推論機械ＳＩＭ－Ｐの構成と言語

新世代コンピュータ技術開発機構

近山　隆　横田　実　服部　隆　高木茂行
Takashi Chikayama　Minoru Yokota　Takashi Hattori　Shigeyuki Takagi

1．　はじめに

新世代コンピュータ技術開発機構では，推論用専用計算機開発の第一段階として逐次型推論マシン SIM-Pを開発中である。 SIM-Pは昭和59年度中に完成し，以後これをワークベンチとして知識表現・推論システムの研究に役立てることになっている。以下,SIM-Pにおける言語とアーキテクチャの設計方針及び現段階までの進行状況について述べる。

2．　アーキテクチャ

SIM-Pは知識表現・推論システムへの適用を目的としており，ベースとなる言語をProlog風の論理型言語に設定している。基本的なハードウェアは従来の計算機と大きくは異ならないが，Prologの実行を効率良く行なうため,SIM-Pは記憶管理支援機構，データ・タグなどをハードウェアで，ユニフィケーション，ガーベージ・コレクションなどをファームウェアでサポートしている。また，１つの領域の論理空間は16MW（ハードウェア実装は 2MW程度）と大量のデータを主記憶に置くことを考慮している。１ワードのデータ幅は32ビットであるが，そのほかにタグを８ビット持つ。Prologインタプリタの基本部分はファームウェアで記述する。

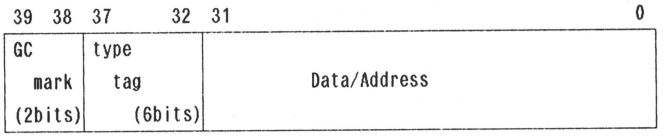

図 1.1　データ語の型式

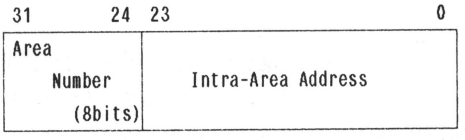

図 1.2　論理アドレス

2.1 記憶管理支援機構

プロセスの実行にはグローバル，ローカル，トレール，コントロールの４個のスタック及びヒープの計５種類の領域を使用する。この各々がそれぞれ16MWまでの論理空間を占めることができる。実装されるメモリはそれほど多くない上，複数のプロセスが同時にメモリ上に存在しうるので，論理空間を実記憶にマッピングする必要が生じる。これをサポートするためにハードウェアにメモリ・マッピングの機構を設けた。従来型の計算機におけるアドレス変換のための連想処理を省略するため，マップ用としてデータ用のメモリとは別に専用の高速メモリを用意する。各スタックやヒープはすべてこのマップを経由してアクセスされる。マップの１エントリはデータ・メモリの１ページ（4KW)を指す。（図 2）

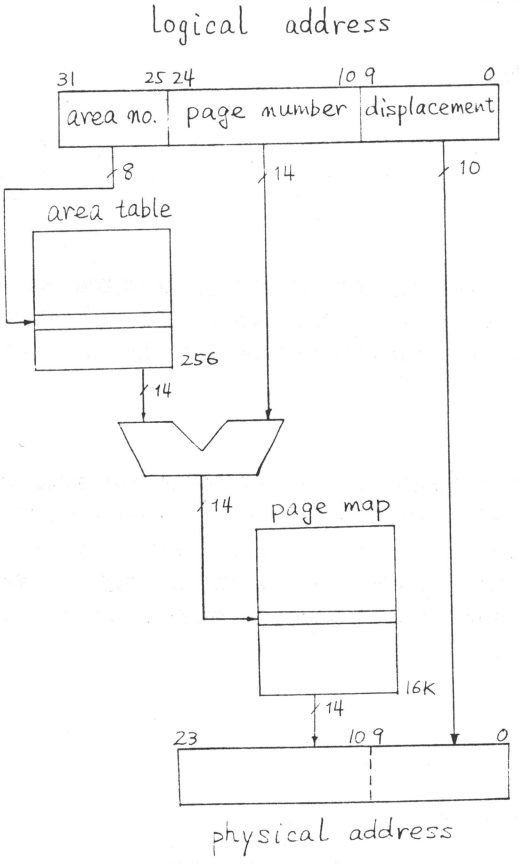

図 2　アドレス変換機構

3．　言語の外部仕様

SIM-PではProlog風の論理型言語をベースとしており，OSや入出力を含むすべてのシステムを拡張版Prologで記述する。そのため従来のPrologにはない制御構造，バックトラックに係わる推論機構，副作用のある構造（データへのアクセス）などを追加している。そのほかにOSの記述に必要なプロセスや空間を扱う述語も持っている。

