ICOT Technical Memorandum: TM-0090

TM-0090

情報処理学会第30回全国大会発表論文集

ICOTならびに再委託先メーカ

January, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191-5
Telex ICOT J32964

Institute for New Generation Computer Technology

データフロー方式並列推論マシンにおけるOR/AND並列効果のシミュレーションによる測定

久野　英治　　伊藤　徳義
(沖電気)　　(ICOT)

1．はじめに　論理型言語Prologを並列で高速に実行するコンピュータの研究やPrologを基礎とした並列型言語の開発がさかんに行われている。著者等はデータフロー方式に基づく並列推論マシン研究の一環としてこのマシン[1]のソフトシミュレータを作成し，論理型言語Prologに内在する並列性の一つの側面であるOR並列に注目した言語（以下OR並列型Prolog）に対するマシンの方式を検討してきた。それによれば並列度が高ければ実行速度の台数効果が上がるということが確かめられている[4]。また論理型言語に対する並列性を他の視点からとらえたものとしてAND並列に基づくProlog（以下AND並列型Prolog）の研究が挙げられる。その代表的なものがPARLOGやConcurrent Prolog(以下CP)である。ICOTが開発を進めている核言語の並列版KL1はこのAND並列型をベースとする方向で検討されている[5]。

我々はAND並列型Prologの実行方式（CPをベース）の検討を行い，上記シミュレータにその機能を付加した。本稿ではシミュレータの台数規模を拡張し，両並列型言語で記述したサンプルプログラムによる方式の評価を行なったので報告する。

2．並列型Prologの実現　OR並列型Prologが，主として全解探索用に使用されるのに対し，AND並列型Prologはオブジェクト指向プログラミングとの整合性を担ったものとして位置づけられる。両言語に共通する統一化機能についてはunify, consistency_check, substitute等のプリミティブにより実現し，引数間の並列処理を行う。OR並列型Prologにおいては得られた複数の解をnon-strictなデータ構造のストリームとしてマージするプリミティブを用意することによって，解を生成するプロセスと消費するプロセスとの間の同期手段が提供される。そこでは生成された解から順にストリームに送られ，生成順序については全く考慮されない(don't know non-determinism)。これを実現するのがcreate_streamとappend_streamプリミティブである(図-1)。

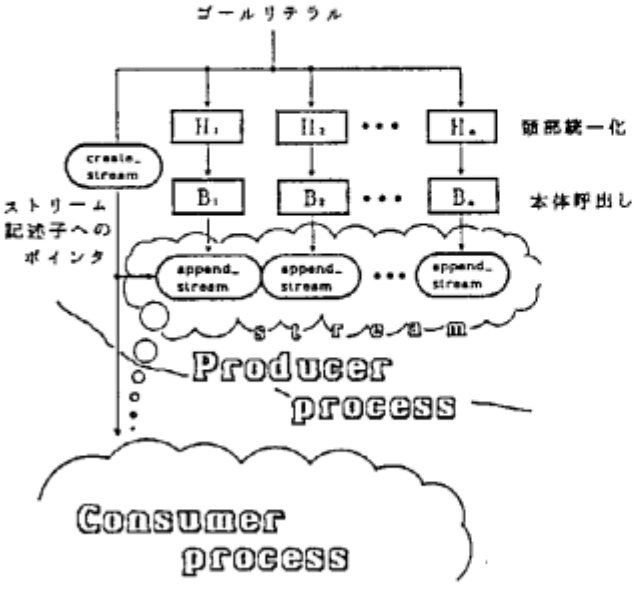


図-1.　ストリームと制御プリミティブ

他方AND並列型Prologにおけるコミットオペレータには二つの機能がある。一つは節間の排他制御を行なう機能であり，図-2のtest&setプリミティブにより実現する。もう一つは節の統一化で得られた変数情報を親ゴールに公開する機能であり，同図bindプリミティブにより実現する。後者の機能は変数の入出力方向を規定する読みだし専用変数とを併せて用いることによりプロセス間の双方向通信手段を提供している[2]。シミュレーションは両言語プログラムを上記のプリミティブ群を用いたデータフローグラフにコンパイルし，シミュレータによりこのグラフをインタプリトすることにより行われる。

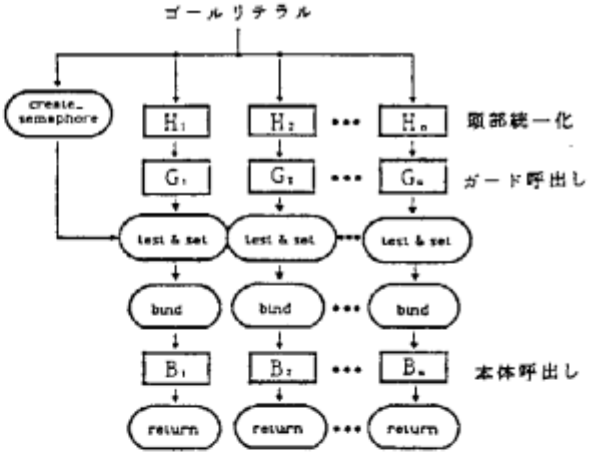


図-2.　コミットオペレータ制御プリミティブ

3．評価と考察　OR並列型評価プログラムとして，トップダウン(以下DCG)とボトムアップ(以下BU

P)の二種の解法による自然言語解析，7-queens，及び整数256個のクイックソートプログラムを用意した。またAND並列型プログラムとして比較対象となる要素がリストであるリストのクイックソート(qsortlist)，同じく要素がリストであるリストの重複をなくす圧縮(compactlist)，簡単なエディタ(editor)，及び，これらを結合したもの(editorsort)等を用意した。

シミュレータに与えるマシンパラメータは文献[1]の設定値とし，プログラムの推論性能はマシンサイクルを250nsとして算出した。LIPS(Logical Inference Per Second)値は単位時間当りの頭部統一化の回数を示しており，統一化に失敗したものも含めて算出する。

図-3はモジュール（処理要素及び構造データを格納するための構造メモリ）台数が変化したときのOR並列型プログラムの台数効果を示したものである。

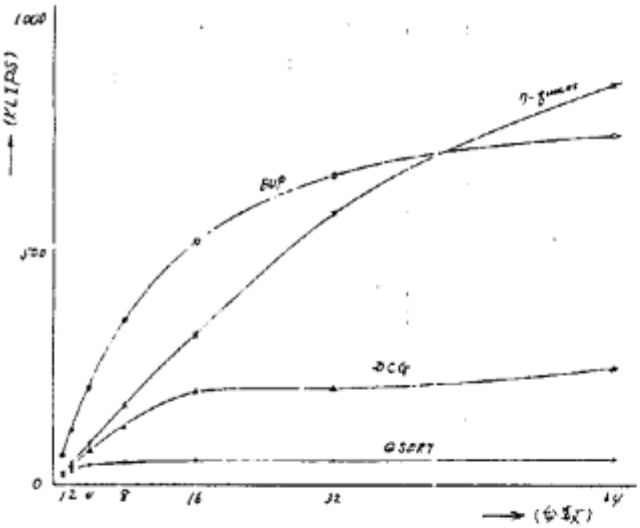


図-3. OR並列型Prolog実行時の台数効果

BUPとDCGは同じ文章を解析した結果であるがLIPS値が異なるのは探索空間の違い（BUPの方が冗長な探索木を設けていることによる）が現われていることを示すものである。プログラムの並列度が高ければ台数効果が期待できることは知られているがBUP，DCGやクイックソートが１６台或いは８台でほぼ飽和するのに対して7-queensではまだ飽和点にいたっておらず更に台数効果が期待できることで立証された。

図-4はAND並列型プログラムのモジュール台数効果を示している。クイックソートにおいてはデータ

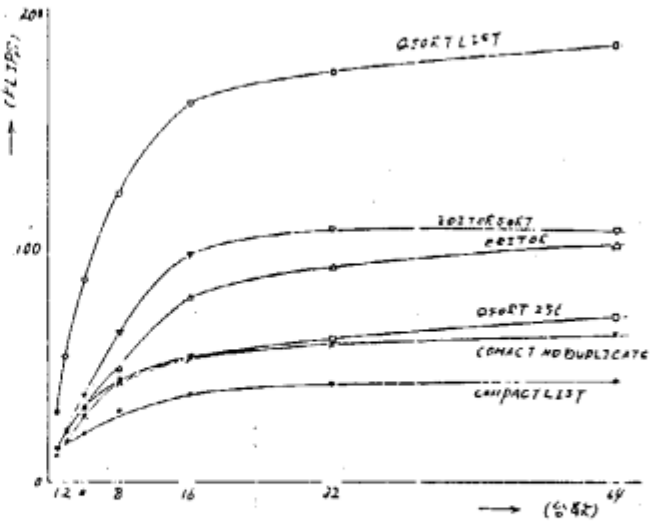


図-4. AND並列型Prolog実行時の台数効果

構造により並べる要素がアトム(qsort256)か構造データ(qsortlist)かで測定値が大きく変化する。リストの圧縮においてはリストデータが重複有り(compactlist)か重複なし(compact-noduplicate)かで測定値に影響を及ぼし，探索空間の違いが現われている。エディタ(editor,editorsort)においても同じことが言える。即ち，並列に動作する個々の処理負荷が重くなるほど台数効果が得られる。

4．おわりに

データフローモデル上に二種の並列型Prolog実行方式を検討しシミュレータ上のインプリメントに成功し，有効に動作することを確認した。現在本モデルに基づいた実験機を試作中であり，今後はこの実行方式を実験機上に実現しシステムの評価を進めるとともにアプリケーションソフトウェア開発をすすめていく予定である。

