI Research Report NO. 39 および 40, Univ. of Edinburgh, May. 1977.

3) R. A. Kowalski: Prolog as a logic programming language, Proc. AICA Congress, Sept. 1981

4) 古川：PROLOGによる問題解決，第23回情処全大，NO. 5G-2, pp. 859-860, 昭和56年10月．

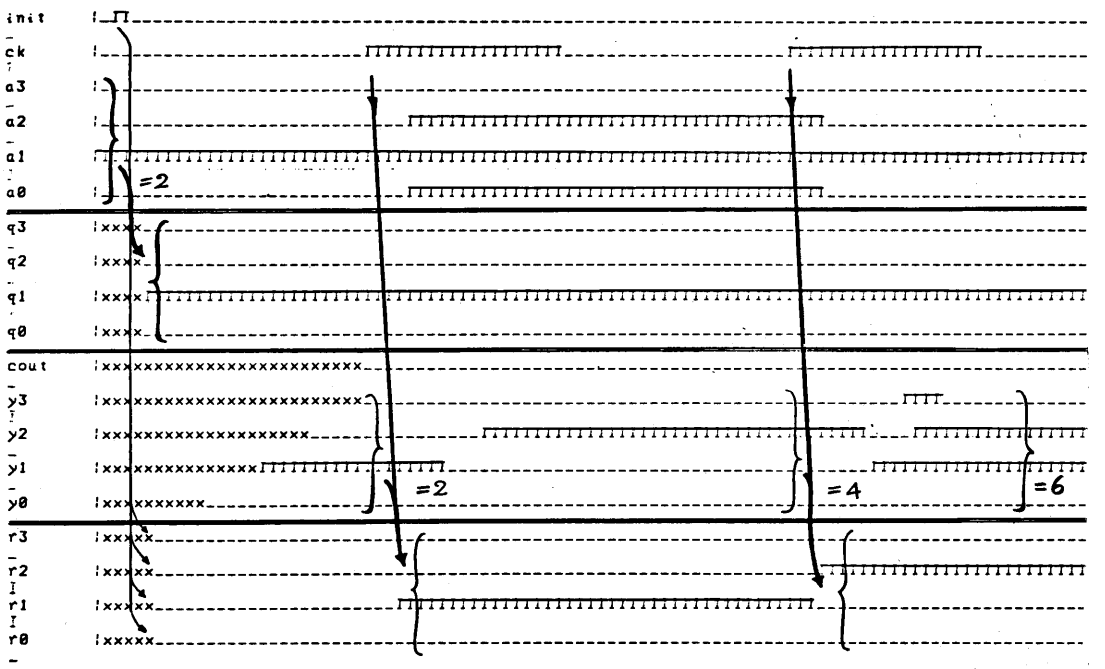
5) 内田，植口：PROLOGと回路シミュレーション，第23回情処全大，NO. 5G-4, pp. 863-864, 昭和56年10月．



(a) 回路構成

```
ra:-
   inverter(1,init,ninit),
   register_4bit(2,a3,a2,a1,a0,
                 q3,q2,q1,q0,
                 init,vcc),
   register_4bit(3,y3,y2,y1,y0,
                 r3,r2,r1,r0,
                 ck,ninit),
   adder_4bit(4,q3,q2,q1,q0,r3,r2,r1,r0,
                 gnd,y3,y2,y1,y0,cout).
%
```

(b) 回路の記述



図－8 4ビットadderとregisterを組合せた回路のシミュレーション ↑(c) タイムチャート

〈資料 No. CA-44 の 5 の追補〉 情報処理学会 計算機アーキテクチャ研究会 昭和57年2月24日

論理型言語PROLOGを用いた アーキテクチャの記述と論理シミュレーション

内田俊一 (電子技術総合研究所)
樋口哲野 (慶応大学理工学部)

上記資料の第5章 LSIMの評価と使い勝手の部分について，一部の説明の訂正と補足を行う．

1. PROLOGにおけるメモリの有効利用のためのプログラミング手法

PROLOGのプログラムにおいて，処理の履歴を保存したい時や，グローバル変数を用いたい時は，内部データベースが用いられる。

内部データベースへの書き込みと消去は，上記の論文中の本論で述べたように，通常 asserta(X), retract(X) で行う。

しかし，これらの操作により，assertされ，新たにHeap中にとられた節Xは，単にretractしたのみでは，論理的に消去されるのみで，Heap中にとられたメモリ領域は，ガーベージとして，自由領域に戻されない。

これは，インタプリタが，バックトラックに備えて，実行に成功した節を結ぶ経路を保存しており，この経路中から，assertされた節が参照されていることによる。

従って，この経路中から，assert も retract，さらに，assertされた節Xを参照するTermを含む節を取りはずせばよく，これは，図-6の行番号4～5や6～7のように，強制的に，失敗(fail)させることで，行える。

しかし，assertされた節Xを参照する時は，読んだ値をとりまさねばならず，その値の保存のために，再び，assertする必要がある。

しかしながら，データベースの内容を書き変えつつ，処理をすすめるような場合や，多くのデータを処理して，少数の答を得るような場合では，きわめて有効である。

このような場合には，図-6の行番号1～3に示すような，ある関数Fと成功した節を結ぶ経路から，取りはずす専用関数を利用するとよい．(本論中の文献3を参照のこと)