3.1　機械語

SIM-Pの機械語はPrologの述語の型式をしており，従来の計算機の機械語のような命令は存在しない。しかし，機能上はそれを包含したものとなっている。

組込述語は引数の入出力関係が規定されている。すなわち，入力引数は既定義である必要があり，一般のユニフィケーションが行なわれるのは出力変数に対してのみである。また，組込述語には successの仕方が一通りしかなく，バックトラックの対象とならない点でもユーザ定義の述語とは異なる。

組込述語は記憶管理システムの上で動作するのが原則であるが，あらかじめ確保した限られた記憶領域内でもある程度の計算は可能である。このことを利用し，記憶管理システム以下の機械の核となる部分や，ガーベージ・コレクション中の入出力のバッファリングなどの処理の一部を機械語で記述し，マイクロコードの負担を減らすことを考えている。

3.2 推論機構

推論に係わる基本機能として，ユニフィケーションとバックトラックをファームウェアのレベルでサポートしている。

3.2.1 ユニフィケーション

構造データの表現には,structure sharing法を採用した。 Structure sharing法は copying法に比べ，大きな構造体データを扱う場合，使用するメモリ量が少なく，従ってメモリ・アクセスが少ないという利点がある。

Structure sharing法を採用すると，ユニフィケーションやガーベージ・コレクションの算法がやや複雑になるが，メモリ・アクセスが全体の性能に最も影響しそうであるとの判断に基き，あえて structure sharing法を採用することにしたわけである。

3.2.2 bind hook

また，ユニフィケーションの時点でデバッグなどの操作が可能なように bind hookの機能を持たせている。これは，ある特定の変数に値がbindされたら，その時点で特定の述語（述語列）を実行するという機能である。ある変数の binding ができた時点でトップ・レベルからの変数の値や predicateの定義の参照・変更を可能にし，デバッグを容易に行なうことができる。また，これを利用して他のプロセスを起動したりすることも可能である。

デバッグ中に，ある変数の値がいつどの述語の呼出しによって設定されたかを知りたい場合は多い。bind hook のハンドラとしてデバッガへの抜け出しを指定しておけば，容易にこの状況は把握できる。特にコンパイル済みの部分など，細かいトレースが困難な部分で値が決まる場合に有効であると考えられる。

さらに，設定されるべき値にある制限条件がある場合に，この機能を用いることによって余分な処理を行なう前に値のチェックを行なうことができ，処理効率の向上に貢献するものと考えられる。

3.2.3 バックトラック

バックトラックは従来のPrologと同様に行なう。拡張機能としてバックトラックの際に行なうべき処理を指定するon-backtrackを設けた。これは次の２点が主な目的である。

a) 副作用を持つ操作を行なった後のバックトラックの際に，その副作用による効果を回復しておく。

b) デバッグの際にバックトラックの状況をユーザ・プログラムから報告する。

on-backtrackの効果は次の例のような場合に有効である。

例 1　現在どのフェーズを実行しているかをassertしておき，それを下請けルーチンが参照してメッセージを出す。

```
:-assert(phase(1)),phase1,retract(phase(1)),
  assert(phase(2)),phase2,retract(phase(2)).
```

これでは，phase2が途中でfailして，phase1にバックトラックすると正しくない。そこで

```
change-phase:-retract(phase(1)),assert(phase(2)).
change-phase:-retract(phase(2)),assert(phase(1)),fail.

:-assert(phase(1)),phase1,change-phase,phase2,retract(phase(2)).
```

とすると，現在のPrologならうまく動く。しかし，phase2からphase1へのjumpが生じるとこれでもうまく動かない。

```
:-assert(phase(1)),phase1,retract(phase(1)),
  on-backtrack((retract(phase(2)),assert(phase(1))),
  assert(phase(2)),phase2,retract(phase(2)).
```

としてやればうまく動く。

例 2　述語 pに入った時，出る時にそれを報告する。

```
p:-write('entering p'),nl,  body-of-p,  write('succeeding p'),nl.
p:-write('failing p'),nl,fail.
```

この定義では，　q:-p,!,r.　という呼出しに対しては，failのメッセージが出ないので，

```
p:-write('entering p'),nl,
    on-backtrack((write('failing p'),nl)),
    body-of-p,
    write('succeeding p'),nl.
```

のようにすればうまくいく。

on-backtrack(X) の実現には,Xを適当なマークと共にトレール・スタックにプッシュしてやれば良い。これは cutによって消失することはないので，バックトラックの際に変数の値を回復するのとあわせてon-backtrack処理ルーチンの呼出しを行なえば良い。