謝辞．ICOT2研の宮崎氏に感謝する。

〔参考文献〕

(1) 伊藤．「並列型Prologのデータフローマシンによる実行方式」
日経エレクトロニクス，1984年11月5日号。
(2) 伊藤 他．「データフロー方式並列推論マシンにおけるAND並列型Prologの実行方式」
情処30全，6C-1，1985.
(3) 伊藤 他．「データフロー方式推論マシンのアーキテクチャ」
Proc. of Logic Programming Conf. ICOT, Mar. 1984
(4)「電子計算機基礎技術開発成果報告書 基礎ソフトウェアシステム編」
ICOT, Mar. 1984

データフロー方式並列推論マシン実験機の構成

6C-2

来住晶介　久野英治　六沢一昭　伊藤徳義
（　神　電　気　）（ＩＣＯＴ）

1．はじめに　PIM-Dは，第５世代プロジェクトの一環として研究を行っている並列推論マシン実験機であり，データ駆動方式であることを特徴とする。

並列推論マシン向きの論理型言語はOR並列型PrologとAND並列型Prologに大別できるが，ハードウェアへのインプリメントという立場からみると，これらはいずれも，データの実体は共有メモリ上に保持しておき複数のプロセッサがポインタを介してそれを共有する構成が適していると考えられる。このとき，メモリアクセスにおけるlatencyが問題となるが，データ駆動方式ではこの回避が期待できる。

本稿では，以上のデータ駆動方式の利点を検証するために開発中のPIM-Dシステムのアーキテクチャについて述べる。

2．並列型Prologの実行方式　PIM-Dは，OR並列型PrologとAND並列型Prologで記述されたプログラムを高速に実行するために，次の特徴を有している。

①Prologプログラムをコンパイルによってデータフローグラフに変換
- コンパイラは，OR並列型Prolog用とAND並列型Prolog用の２種用いる。

②引数間のユニフィケーションの並列実行
- 複数の引数によって共有される論理変数については，並列にユニフィケーションを行ったあとで，無矛盾性の検査を行なう。

③OR並列型Prologは，ストリームを用いて実現
- 解を生成する複数のプロセスと，消費する複数のプロセスを１本のストリームで接続することにより，全解探索を行う。
- ストリームは共有メモリ上に生成される。

④AND並列型Prologは，テスト&セット命令等による排他制御機構により実現
- ガード機構を実現するために，セマフォ変数を設ける。
- ガードの呼出しにおいて，セマフォ変数に対してテスト&セット命令を実行する。
- テスト&セットが成功した節は，頭部及びガードの処理によって生成された結合環境をbind命令によって上位の節に公開する。

以上の方式により，P([X.a].b)<- ・・・なる節は，図１に示すデータフローグラフにコンパイルされる。

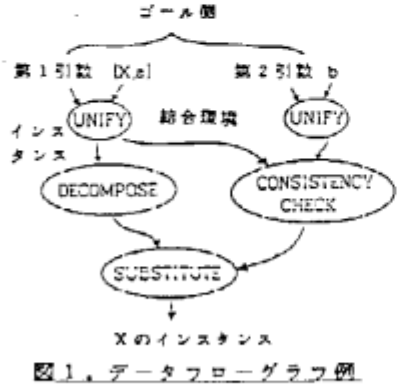


図１．データフローグラフ例

3．PIM-Dのアーキテクチャ　PIM-Dは，図２に示すように演算エレメント(PE)．構造メモリモジュール(SMM)．サービスプロセッサ(SVP)，及びこれらを接続するネットワークからなる。

PEは，データフローグラフを解釈・実行する。PEは，図３に示すように，トークンのバッファリングを行なうPQU(Packet Queue Unit)，オペランドの待合せを行なうICU(Instruction Control Unit)，発火した命令を実行するAPU(Atomic Processing Unit)．及びローカルメモリLM(Local Memory)からなる。

SMMは，共有メモリとして用いられ，構造体データを保持する。SVPは，PIM-D全体の初期化・保守を行なう。ネットワークはこれらのモジュール間で通信されるトークンの交換網である。各モジュールの諸元を表１に示す。PIM-Dのアーキテクチャ上の特徴を以下に示す。

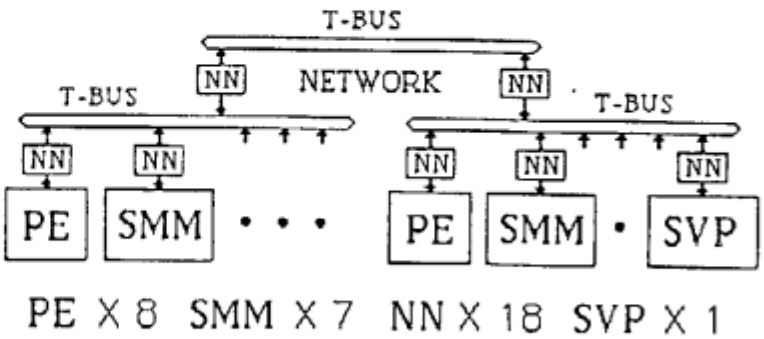


図２．ＰＩＭ－Ｄのハードウェア構成

①両方のPrologをサポートする命令セット
- 共通に使われる命令 ― unify. consistency-check. substitute 等
- OR並列型Prolog用命令 ― create_stream , append_stream等
- AND並列型Prolog用命令 ― test&set. bind 等

②構造体データ共有方式による共有メモリ
- リファレンスカウント方式によるデータの共有
- データに Ready フラグと Pending フラグを付加し，非同期アクセスを実現

③２０ビット長データ
- 符号付整数データとして 524287 ～ -524288
- 構造体データのアドレス空間として，１SMM当り１M語

④データ型をタグとしてデータに添付
- 整数，文字列，ストリーム等２１種のデータ型
- データ型の判定を専用回路化し，ユニフィケーション等を高速化

⑤階層型ネットワーク
- 並列処理におけるローカリティを活かすことにより，高スループットを実現
- 均一型ネットワークに比べ，ハードウェアの負担が少なく，PEの台数が増加しても対応容易

⑥メモリ参照の高速化
- SMMのフリーセルアドレスを予めPEに与えておくことにより，PEのフリーセル要求を削減
- 共有変数の結合情報等の特定のPEにしか参照されないデータは，ローカルメモリ(LM)に保持

⑦測定用回路の組込みによるリアルタイムのモニタ
- 各ネットワークノード(NN)の負荷
- トークンキュー(PQU)，オペランドメモリ(OM)等のハードウェア資源の使用率

５．おわりに　データ駆動により並列に推論を行なうことを特徴とする並列推論マシンPIM-Dについて述べた。現在，ハードウェアのデバッグ中である。今後，ソフトウェアの充実と，システムの評価を通して中期末に１００台規模のPEからなるシステムの実現に目処をつける予定である。

謝辞　ICOT関係各位に感謝する。

【参考文献】

1) Arvind and Iannucci. R. A.. "A Critique of Multiprocessing von Neumann Style." Proc. of 10th Int. Symp. on Computer Architecture. June 1983.

2) 伊藤 他，「データフロー方式推論マシンのアーキテクチャ」. Proc. of Logic Programming Conf.. ICOT. Mar. 1984.

3) 伊藤，「並列型Prologのデータフローマシンによる実行方式」，日経エレクトロニクス．1984年11月5日号．

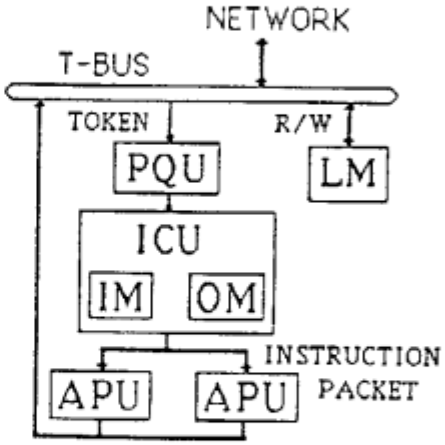


図３．ＰＥのハードウェア構成

表１．モジュールの諸元

ＰＥ	
トークン長	86ビット
トークンキュー(PQU)容量	16kトークン
命令メモリ(IM)容量	96k×59ビット
オペランドメモリ(OM)容量	32kトークン
演算ユニット(APU)	× 2
マシンサイクル	300n秒
マイクロプログラム制御	
演算器	20ビット巾
ローカルメモリ(LM)容量	0.5M×32ビット

ＳＭＭ	
演算部	ＡＰＵ×１
記憶部	
容量	0.5Mセル
１セルの構成	
・データ＋タイプ	32ビット×２語
・リファレンスカウント	10ビット
・Ready. Pendingフラグ	

ネットワーク	
ネットワークノード(NN)数	１８
ネットワークノード(NN)	
転送時間	450n秒(min)／トークン
内部バッファ容量	128トークン

データフロー方式並列推論マシンにおける ＡＮＤ並列型Ｐｒｏｌｏｇの実行方式

伊藤 憲義　来住 晶介　久野 英治　六沢 一昭
（ＩＣＯＴ）　（沖電気）

１．はじめに

著者等はデータフローモデルに基づく並列推論マシンPIM-D(Parallel Inference Machine Based on the Data Flow Model)の研究開発を進めており，このマシン上で目的を異にする２つのタイプの論理型プログラミング言語（ＯＲ並列型Prolog，及び，ＡＮＤ並列型Prolog）をサポートするための検討を行っている．これらの言語で記述されたプログラムは基本的にマシン言語に相当するデータフローグラフに翻訳され，いずれもストリームを介した並列処理を実現する．このうちＯＲ並列型Prologの処理方式に関してはその細部の検討を行ってきた［１］．Concurrent Prologに代表されるＡＮＤ並列型Prologは優れた記述性を有しており，オブジェクト指向型プログラミングの可能性を与える［５］．本稿では，PIM-D におけるＡＮＤ並列型Prologの実行方式について述べる．