この関数 do_recycle(F, A) は，関数Fを実行後，その各Aを，一たんデータベースへ temp(A) の形で，assertしている．それは，使用後，retractされるが，この分の Heap

```
(行番号)
1   % Test of Internal DB R/W
2   % 820223  S.Uchida
3   %
4   do_recycle(F,A):-do_fail(F,A).
5   do_recycle(F,A):-temp(A),
                     retract(temp(A)),!.
6   %
7   do_fail(F,A):-F,asserta(temp(A)),
                     !,fail.
8   %
9   w_db(X):-asserta(X),fail.
10  w_db(X):-!.
11  %
12  e_db(X):-retract(X),fail.
13  e_db(X):-!.
14  %
15  wt(N):-N<0,!.
16  wt(N):-D is N*N,w_db(data(N,D)),
           Nn is N-1,wt(Nn).
17  %
18  et(N):-N<0,!.
19  et(N):-e_db(data(N,_)),
           Nn is N-1,et(Nn).
20  %
21  rt(N,T,T):-N<0,!.
22  rt(N,T,X):-data(N,D),Nn is N-1,
               Tn is T+D,rt(Nn,Tn,X).
23  %
24  tr(N,X):-statistics,
             wt(N),
             statistics,
             do_recycle(rt(N,0,X),X),
             et(N),
             statistics.
25  %
```

% 図-6. データベースアクセスの例

は，解放されない．　また，データベースの中を書きかえつつ処理がすすむLSIMのような場合は，このAは，不要であり，より簡単となる．

図－6の行番号14の関数trは，1～Nまでの値Nと，その2乗Dをペアにして，aa(N, D)の形で，assertしている．　その後，各値を読み出し，総和 $\sum_{i=1}^{N} i*i$ を計算し，assertしたN個の節を消去している．statisticsは，Heapやstack量を表示する組込み関数である．

この例では，assertやretractにより，小まめに，上記の手法を適用しているが，trのようなトップレベルの関数を直接 do_recycle(tr(10,X)) のように呼ぶしても，Heapは回収される．　しかし，小まめにやる方が，よりメモリ利用率はよくなる．

2. LSIMにおけるHeap回収の効果

上記のような手法をLSIMに適用しHeapの消費量をしらべた．

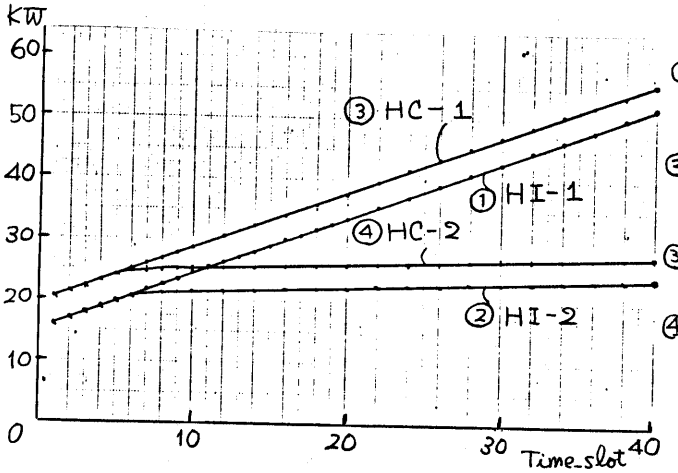
LSIMでは，シミュレーションの各時刻ごとに，Heapの回収が可能で，ここに，do_recycleのような関数を適用している．　この効果は，きわめて大きく，図－7に示すように，メモリ消費量は，ほぼ横ばいとなる．

また，メモリに余裕があることからガーベージコレクションの回数も減り実行時間も，早くなる．（図－7のTeで示す．単位は，秒である．）

コンパイルした場合も示したが，エンベう版では，そのままコンパイルしたのでは，Heapは回収されなくなる．　このため，do_recycleの関数（3行のプログラム）のみ，コンパイルせず，インタプリタで直接実行するようにした．

コンパイルすると，プログラム本体のサイズは，少し増えるが，必要なスタックのサイズが，大幅に減少する．実行時間は，図－7のTeからもわかるように，データベースのアクセスが多くの時間を要するためか，コンパイルしても，それ程早くはならない．（本論で述べた，コンパイルにより，4～5倍早くなるというのは，LSIMの場合は，あてはまらない．）

以上のような工夫をすることで，LSIMは，実用的価値があることが確認されるとともに，PROLOG処理系自体も，実用的システムとして，有用であることが，確認された．



① HI-1：インタプリタによる直接実行．Heapの回収なし．Stack＝35KW, Te＝110s

② HI-2：インタプリタによる直接実行．Heapの回収あり．stack＝16KW, Te＝92s

③ HC-1：コンパイルし，Heap回収無し．Stack＝0.6KW, Te＝62s

④ HC-2：コンパイルし，Heap回収あり．Stack＝2KW, Te＝62s

図－7．LSIMにおけるHeapの使用量