3.3 非局所的制御構造

3.3.1 cut

現在のPrologには実行を制御するための機構として cutがある。この cutは最も最近の自分より上の or-nodeの選択枝を刈るものであり，現在実行中のclause内の；で書かれた選択枝は cutしない。そのためにわざわざ短いclauseを作らなければ目的が達成できない場合がある。そこでこの様な問題を回避できるように，最も最近の（自分自身のclauseを含めての） or-nodeを cutする「近いcut 」と， cutすべき位置をあらかじめマークしておき，そのマークまでを cutする「遠いcut 」を設けることにした。（図 3）従来のPrologにおける cutは，暗黙の内にマークを指定した cutとして実現できる。これによって，より柔軟に選択枝をプログラムから cutできる。また，「ある変数の値が決まった時点まで cutする」ということができれば，バックトラックを効率化することができる。「intelligent backtracking」のためにはこの機能は有効であろう。

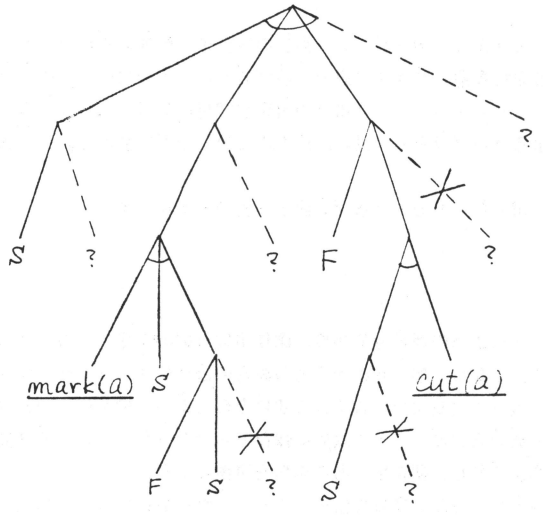

図 3 遠いcut

3.3.2 catch-throw

Lispでは大域的な実行制御のために catchと throwという関数が用意されており，関数から別の関数の中までjumpすることができる。これは大域的なgotoにあたる。Prologの実行の際にはfailの情報を単なるバックトラックによって返すよりも，上位のclauseに直接failを返すことができれば処理が単純になる場合や，ある条件が成立した時点で以下の処理をすべて省略して successして良い場合などがある。特に非同期の外部要因によって処理を変更したい場合には明示的に処理の流れを変える必要がある。

この様な処理を用意に記述するために，あらかじめマークを設定しておき，マークを指定してその場所に successを返すことができるようにした。なお，failを throwしたい場合は「遠い cut」とfailの組合せで実現できる。

3.4 例外処理

プログラム実行中の例外的事象（オーバーフロー，添字の範囲オーバーなど）に対して，例外処理ルーチンを登録することができる。例外処理ルーチンからは，例外を生じた述語の success/failureによる終了や再試行を指定できる。また，例外処理ルーチンとしてデバッガを指定すれば，例外を生じた環境を調べ，例外の原因を究明することも可能である。

3.5 構造のあるデータと副作用

従来のPrologで扱うことのできた数値やリストの他に，構造型データとしての配列，基本データとしての文字列・ビット列を扱う機能を設けた。OSのようにテーブルのデータを扱うことが必要な場合，従来のユニフィケーションとバックトラックによる値の設定では不十分であり，配列の要素を書きかえるというようなロジック・プログラムの外側の機能が有用である。また，入出力ではビット列や文字列を直接扱うことが必要である。そこで，配列やビット列を機械語のレベルで扱うことにした。インタプリタはこれらの命令によってプログラムの上で書かれたビット列や配列を処理する。

3.5.1 配列・文字列・ビット列

配列はヒープ領域に取られたデータ・エリアとその構造（先頭アドレス，添字の最小値，配列の長さなど）を保持するヘッダ部分から成る。機械命令では領域の確保，要素の参照，要素への値の（破壊的）代入，配列の一部を別の配列として再定義(sub-array) などの機能をサポートする。これによってデータ構造の設定が容易になり，プログラムの記述性が向上する。また，再帰呼出しではなく，ループによるプログラムが自然に記述できるので処理の効率も向上する。機械語のサポートする配列は一次元のみである。

文字列・ビット列も要素が一般のデータではなく限られたものである点を除き，配列と同様である。

3.5.2 副作用

Prologでは変数への値の設定はユニフィケーションによるのが普通であり，副作用はProlog本来のメカニズムとは別の，meta-logicalあるいはextra-logical なものである。しかし，配列要素への値の設定のようにユニフィケーションだけでなく破壊的代入が必要なものや，バックトラックによって値が回復されてはまずいものもある。例えばカウンタのようなものを実現しようとすると，従来のPrologではその値をassertしておく以外に保存できなかった。 SIM-Pでは変数に対して数とシンボルなどの非構造体の値をを代入できるようにし，副作用のある処理を可能にした。