２．ＡＮＤ並列型Prolog

上記２つの言語はいずれも統一化，及び，非決定性の制御機能を内蔵している．このうち統一化の機能については，両者で共通であるが，非決定性制御機能に関しては，ＯＲ並列型Prologが'don't know nondeterminism' の機能を必要とするのに対して，ＡＮＤ並列型Prologは'don't care nondeterminism' の機能を必要とする．即ち，ＡＮＤ並列型Prologは，ガード付きコマンドの機能を用いてＯＲプロセス間の排他的制御を行い，実行を継続するＯＲプロセスをたかだか１つに限定している．ＡＮＤ並列型Prologの特徴は並列動作するＡＮＤプロセス間で論理変数を共有させるとともに上記ガード制御機能を用いてプロセス間のインタラクティブな通信手段を提供することにある．以下に，Concurrent Prolog によって記述した簡単なエディターの例を示す．

```
editor([insert(E)|C],DB) <- | editor(C?,[E|DB]).  ①
editor([delete(E)|C],DB) <- |
        delete(E,DB?,NewDB) // editor(C?,NewDB).  ②
editor([replace(E,F)|C],DB) <- |
        replace(E,F,DB?,NewDB) //
        editor(C?,NewDB).          - - - - - - -  ③
editor([display(DB)|C],DB) <- | editor(C?,DB).   ④
```

図１．editor述語の定義

ここで，小文字で始まる文字列はシンボル定数を示しており，大文字で始まる文字列は変数を示す．記号”<－”は含意を示しており，その右辺が成立するならば左辺が成立することを意味する．記号”｜”はコミットオペレータであり，ガード付きコマンドと同様に，最初にコミットオペレータまでの処理が成功した節のみが以降の処理を継続することができる．また，記号”／／”はその両辺のリテラルを並列呼出しすることを意味する．各々の節において，含意記号の左辺を頭部，その右辺のうちコミットオペレータまでをガード，それ以降を本体と呼ぶ．

editor述語はその第一引数及び第二引数として，それぞれ，コマンド列及び現在の内部状態（データベースDB）を受取り，コマンドで指定された処理を行った後，新しい内部状態を生成して，editor述語を再帰呼出しする．即ち，無限再帰呼出し機構を用いて，内部状態を持つ一種のプロセス（オブジェクト）を実現しておりオブジェクト指向プログラミングの可能性を与える．コマンド列はnon-strictな構造データであるストリームとして表現されており，読出し専用タグ”？”を用いた待合わせ機構により，先頭のコマンドが到着すれば（即ち，コマンド列を示す変数Ｃがリストに結合されたとき），節①から④までの頭部統一化の処理が起動される．起動された節群はコマンド列から先頭コマンドを取出してその判定を行うが，コミットオペレータにより唯一つの節のみが選択されて以降の本体の呼出しが継続される．

このようなプログラム上で陽に”見る”ことのできるストリームを可視ストリーム(visible stream)と呼ぶ．ＯＲ並列型Prologにおいても，ゴールに対する複数の解をストリームの形で次の処理へ渡しながら並列処理を実現するが［１］，このようにプログラマに対して明示的に見えないストリームを不可視ストリーム(invisible stream)と呼ぶ．

ガード呼出しが存在する場合，呼出された節で生成された論理変数（共有変数）に関する結合情報はコミットオペレータ実行までその節内で局所的である．即ち，共有変数に関する結合情報は，節の頭部及びガードを実行している間は，節を呼出した親ゴールに対して隠蔽される．このような機構を実現するために，ガード引数として共有変数（又は共有変数を含む構造データ）が存在する場合に共有変数をガード呼出し前にコピーする事前コピー法と，実際に必要が生じた際にコピーする遅延コピー法がある［２］．事前コピー法は共有変数をメモリセルへのポインタとして表すことができ，コピーする際に変数の名前変換機能を用意することによって実現される．遅延コピー法は共有変数に対する結合環境を個々の節毎に連想リストの形で管理し

ておきコミットオペレータを実行した時点で親に結合環境を転送することによって実現される．ガード呼出しが頻繁に発生する場合は遅延コピー法の方が効率が良いと思われるが，一般にはガードで使用されるのはその多くが引数の内容をテストするリテラルである場合が多い．この様な場合は上述のようなコピー操作は不要である．従って，実現が比較的容易な事前コピー法を使用する（必要な場合のみ明示的にコピーする）方向で検討を進めた．

3．　実行方式

上述のような処理を行うためのデータフローグラフ表現を図２に示す．同図は上例の節④をグラフ表現したものであり，頭部統一化，コミットオペレータの処理，及び本体の呼出し処理に分けられる（ガードは空であるのでその呼出し処理は省略される）．unify&decompose プリミティブはunify プリミティブと構造を分解するdecompose プリミティブを組合わせたものであり，その入力オペランド（ゴール側引数）が構造データ（リスト又はベクタ）であるか否かの判定を行い，これが成功した場合は要素に分解する．この場合，入力オペランドが共有変数であれば（変数自体を分解して）新しい構造データを生成して，変数をその構造データ（インスタンス）に結合するための結合環境を生成する．結合環境は変数とそのインスタンスからなるリストの形で表現され，インスタンスを共有変数の指すメモリセルへ書込む操作はbindプリミティブによって行われる．unify プリミティブは２つの項（図２の場合は節④の頭部リテラルに２度現れる変数DBのインスタンス）の間の統一化を行うものであり，それらの共通インスタンスと共有変数に関する結合環境を出力する．check-consistency プリミティブは上記unify プリミティブ群によって得られた結合環境の間の無矛盾性検査を行う．この結果は頭部に現れる変数群に対するインスタンスの置換のためにsubstituteプリミティブ群に送られる．

上述の頭部統一化の処理が成功裏に終了すると（即ち，先頭コマンドが到着ししかもそのコマンドがdispaly(D)の形をしていれば）test&set及びbindプリミティブが起動される．test&setは並列に実行される他の節群との間で排他的制御を行うプリミティブであり，節間で共有されるセマフォ（定義を呼出した時点でcreate-semaphoreプリミティブによって生成され初期化される）の内容を返すと共にセットする．このプリミティブの結果は頭部統一化及びガードの呼出しが最初に成功終了した節であるか否かを示しており，その場合は節の本体を呼出す．そうでなければ，本体の呼出しは抑止される．定義中の全ての節の頭部及びガードの処理が失敗したとき，定義を呼出したゴールに対してその旨を知らせるために，参照数による管理が行われる．即ち，参照数は節の数に初期化され，節の処理が失敗する度に１ずつデクリメントされる．この数がゼロに達するとゴールに対して'fail'が返される．ゴールはこの'fail'を受取ると以降の処理の実行を抑止する．

bindプリミティブは上記check-consistency プリミティブによって得られた頭部統一化の結合環境をもとに，共有変数へのインスタンスの書込みを行う．即ち，このbindプリミティブを実行するまでは共有変数の結合情報は実行中の節内で局所的である．

4．　おわりに

データフロー方式並列推論マシンPIM-D 上でのＡＮＤ並列型Prologの実行方式を示した．現在そのソフトウェアシミュレータによる評価を進めており［４］，さらに実験機［３］による評価も行う予定である．

最後に，日頃御指導いただく村上第一研究室長及び尾内主任研究員をはじめとするＩＣＯＴ関係者に深謝する．

＜参考文献＞

［１］伊藤他，「データフロー方式Prologマシンにおける非決定的制御機構」，情報処理第２７回全国大会講演論文集 4p-3，１９８３．１０月．

［２］伊藤，「並列型Prologのデータフローマシンによる実行方式」，日経エレクトロニクス，1984.11.4 日号

［３］来住他，「データフロー方式並列推論マシン実験機の構成」，本大会講演論文集　6C-2，１９８５．３月．

［４］久野他，「データフロー方式並列推論マシンにおけるＡＮＤ／ＯＲ並列効果のシミュレーションによる測定」，本大会講演論文集　6C-3，１９８５．３月．

［５］Shapiro,E.Y., "A Subset of Concurrent Prolog and Its Interpreter," TR-003,ICOT,Jan. 1983.

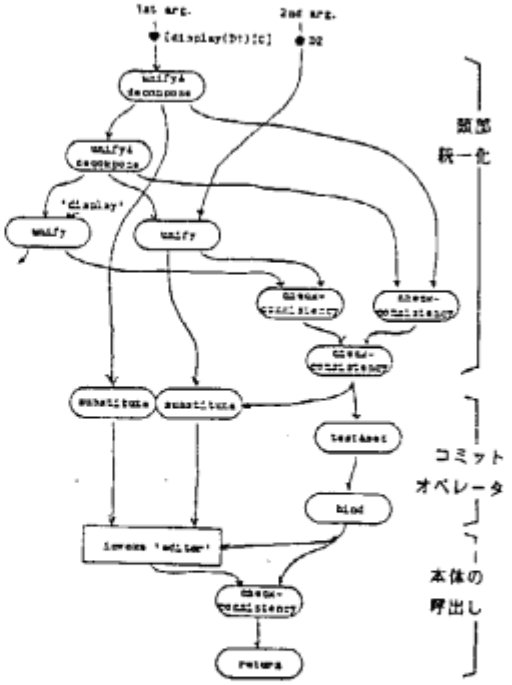


図２．　節④のデータフローグラフ表現

PSIの性能評価（1）

1C-2

西川 宏　山本 明　横田 実（新世代コンピュータ技術開発機構）
中島 克人（三菱電機）　三井 正樹　吉田裕之（沖電気）

1．はじめに

逐次型推論マシンPSI は核言語KL0 を機械語とし，これをマイクロプログラムで直接実行／解釈する論理型プログラム向きの高級言語マシンである[1,2]．第五世代コンピュータプロジェクトにおける前期三年間を通したPSI マシンのハード／ファームウェア開発もようやく終わりに近づき，各種性能評価をPSI 上で行えるようになった．本報告では各種ベンチマークプログラムによるPSI の基本性能の評価結果について述べる．あわせて評価結果に基づいて行った高速化についても論ずる．