構造体データを代入するような副作用を導入するとローカルな変数の値をグローバルな変数に代入した結果，グローバル変数に設定された値が参照される時にその実体は既にスタックから消滅しているということが起こりうる。これは代入される値をヒープ領域にコピーしてから代入することで解決できる。

副作用はロジック・プログラミングからは逸脱するものであるが，実行効率・記述性両面を向上させる上で非常に有用と考える。

4．　言語処理系

SIM-Pの言語処理系はユーザ・レベルでアプリケーションなどの記述に使用されるもの，ユーザ言語から機械語になる途中のインタプリタによって処理される中間語，及び SIM-Pのファームウェアによって直接実行される機械語レベルのものがあり，それぞれ目的に応じて使い分ける。

4.1　記憶管理とガーベージ・コレクション

アーキテクチャの項で述べたように，Prologのプログラムの実行には，４本のスタックとヒープの計５種の領域を用いる。これを管理するためにハードウェアではマップの機構を持っている。プログラムの実行の際には，まずOSがメモリを割当ててプログラムに制御を移す。実行の途中で必要になるたびにページ単位でメモリが割りつけられ，各領域はそれぞれ最大16MWまで伸びることができる。その内，ヒープ領域とグローバル・スタックはガーベージ・コレクション(GC)が必要である。ページ単位でGCできる場合には単なるマップの書替えで対処する。これによってGCのオーバーヘッドを大幅に削減できる。マップの処理だけではすまない場合は，Lispの処理系と同様のGCを行なう。GCにはMorrisのSliding GCを採用した。この方法はやや複雑であるが，余分の領域を必要とせず，実行時間もGCされる領域に比例するのでオーバーヘッドが少ない。

GCが起こると実行中のすべてのプロセスを停止しGCが終了するまで待たせる。これは SIM-P仮想記憶方式を採用しないためGC時間が比較的短いこと，パーソナルなワークベンチとして利用され，人間以上に高速な機器とのリアルタイム入出力が不要なことによる。

4.2　開発手順

SIM-Pの開発はハードウェアとソフトウェアの開発が並行して行なわれており，実機でソフトウェアの開発ができない。そのため，まずDEC-2060上にインタプリタを開発し，次にこのインタプリタを用いて，クロス・ソフトを作り，最終的にレジデント・システムとするブート・ストラップの手法を用いている。中間語のインタプリタは既にDEC-2060上で利用可能になっており，現在アプリケーションの作成に用いるユーザ・レベルのコンパイラを開発中である。最終的にはOS・コンパイラを始めとするすべての SIM-Pのソフトウェアをレジデント・システムで稼働させる予定である。

5．　おわりに

SIM-Pが実用に供されれば人工知能の分野に大きく貢献するものと期待している。ハード・ソフトともに現在開発中であり，皆様の御意見，御指導がいただければ幸いである。

謝辞　　本報告をまとめるにあたり，ICOTメンバの諸氏，ICOTのワーキング・グループの皆様に貴重な御意見をいただいた。ここに謝意を表する。

参考文献

1: F.L.Morris: A time- and space-efficient Garbage Compaction Algorithm
Comm. ACM, vol.21, no.8 (Aug. 1978), pp.662-665

2: W.F.Clocksin,C.S.Mellish: Programming in Prolog. Springer-verlag (1981) p.279

3: D.H.D.Warren: Implementing Prolog - compiling predicate logic programs
D.A.I.Research Report no.39-40 (May 1977)

4: L.M.Pereira,F.C.N.Pereira,D.H.D.Warren: User's GUIDE to DECsystem-10 PROLOG
Edinburgh univ. (Sep. 1978)

5: T.F.Knight Jr.,D.A.Moon,J.Holloway,G.L.Steele Jr.: CDAR. MIT AI Memo 527 (Mar. 1981)

6: 横田　実，梅村　護: 拡張PROLOG(ShapeUp) の実現について
情報処理学会　記号処理研究会　資料 20-3 (1982-10-18)

7: 梅村　護，横田　実: PROLOGマシン・アーキテクチャの一考察
情報処理学会　計算機アーキテクチャ研究会　資料 46-4 (1982-9-17)

8: D.Weinreb,D.Moon: Lisp Machine Manual. Symboloics Inc. (Jul. 1981)