2．評価について

各種ベンチマークプログラム及び応用プログラムの諸特性を実機上で測定／評価することで，ユーザの観点からはそのマシンの大枠の実行速度の目安が得られる：一方設計者の側からは各種基本機能の評価のためのデータ，アーキテクチャ全般に渡る考察の材料が得られる．

特に設計者の立場からは，

(1)データ収集結果を解析することで各基本機能の実行速度が得られ，どの機能の実行が"重い"かがわかる．

(2)応用プログラムの各特性を測定することで各基本機能及び組込述語の使用頻度が得られ，(1) の結果とあわせどの機能の最適化を図ればよいかがわかる．

(3)さらにアーキテクチャ上の種々の試みについても，その機能を具備しない場合との比較を同一マシン上で行って，それらが有効であったかの検討を行える．

このような目的で我々は各種プログラムを用意した．それらの内分けについて記す．

Prologの基本機能は，クローズの呼出し及びその環境の退避，引数間のユニフィケーション，そしてバックトラックに分類できると考え，これらの特定項目を測定する小さいベンチマークプログラムを作成し計測を行った．

それと並行してすこし大きめのベンチマークプログラムとしてPrologコンテスト[3] の中で取上げられたプログラムのいくつかについてその実行速度の計測も行った．これらはいずれもDEC-10 Prolog 上でも実行時間が測定され，PSI の結果との間で比較が行われた．これにより当所我々の開発したマイクロインタプリタはDEC-10のほぼ40～70%の性能であったが，後述する高速化の検討を加えたことで，現在ではほぼ同等の性能が達成されている．

応用プログラムとして，SIMPOSのWindowシステムの一部，8-パズル及びBUP システムを選んだ．これらを選んだ理由はシステムプログラムとアプリケーションプログラムとでは組込述語やPrologの機能の使い方に違いがあると思われる：またオブジェクト指向で書かれたプログラムの特性を計測したいためでもある[4]．

3．各種評価方法とその結果について

a．評価方法

実行時間に関する測定は，PSI 内のリアルタイムクロック（精度は1msec)を用い，適当な回数のループでプログラムを複数回走らせ，実行時間を計りその平均をとった．DEC-10 Prolog の実行速度の測定には組込述語statisticsを用いて，同様の手法で行った．

実行頻度に関するものは，マイクロプログラムの適当な場所にパッチをあてて，計測用のマイクロプログラムをうめこんだ状態でプログラムを走らせ測定を行い，その計測結果は主記憶内のテーブルに格納することにした．

ハードウェア機能（例えばキャッシュのヒット率等）に関する測定はマイクロ命令の全トレース結果から解析を行っている[5]．

b．評価結果

表1には簡単なプログラムの説明とともに基本機能の測定結果をDEC-10 Prolog との比較で示してある．

1 は，クローズ呼出しと，アトム同志のユニフィケーションについての測定結果を，2 はクローズ呼出しと構造体同志のユニフィケーションを，3 は 1に加えて，環境の退避とその復帰作業が含まれる．4,5 はいずれもバックトラックに関するものである．

この結果をみると，PSI はクローズ呼出しに伴う環境の退避，およびその情報の復帰に時間がかかっている．これはPSI のコントロール情報が10語とDEC-10 Prolog に比して大きいためと考えられる．実際マイクロ命令のトレース結果からは3 の場合では約1/3 がこのコントロール情報の退避に費されていた．しかしこれは，コントロール情報の退避タイミングをマイクロプログラム中に分散させることで，高速化できると考えられる．バックトラックについては，DEC-10 Prolog ではクローズ呼出しの際，第一引数の値でハッシングを行い（クローズインデキシング），余計なバックトラックを生じない工夫がなされており 4,5では

その機能が働かないよう第一引数の値は未定義としてある．shallow バックトラックがDEC-10より遅いのは今後の検討としたい．

表 2はPrologコンテストのプログラムの一部を測定した結果である．これをみるとほぼDEC-10 Prolog と同等の性能がPSI 上でも実現されている．リスト30要素の逆転（nreverse）が遅い理由は，PSI 上の測定ではクローズインデキシングを用いなかったこと，及び単純なプログラムではマイクロインタプリタのオーバーヘッドが小回りのきく通常の機械語と比べ大きいことによると考えている．一般に高級言語マシンでは，単純な場合にはオーバーヘッドが大きいが，問題が複雑化すればそれが相殺され，逆にマイクロインタプリタの良い点が出てくるとされているが，PSI でもその傾向が見られる．(表 2参照)

4．高速化について

評価作業を進めるうちに，予定された性能が達成されてない事がわかり，マイクロインタプリタの高速化の検討を並行して行い，最適化したほうがよい次の項目洗い出した．

マイクロインタプリタ全体に渡る項目としては

(1)使用頻度の高いマイクロモジュールの最適化

ユニファイヤに関しては

(1)ソースコード中に出現するデータの処理とスタック中に生成されたのデータの処理を分離し，コード中のデータに対する最適化ルーチンの作成

マイクロインタプリタの基本制御部に関しては

(1)メモリ割当てタイミングの変更によるメモリセル初期化の回数の削減

(2)さらに応用プログラムを評価するとゴール側の引数が単にパラメータパッシングとして用いられる場合が多く，すべての引数が単なるパラメータ渡しとして機能するときは，ユニファイヤを起動しないバイパスルーチンの作成．

(3)組込述語の引数の属性があらかじめ判定できる場合の最適化引数取り出しルーチンの作成

これらの項目のうち最初のステップとして，インタプリタの基本制御部とユニファイヤの最適化を行った．その変更をした結果，appendプログラムの場合，1 ロジカルインファレンスあたりの実行がマイクロ命令で 244ステップを要したが，本最適化により 143ステップに短縮できた．組込述語については最適化はまだ行っておらず，すべてがこのように速くなるわけではないが，Prologコンテストのプログラムの実行では平均50 %の高速化が，応用プログラムでは20～30% の高速化が達成できた．

5．おわりに

本報告ではPSI の現時点での評価結果についてまとめた．今後も評価作業を続けて，さらにいろいろなデータを収集していくことを考えている． 2項でのべた(1),(2) はマシンをより実用的にするためにも必要であるが，我々は(3)で述べたように，アーキテクチャ上の各種機構の有効性を検証するためにマイクロプログラムの書き替えを行って各種計測/ 評価を始めたところである．

最後にPSI マシンの開発に携った方々に深謝します．

- 参考文献 -

[1]Yokata.H et al.: A microprogrammed Interpreter for the Personal Sequential Inference Machine. PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON THE FIFITH GENERATION COMPUTER SYSTEMS ,1984

[2]Taki.K et al.: Hardware design and Implementation of the Personal Sequential Inference Machine (PSI). ibid .

[3] 奥野：第三回LISPコンテストおよび第一回PROLOGコンテストの課題案．記号処理 28-4,1984

[4] 横田他: PSI におけるオブジェクトサポート (本稿)

[5] 中島他: PSI の性能評価(2) (本稿)

Table 1 : Basic Charateristics

Test	1	2	3	4	5
PSI[msec]	0.89	1.42	1.26	0.65	2.73
DEC[msec]	0.99	1.69	1.19	0.49	3.68
DEC/PSI	1.1	1.2	0.9	0.8	1.3

1) call + unify(atoms)

```
test   :- p1(a).
p1(a)  :- p2(a).
 ........
p99(a) :- p100(a).
p100(a) .
```

2) call + unify(structures)

```
test :- p1(f(a)).
p1(f(a)) :- p2(f(a)).
   ............
p99(f(a)) :- p100(f(a)).
p100(f(a)).
```

3) call + save_control + unify(atoms) + return

```
test :- p1(a),p2(a), .... p99(a),p100(a).
p1(a).
p2(a).
 ..
p100(a) .
```

4) backtrack(shallow)

```
test :- p1(X,a).
p1(X,b1).
p2(X,b2).
.....
p99(X,a).
p100(X,a).
```

5) backtrack(deep)

```
test :- p0(X,a).
p0(X,a) :- p1(X,a).
p0(X,b).
....
p99(X,a):- p100(X,a).
p99(X,b).
p100(X,b).
```

Table 2 : Result of Benchmark

benchmark programs	DEC [msec]	PSI [msec]	DEC /PSI
nreverse (30 elements)	9.48	14.6	0.65
quick sort (50 elements)	14.6	16.1	0.91
tree traversing	61.1	52.2	1.17
lisp interpreter (trai 3)	4360	4050	1.08
lisp interpreter (fib 10)	402	375	1.07
lisp interpreter (reverse 30)	194	174	1.11
eight queen	97.5	105	0.93
reversible func. (solve)	41.7	38.9	1.06
slow reverse (srev 4)	5.42	6.13	0.88

* DEC-10 Prolog : mode declaration + fast code (on DEC-2060 ; load average 2～3)

並列処理方式論理シミュレータの一検討

野田 泰徳　木下 哲男　奥村 晃　平野 達郎
沖電気工業(株)

1．はじめに

近年、論理回路の大規模化に伴い論理シミュレータの高速化への要求が強まるばかりである。これに対処するため、マルチプロセッサ構成による並列処理方式の専用マシンが開発されている。

我々は、ソフトウェアの面から並列処理方式を考え、第5世代コンピュータ計画で並列推論マシン上の核言語の母体として検討されている並列事象記述言語 Concurrent Prolog (CP)[1]を用いて論理シミュレータを試作した。

本稿では、そこで用いた並列シミュレーション方式、CPによる実現法等について、実験例と共に検討を加える。

2．Concurrent Prolog (CP)

CPは並列実行のセマンティクスを持つ並列プログラミングのための論理型言語である。このCPは sequential Prolog を subset として含み、同時にまた and_parallelism と or_parallelism を導入し、前者を並列プロセスの記述に、後者を non_deterministic プロセスの記述に用いている。また並行して走るプロセス間で共有される変数をプロセス間通信用に用いており、この共有変数でプロセス間の同期が記述できる。

このプロセスを tail recursion program により表現すると、同一の名前のプロセスが生き続けることになり、このプログラムの引数をプロセスの保持する状態と解釈するとプロセスはオブジェクトを表すことになる。即ち、CPでオブジェクト指向的なプログラミングが可能となる。[2]

3．並列シミュレーション方式

論理シミュレータに並列動作の概念を導入する場合、実際のハードウェアの動作に基づき、各コンポーネント毎の並列性を考えるのが自然である。この時、これらコンポーネント群の並列動作の制御方法として、グローバル・タイマー方式とローカル・タイマー方式が考えられる。

(1) グローバル・タイマー方式

シミュレータがグローバル・タイマーを有し、プロセスとして表現されたコンポーネント全体をこのタイマーが発するクロックにより同期的に実行させる方式である。各コンポーネントが同期している（同じ時刻を持つ）ことから、コンポーネント間の通信はイベント値だけで良く、通信の起った時刻がそのイベント時刻となる。

(2) ローカル・タイマー方式

グローバル・タイマーを設定せず、各コンポーネントに分散しているローカル・タイマーに従ってシミュレーションを行なう方式である。各コンポーネントの時刻が異なることからイベントの値と時刻を通信する必要があり、コンポーネントの次の時刻は到着するイベントの持つ時刻に依って決定される。

4．CPによる並列シミュレータ

筆者らは以前、オブジェクト指向プログラミング環境の基でシミュレータを試作し、シミュレーションに於けるオブジェクト指向プログラミングについて考察を加えている。[3]そこで今回は、CPでもオブジェクト指向プログラミングの形態が可能なことから、基本的考え方を踏襲し、新たに並列性を導入してシミュレータを実現した。この時、CPの持つ並列プロセスの概念とストリームによるメッセージ通信機能をシミュレーションメカニズムの中に有効に取り入れるよう配慮した。 以下、グローバル・タイマー，ローカル・タイマー各方式についてその実現形態の概念を示す。

(1) グローバル・タイマー方式シミュレータ（図1参照）

timer_process は同期を取る為に総てのプロセスに対してクロックを発し、Chain[4]の概念を用いて総てのプロセスで処理が終ったことを知れば、クロックをインクリメントする。

evaluator_process は、クロックが到着する（共有変数が non_variable になる）と内部状態として保持していた各入力端子の値を基に出力値を計算し、イベントが発生すればその値を delay_processへ送る。そして、次の時刻に備えて到着している入力イベントを取り込む。また、delay_process からイベントを受け取ると次段の素子へ送る。

delay_process は素子の遅延を実現するもので、evaluator_process からイベントを受け取った時にプロセスは生成され、所望の遅延時間が経過した後イベントを返して消滅する。

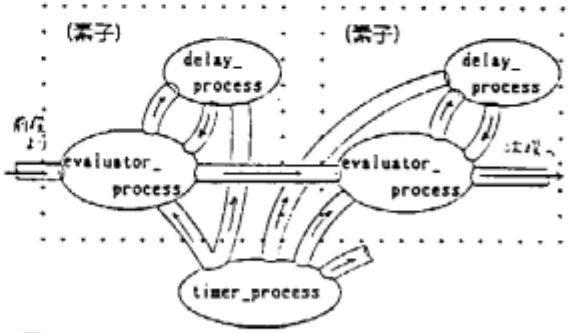


図 1．グローバル・タイマー方式シミュレータのプロセス構成

(2) ローカル・タイマー方式シミュレータ（図2参照）

port_process は素子の各入力端子毎に存在し、前段の素子からのイベントを受け取る queueを持つプロセスである。

port_manager_process は port_process から到着したイベントを受け取り、その中で最も若いイベント時刻を持つ端子について、内部状態として持っているその現在値を更新すると共に、採用したイベント時刻をその素子の現在時刻とする。このport_manager_process は、port_process にイベントが到着していない時にはsuspend するよう同期がとられている。

evaluator_process は port_manager_process で設定された現在値を基に出力値を求め、遅延を考慮して出力イベントとして次段の素子へ渡す。

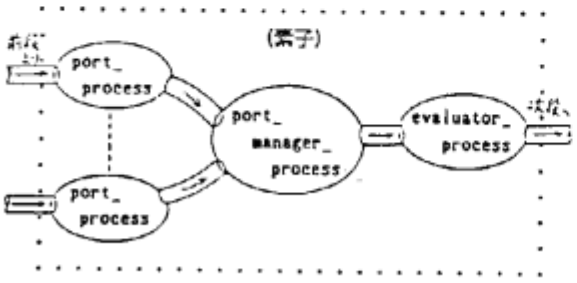


図 2．ローカル・タイマー方式シミュレータのプロセス構成

```
mpx(N. T. Inx. Iny. Ctr. Out. F1-F8) :-
   nand2(N_na1. T. M2?. M3?. M4. F1-F2).
   nand2(N_na2. T. Ctr. M1?. M5. F2-F3).
   nand2(N_na3. T. M4?. M5?. M6. F3-F4).
   inv(N_iv1. T. Ctr. M2. F4-F5).
   inv(N_iv2. T. Inx. M3. F5-F6).
   inv(N_iv3. T. Iny. M1. F6-F7).
   inv(N_iv4. T. M6?. Out. F7-F8).
                  Read only Annotation

fadd(N. T. Inx. Iny. Cin. C. S. F1-F6) :-
   inv(N_iv1. T. Inx. M1. F1-F2).
   inv(N_iv2. T. M2?. M3. F2-F3).
   mpx(N_mpx1. T. Inx. M1?. Cin. M2. F3-F4).
   mpx(N_mpx2. T. Inx. Iny. M2?. C. F4-F5).
   mpx(N_mpx3. T. M2?. M3?. Iny. S. F5-F6).
                                  Chain用生成

add_3(T. Ix0. Ix1. Ix2. Iy0. Iy1. Iy2.
                  S0. S1. S2. C. F1-F4) :-
   fadd(fa1. T. Ix0. Iy0. G?. C0. S0. F1-F2).
   fadd(fa2. T. Ix1. Iy1. C0?. C1. S1. F2-F3).
   fadd(fa3. T. Ix2. Iy2. C1?. C. S2. F3-F4).
   fix(g. 0).

add3 :-
   timer(add3. [A|F1]-[A|F7]. T-(-1)).
   input(T?. Ix0. Ix1. Ix2. Iy0. Iy1. Iy2. F1-F2).
   add_3(T?. Ix0. Ix1. Ix2. Iy0. Iy1. Iy2.
                  S0. S1. S2. C. F2-F3).
   probe('Output of SUM_0'. T?. S0?. F3-F4).
   probe('Output of SUM_1'. T?. S1?. F4-F5).
   probe('Output of SUM_2'. T?. S2?. F5-F6).
   probe('Output of CARRY'. T?. C?. F6-F7).
```

図 3．3_bits adder の構造記述（グローバル・タイマー方式）

ＣＰにより上記各方式でシミュレータを実現してみると、グローバル・タイマー方式に於て同期を取るクロックの管理に関する記述が面倒であり、ローカル・タイマー方式の方がシンプルに記述できた。

5．シミュレーション適用例

3_bits adder を対象としてシミュレーションを行なってみた。図3にグローバル・タイマー方式に於ける対象の記述例を示す。ここで、述語 add_3が対象の構造を示しており、add3 でシミュレータを起動させる。また、probe は値を観測する為のプロセスである。この方式では構造記述の中に、タイマーからのクロックと Chainのための変数が必要となっている。しかしローカル・タイマー方式ではこれらは不要であり、実際のハードウェア構成を示すだけで良い。

シミュレーションを実施した結果、グローバル・タイマー方式は実際のハードウェアの動作イメージに従っているため結果のトレースも容易であった。一方、ローカル・タイマー方式は各コンポーネントで時刻が異なっているため、probe プロセスからコンソールへの出力が時系列的に得られない。従って、こうしたインターフェイスを工夫する必要性等が判明した。

6．まとめ

ＣＰの持つ並列プロセス、ストリームによるプロセス間通信の概念を用いることにより、制御が難しいと考えられる並列型シミュレータを実際のハードウェア構成に基づいて簡潔に構成できた。

特に、ＣＰでプロセス間の同期を実現する Read only Annotation[1]がシミュレータの実現に有用であった。またオブジェクトの概念に於て、ＣＰでは状態を静的に表現する手段を持たないため述語の引数で内部状態を表現している。従って、メモリー等の内部状態数の多い素子の表現については検討が必要と思われる。現在、更に対象回路の規模を拡大し、同時に、多種の機能素子を表現して検討を進めている。

尚、本研究は、第５世代コンピュータプロジェクトの一環として行なわれた。御討論いただいたＩＣＯＴ第２研究室古川室長に感謝する。

文献

(1) E. Y. Shapiro: A Subset of Concurrent Prolog and Its Interpreter. ICOT Technical Report TR-003 (1983).
(2) E. Y. Shapiro & A. Takeuchi: Object Oriented Programing in Concurrent Prolog. New Generation Computing Vol. 1 No. 1 (1983).
(3) 野田 泰徳，木下 哲男，奥村 晃，平野 達郎：ＣＡＤシステム構築に関する一検討. Logic Programing Conference'84. 14-5 (1984).
(4) How to Solve It in Concurrent Prolog. ICOT.

並列推論処理システム
——株分け方式の実験——

7c - 7

板敷晃弘　久門耕一　佐藤健　増沢秀穂　相馬行雄
（富士通株式会社）

1．はじめに

筆者らは，第５世代計算機用の並列推論マシンの研究を行っている。58年度には，並列推論処理方式の一つを提案し，シミュレータや小規模実験機を試作し，並列推論に関する基礎データの収集・評価を行った。この経験を基に，今回新たな並列推論処理方式を提案する。「株分け方式」と呼ぶ本方式の詳細は本大会の7C-8を参照されたい。本方式の基本は，

「深さ優先の逐次処理に基本を置いて処理を進めながら，他のプロセッサから仕事を要求されたなら，自分の仕事から未処理でかつ大きな仕事を分け与える」

というものである。

ここでは，本方式の実験結果を報告する。

2．実験の目的

株分け方式の有効性を定量的に確認する事が主眼であり，同時に，問題点の抽出を行うことも目的とする。具体的には，主として次の二点を確認する事である。

① 株分け方式は，逐次処理に基本を置いて，PE単体の性能を高くし，その PE を多数台結合して高性能化を図ろうとするものである。従って，PEの性能が通常の逐次型処理系の性能に比して遜色ない事を確認する。

② 株分け方式は，PE内で逐次型処理と同じ処理をする時間を多くし，通信を極力減らすことを狙っている。そのために要求された時にだけ，かつなるべく大きな仕事を分け合うようにした。その効果を確認する。

3．　実験システムの概要

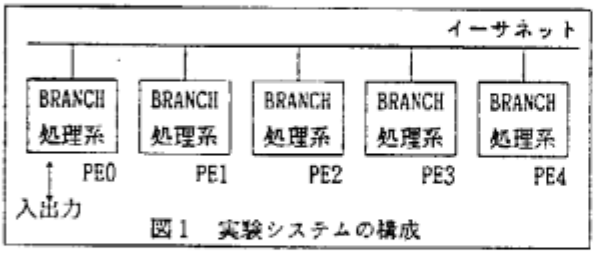


図１　実験システムの構成

ハードウェア構成は，要素プロセッサ（PE）としてのSUN ワークステーションを５台使いイーサネットで結合したものである。各PEには株分け方式に基づくprolog並列処理系（BRANCH処理系）をインプリメントしている。５台のうちの１台はユーザとのインタフェースなど入出力を扱うようにした。

4．実験結果

4.1　要素プロセッサの単体性能

(1)　並列化に起因するオーバヘッド

並列処理用として考えたBRANCH処理系１台の性能と，逐次処理用として考えた逐次処理系の性能とを比較した。数種類のプログラム（クィックソート，n-queenなど）による実験データから分かった事は，本BRANCH処理系のオーバヘッドは逐次型処理系に比して約 6%以下であり，連続的にPE内で処理をしている時には逐次型処理系に遜色ない事が明らかとなり，狙いは充分達せられている。

(2)　既存の処理系との性能比較

処理系の性能を表す尺度の一つとしてBRANCH処理系のLIPS値を表1 に示す。本処理系は約 1KLIPS の性能を得ている。

表1．　BRANCH処理系の性能

プログラム	実行時間	成功したユニファイ回数	LIPS
reverse 30	0.43 sec	496	1.1K
qsort 50	0.49	601	1.3K
6 Queen	2.76	2943	1.0K
8 Queen	53.74	55047	1.0K

本処理系をDEC2060 上のDEC10-prologおよびVAX780上のC-prologの性能と比較すると表2 のようになる。

表2．　BRANCH処理系と他の処理系の性能比

	revese 30	qsort 50
BRANCH処理系（SUN ）	1	1
C-prolog処理系（VAX780）	1.4	1.0
DEC10-prolog処理系（DEC2060 ）	2.3	1.9

4.2　処理の連続性

(1)　通信回数と仕事の大きさ

PE間の通信量（分岐された仕事の数）を調べた結果を表3 に示す。表中の値は，該PEに対して他のPEから依頼された処理の数を表す。

表3. 依頼された仕事の数

PE数：4 台	6 Queen	7 Queen	8 Queen	9 Queen
PE 0	1	2	9	3
PE 1	1	3	10	35
PE 2	1	3	6	23
PE 3	2	3	3	8
合計	5	11	28	69

一見して，通信の回数は非常に少ない事が分る。例えば，前回の方式では 6Queen の通信回数は約 100回であった。平均の仕事の大きさ（PE内で逐次処理型と同様な処理を行っている時間に相当する）は表4 のようになる。

表4. 仕事の大きさ

	総ユニファイ回数	通信回数	平均的な仕事の大きさ
6Queen	7947	5	～ 1600
7Queen	35139	11	～ 3100
8Queen	161227	28	～ 5700

仕事の大きさと，仕事の分岐に関して更に詳細に調べた結果を以下に示す。図2 は，7Queenのプログラムを 4 台のPEで実行した時のデータである。 図中の数字は，実行したユニフィケーションの総数である。分割した所では線の下に受け取ったユニフィケーションの数を示した。

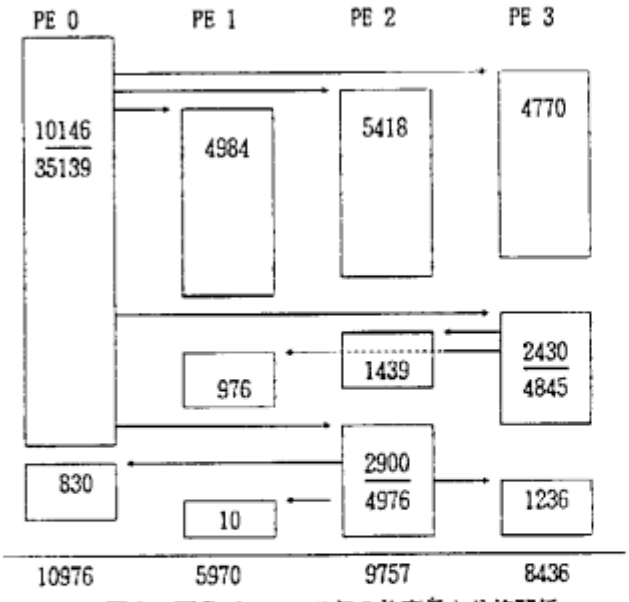


図2 要素プロセッサ毎の仕事量と分岐関係

株分け方式は，出来るだけ大きな仕事の単位で分岐して，通信を減らす事を狙っており，今回の実験で見ると，図2の様にこの目的はほぼ達せられている。

(2) 台数効果

要素プロセッサの台数を変えて，性能の変化を調べた。性能向上の比を図3 に示す。

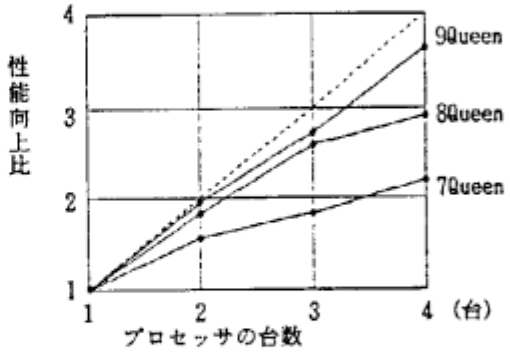


図3.台数効果

グラフでは，9Queenは性能向上比が大きく，8Queenと7Queenは伸びが鈍い。しかし，7Queenでもプロセッサ台数を優に凌ぐ充分な並列性（＞100 ）があることを既に確認している。グラフで伸びが鈍いのはPE内で通信と推論処理を並列に動作させていないことによる。PE内では推論処理の合間に一定間隔でidleなプロセッサがいないかをチェックしている。この間隔が長いために分岐するのが遅くなり，idle なプロセッサが仕事をもらうまでの時間が増えたためと思われる。

現在，上記の問題を解決するような構成とした並列推論マシンを試作中である。

5. まとめ

「株分け方式」と呼ぶ並列推論処理方式に基づいた実験システムを作成し，データ収集及び評価を行った。

その結果，以下のことが明らかに成った。

(1) 要素プロセッサの単体性能に関して

PE単体で動作させた時に，並列化に起因するオーバヘッドは6 %以下であり，既存の逐次型処理系と遜色ない性能を得た。

(2) 仕事の分割単位に関して

複数台の要素プロセッサによって処理する時の仕事の分割単位を予想どおり大きく出来，通信量を大幅に減らす事ができた。

なお本研究は第５世代計算機プロジェクトの一環として，ＩＣＯＴの委託で行ったものである。

参考文献

(1) 久門ほか「並列推論処理システム― 疎結合型並列処理方式―」本大会 7C-8

(2) 板敷ほか「ＰＲＯＬＯＧの並列処理システム」第29情処全国大会7B-4

6c－5

並列推論マシンＰＩＭ－Ｒの構造体メモリの一構成法

益田 嘉直　清水 肇　麻生 盛敏　尾内 理紀夫
（財団法人 新世代コンピュータ技術開発機構）

1．はじめに

関数型言語等をサポートするデータフローマシンにおいては、リストやベクタ等の構造体データの効率的な処理方式について長年各所で研究が進められて来ているが、論理型言語をサポートする並列推論マシンにおいては最近ようやく検討が開始されたばかりである[1],[2]。

現在、当機構においてリダクション方式に基づいた並列推論マシンPIM-R の検討ならびにシミュレーションによる評価が進められているが[3],[4]、本稿では構造体データの効率的な処理を実現するものとしてPIM-R における構造体メモリ(Structure Memory)の一構成法について紹介する。

2．構造体メモリの構成

図１に示すように、Structure Memory Module(以下SMM と呼ぶ）はIM-SM Network を介して多数台の Inference Module（以下IMと呼ぶ）と接続されるので、ある一定の条件を満たすSMM 内に格納される構造体データは多数台のIM より共有される。このためSMM の構成に際しては、将来のVLSI化のことも考慮し、以下の点に特に留意しながら検討を進めている。

(1) メモリアクセス競合の集中化を避ける。
(2) メモリアクセスの頻度を減らす。
(3) ネットワークや接続パスの信号線を減らす。
(4) 処理の実行中における不要セルの回収(Garbage Collection) をなるべく行わない。

上記(1),(2) のメモリアクセス競合等の対策としては、

(1) SMM をBank分け等により複数のSMM に分散化することにより、SMM の共有化により生じるアクセス競合の集中化を避ける。
(2) 数台のIMに対してSMM を１台接続したものを単位とするモジュールを構成し、それらを階層的に組合わせることにより全体システムを構成する[5]。
(3) 数台のIMに対して、同一の内容を持つSMM を１台ずつ接続して行き全体システムを構成する（図１）。

等の方法が考えられるが、現段階では上記(3) の方法によりメモリアクセス競合の集中化を回避する。この場合SMM の内容は同一であるため、各IM内のUnification 時にSMM から構造体データを読出す時には各IMにそれぞれ接続されているIM-SM Network を介して構造体データは読出される。

また、基本的には未定義変数を含まないリストやベクタ等の構造体データをSMM 上の専用メモリに格納することにより部分的に構造体データを共有する方式を採用することで処理の高速化、資源の有効活用を図る。即ち、プログラムのCompile 時にclause中の構造体データのうち、SMM に格納すべき未定義変数を含まないGround Instance の部分の切り分けを行い、それをSMM に格納する。この方式では読出しだけを行い、書き換えの生じないGround Instance の部分だけを共有するので、共有度は低いが次々と発生するプロセス間の独立性を高く保つことが出来るという特徴がある[2]。

3．構造体データの共有化の方式

clause中の構造体データのうち、指定された Ground Instanceの部分がSMM に格納され、そこへのポインタが持ち運ばれることによりSMM 内の構造体データがIM群から参照される。この時、clause中のポインタによって参照されている構造体データはUnification による新しいプロセス生成時にそのポインタが伝播されて、新しく発生するプロセスやgoalからもこの構造体データが参照されることになり共有化される。

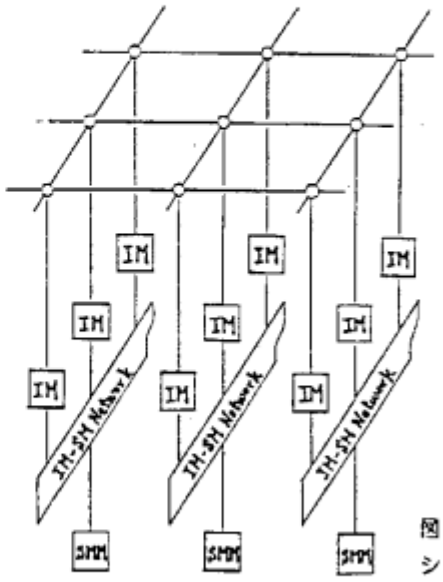


図１．
システム構成図

その結果、新しく発生するプロセスやgoalも長い構造体データではなくポインタを持ち運ぶことになり、プロセスやgoalの大きさが小さくなり、サイズの大きいリストやベクタ等の構造体データを多数含むプログラムに対しては高速化が期待できる。

この場合、clause中の構造体データのうちLengthや構造等の条件指定を行うことによりCompile 時にSHM へ格納すべきGround Instance の切り分けを行い、それをSHM へ格納する訳であるが、現段階ではプログラマによってGround Instanceの指示を行うことにより切り分けを行う。

4．構造体データの格納形式

構造体データを格納する方法にはリスト形式と連続するメモリセルにデータを格納するレコード形式があるが、リストとベクタの格納メモリを分離することにより上記の両形式の混用で検討を進めている。即ち、リストに対してはリスト形式で、ベクタ等の構造体データに対しては一次元配列のレコード形式でそれぞれ専用のメモリに格納する。リスト形式のリストについては、メモリの使用効率は落ちるが、必要に応じてポインタを１段ずつたぐりながらリストデータを取り込むなどの実現が可能で、ポインタを利用して容易にデータを取り出せる利点がある。一方、レコード形式のベクタについては、アドレステーブル経由でデータを読出すことによるオーバーヘッドや高速な連続読出しの実現などの問題があるが、ベクタのサイズが大きくなるにつれてアドレステーブルを読出すオーバーヘッドの割合いは減少するし、またメモリの使用効率が良いなどの利点がある。

これらSHM に格納された構造体データの読出しはBlock 単位に行う。即ち、リストの場合にはList Data Memory中のcar 部とcdr 部の２セル（List Blockと呼ぶ）がBlock 転送され、Unification Unit（以下UUと呼ぶ）内のバッファメモリに読出され、ベクタの場合にはVector Address Table 内のアドレスで指示されたVector Data Memory内のベクタの一まとまり（Vector Blockと呼ぶ）がBlock 転送されUU内のバッファメモリに読出される。この時、UU内のバッファメモリに読出されたベクタが、新たに別のベクタやリストのポインタを含んでいる様な構造体データであり、度重なるBlock 転送を必要とする場合などではUU内のバッファメモリは処理中の引数間のUnification が終了するまで解放されない。

更に、構造体データのGround Instance の部分を格納する方法としては、基本的には同一の内容に対しても複数のコピーを持たせるコピー方式を採用するが、実行時においては子プロセスの生成の時にポインタが受け渡されて行くことになり、同一の構造体データが複数のポインタにより多重参照されることになり共有化される。これにより、参照しているポインタのアドレスが一致しているかどうかにより容易にUnification の成功／失敗を判別することが一部可能となる。また、この方法ではメモリスペースは多量に必要とされるが、SHM に格納して行く時に同一の内容が存在するかどうかのチェックは不要であり、且つ同一データへのポインタによる参照が分散化でき多数台IHからのメモリアクセス競合の集中化を緩和できるという利点がある。

5．引数間のLazy Unification

SHM に格納されている構造体データの参照を必要とするUnification は、SHM に対してリスト読出し要求又はベクタ読出し要求がIH内のUnification Unitより送られ、UU内のバッファメモリに構造体データが取り込まれてUnificationが実行される。この時、引数間のUnification は並列には行わず、SHM を参照する引数がreducible goal又は選択されたclauseのどちらか一方に存在し、且つ他方が変数又はAtomでない場合には、その引数間のUnification を後回しにし、その他のUnification を優先して実行する様にし、それらが全て成功した場合に限りSHM を参照する引数間のUnification を開始することにし、無用の引数間のUnification を避ける。また、リテラル内の引数間で変数が共有されている場合でも並列処理は行わないのでConsistency Check は必要とされない。

6．おわりに

ここでは静的に決まるclause中のGround Instance だけをCompile 時にSHM にInitial LoadしSHM への動的な書込みは行わない方法を述べたが、この場合SHM 中の構造体データは少なくともclauseからの参照があり一つのQuery による解が全部得られるまではガーベッジとはならないので現段階ではGarbage Collectionは行わない。今後はoccamにより記述されるソフトウェアシミュレータを開発し、それによりアーキテクチャ上の評価を進める予定である。

最後に、日頃ご指導をいただく村上第１研究室長はじめ並列推論マシングループ諸氏に感謝の意を表する。

〈参考文献〉

[1] 伊藤他，“データフロー方式の並列PROLOGマシン”，Logic Programming Conference '83, Tokyo (1983.3).

[2] 平田他，“高並列推論エンジンPIE における構造データの効率的な処理方式について”，信学技報 EC83-38 (1983.12).

[3] 尾内他，“並列推論マシンPIM-R のアーキテクチャ”，本大会予稿集 6C-6 .

[4] 尾内他，“並列推論マシンPIM-R のソフトウェアシミュレーション”，本大会予稿集 6C-9 .

[5] 相田他，“並列推論エンジンPIE の階層的構成法”，情報処理第29回全国大会 28-4 (1984.9).

関係データベースマシン“Delta”における関係代数演算の最適化

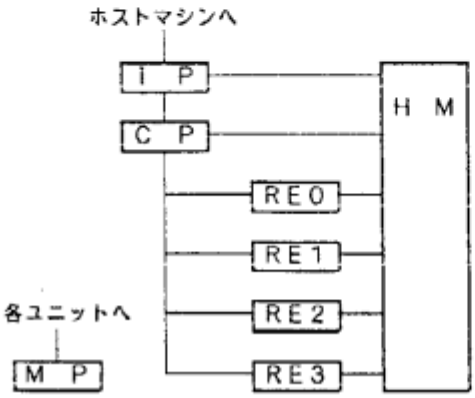
伊藤　文英　　星野　康夫　　（株式会社東芝　青梅工場）

1．はじめに

関係データベースマシン“Delta”は通産省第5世代コンピュータプロジェクトの一環として開発された。[1] Deltaはホストマシンである推論マシンと接続され，外部データベースを提供する。

Deltaの構成を図1に示す。

図1　Deltaの構成



インタフェースプロセッサ（IP）はホストマシンとの通信を制御する。

コントロールプロセッサ（CP）はホストマシンからのデータベース問合せを解析し，演算処理を制御する。

関係データベースエンジン（RE）は関係代数演算を専用のハードウェアで高速に処理する。

階層構造メモリ（HM）は大量の関係データを格納する。

メンテナンスプロセッサ（MP）はDelta全体の保守，運用を制御する。

本報告では，ホストマシンからの関係代数にもとづくデータベース問合せに対して，CPでおこなう演算処理制御の方針について述べる。

2．Deltaの概要

2．1　関係代数演算の種類

Deltaが提供する関係代数演算には次のものがある。

(1) 射影演算

関係の属性構成を変えるものであり，不要な属性の削除および属性順の変更を指定できる。

(2) 制約演算

関係から指定された制約条件をみたすタプルのみを選択するものであり，制約条件は，属性と定数との比較条件，属性同士の比較条件，属性と定数の間での四則演算の結果，および以上の比較条件をコンジャンクティブノーマルフォームで結んだものが許される。比較演算子は6種類（＝，≠，＞，≧，＜，≦）ある。

(3) 結合演算

2つの関係のあいだで属性同士の比較条件をみたすタプルの組をもとめるものであり，θ－結合，自然な結合，準結合の3つがある。

θ－結合は2つの関係の間で，属性同士が指定された比較条件をみたすタプルの組合わせからなる新しい関係を求めるものであり，比較演算子は6種類（＝，≠，＞，≧，＜，≦）ある。自然な結合はθ－結合で比較演算子が等号（＝）の場合とほぼ同じであり，結合の結果として作られる関係の属性構成のみが異なる。準結合は一方の関係のタプルのうち，ある属性の値が他方の関係のある属性に含まれているようなもののみを選ぶ演算であり，他の関係を条件とした制約演算と考えることができる。

(4) 集合系演算

2つの関係のタプルの間で集合演算をおこなうものであり，和集合，共通部分集合，差集合，全積を求めるものがある。

2．2　演算処理の特徴

Deltaにおける関係代数演算の処理は次のような特徴をもっている。[2][3]

(1) 関係をタプル方向ではなく属性方向のデータの集まりとしてもつことを原則とする。各データにはタプル方向のつながりを示すためのタプル識別番号(TID)がつけられる。

(2) パーマネントデータとして保持される属性方向のデータは，値およびTIDにより2段階にクラスタ化される。HMに対するパーマネントデータの要求では，値またはTIDを条件として指定することにより，演算前にデータをある程度絞り込むことができる。

(3) 演算は，その演算に必要な属性のデータのみを用いておこなわれる。従って，HMに対するパーマネントデータの要求は，その属性が演算対象となるか，または最終結果としてとして必要な場合に，はじめておこなわれる。

(4) 集合演算など，タプル方向につながったデータが必要な場合は，各属性のデータをTIDでソートし，属性方向からタプル方向へデータ形式を変換する。逆に，属性方向のデータを扱う演算において，タプル方向につながったデータしか存在しない場合は，タプル方向から属性方向へデータ形式を変換する。

3．演算処理方式

3．1　射影演算

関係が属性方向のデータの集まりの場合は，データの処理は発生せず，関係の管理情報の変更のみがおこなわれる。タプル方向につながったデータしか存在しない場合は，データ形式を属性方向に変換したあと，管理情報

の変更がおこなわれる。

3．2　制約演算および準結合演算

制約演算の場合は，制約条件に含まれる属性のデータが用意され，ＲＥにおいて制約条件をみたすデータが選択され，条件をみたすデータのTID のみが出力される。用意される属性のデータがパーマネントデータであり，かつその属性に対する比較条件が定数との比較である場合は，値についての条件つきでＨＭにデータが要求される。

準結合演算の場合も，結合条件となる２つの属性のデータが用意され，ＲＥにおいて結合条件をみたすデータが選択され，条件をみたすデータのTID のみが出力される。

制約または準結合の結果の関係のある属性のデータが必要な場合には，その属性の制約前のデータが用意され，そのデータに対してTID を条件とした制約がＲＥにおいておこなわれる。用意される属性のデータがパーマネントデータである場合は，TID についての条件つきでＨＭにデータが要求される。

3．3　θ－結合演算および自然な結合演算

結合条件に含まれる２つの関係の属性のデータが用意され，ＲＥにおいて条件をみたすデータの組合せが求められ，それらのデータのTID の組に新しいTID をつけた３つの組のTID が出力される。結合の対象となる関係が制約または他の結合の結果の関係である場合は，用意される属性のデータには，あらかじめその制約または他の結合の結果を反映させる。

結合結果の関係のある属性のデータが必要な場合には，その属性の結合前のデータが用意され，そのデータと３つの組のTID の間で，結合前のデータのTID を条件とした結合がＲＥにおいておこなわれる。用意される属性のデータがパーマネントデータである場合には，３つの組のTID のうち結合前のデータのTID についての条件つきでＨＭにデータが要求される。

θ－結合演算と自然な結合演算におけるデータ処理は同じであり，関係の管理情報の変更処理が異なる。

3．4　集合演算および最終結果の作成

いずれもタプル方向につながったデータが用意される。集合演算の場合は，２つのタプル方向につながったデータの間での演算がＲＥにおいておこなわれる。最終結果の場合は，その関係はＩＰ経由でホストマシンへ転送される。

タプル方向につながったデータは，全ての制約，結合が反映された各属性のデータを作り，TID でソートし，タプル方向へ変換することにより作られる。すべての制約，結合は次のようにして反映させる。

制約または準結合のみがおこなわれた場合は，次のようにして結果を反映させる。

① 対象となる属性のデータがパーマネントデータである場合は，TID についての条件つきでＨＭにデータが用意される。

② そのデータに対して，ＲＥにおいてTID を条件とした制約がおこなわれる。

θ－結合または自然な結合がおこなわれた場合は，次のようにして結果を反映させる。

① 最後のθ－結合または自然な結合のあとでおこなわれた制約または準結合がある場合は，ＲＥにおいて，その制約または準結合の結果を，最後のθ－結合または自然な結合の結果の３つの組のTID に反映させる。

② 複数のθ－結合または自然な結合がおこなわれた場合は，ＲＥにおいて，それらの結果の３つの組のTID の間で，TID を条件とした結合をおこない，最初のθ－結合または自然な結合の対象となった属性のデータのTID と，最後のθ－結合または自然な結合で新しくつけたTID との間の対応を求める。

③ 対象となる属性のデータがパーマネントデータである場合は，TID の対応のうち対象となる属性のデータのTID についての条件つきでＨＭにデータが要求される。

④ 対象となるデータとTID の対応との間で，ＲＥにおいて対象となるデータのTID を条件とした結合がおこなわれる。

制約も結合もおこなわれなかった場合で，対象となる属性のデータがパーマネントデータである場合は，ＨＭに対し条件指定なしでデータが要求される。

4．　まとめ

以上の処理方式により，Delta における関係代数演算処理は次のような利点をもつ。

(1) 演算対象とはならず，かつ，最終結果としても不要な属性のデータは，演算処理に全く影響を与えない。

(2) 射影演算は，ホストマシンからの演算処理指定の並びのどこに置かれていてもよい。

(3) 制約演算および準結合演算は，各属性のデータに対して独立に行ない，あとでTID による共通部分集合を求めればよいので，演算データ量が少なくなりＲＥにおける専用ハードウェアによる高速処理にむく。

(4) 演算によりデータ量が増加するθ－結合および自然な結合は，可能な限りあとに回される。

(5) 複数のθ－結合および自然な結合の結果を，TID 同士の演算により１つの結合演算の結果にまとめてから各属性のデータに反映させるので，やはり演算データ量が少なくなり，ＲＥにおける処理にむく。

(6) タプル方向につながったデータまたは最終結果の関係を求める処理は，各属性に対して独立におこなえるので，複数ＲＥによる並列処理にむく。

謝辞

本開発に関し，御指導賜いたＩＣＯＴ第１研究室知識ベースマシングループの方々に深謝いたします。

参考文献

[1] 角田，柴山，横田他「RDBM Delta（Ⅰ）～（Ⅲ）」情報処理学会第26回全国大会講演論文集，4F-6～8,1983

[2] Shibayama 他「Relational Database Processing on an Attribute-Based Schema」情報処理学会第29回全国大会講演論文集，3F-3,1984

[3] 伊藤，星野他「関係データベースエンジンの開発（その５～６）」情報処理学会第29回全国大会講演論文集，4F-9～10 ,1984

並列推論処理システム

7c-8

――改良型節単位処理方式――

久門耕一　板敷晃弘　佐藤健　増沢秀穂　相馬行雄

（富士通株式会社）

1　はじめに

これを改良し、

我々は、先に"節単位処理"と呼ぶ並列推論方式を提案しシミュレーション等により検討して来たが、その経験を基に、今回"枝分け方式"と呼ぶ、逐次実行を基本とした新しい並列推論処理方式を提案する。

2　節単位処理方式の評価

５８年度には、並列推論向けに「節単位処理」と呼ぶ方式を提案し、シミュレータ及び小規模実験機によって、以下の問題点を確認した。

①要素プロセッサの基本処理方式に横型探索を採用しているので

- リテラルの処理毎に実行環境を入れかえるメモリアクセスが多い
- 処理の単位が小さい

ソフトウェアシミュレーションの結果では、メモリアクセス時間（ゴールプールおよび処理の管理表）が総実行時間の６０％を占めていた。

②仕事を分け与える時に、受け手の状態を考えていないので

- Bysy状態のPEが他のPEからゴールの処理を依頼されると、一旦自分の処理を中断してその転送されてきたゴールを受け取る必要がある。
- Idle状態のPEの存在が分からない為、システムに仕事がPE台数以上存在していても、仕事がもらえないIdleなPEが出る可能性がある。

3　枝分け方式

以上のような点を考慮し、逐次型処理系を基本とした並列処理系を考えた。

以下では、PROLOGの実行過程を表わすSearch-tree に基づいて、本方式を説明する。

3.1 Search-tree

Search-tree とはPROLOGの実行の様子を表す木で、各アークがOR分岐を表し、各ノードがその時点で解くべきゴールを表す。（図１）

一般の逐次型PROLOG処理系ではこれをBacktrack を用いて、Depth-first に解くのである。

3.2 Search-tree の分割

枝分け方式では、このSearch-tree を空きPEからの要求がある毎にSub-treeに分割して分け与える。その空きプロセッサは受け取ったSub-treeをDepth-first で解くものである。この際、通信回数をへらす為に、　なるべく大きなSub-treeを分割することが、重要である。

従って、出来る限り上方でtreeを分割して、Sub-treeを取り出す事により、大きなSub-treeが得られる様にした。図１では、PE0 が(1)から推論を実行して(3)まで処理が進んだ時に、PE1 から要求が来た為、一番上方に近い分岐可能なノードである(1)から分け与える事を示している。即ち、PE0 は、未だ実行していないSub-treeである、(1)の右側の２つの枝、(5)と(8)からなるSub-treeをPE1 に分け与える。

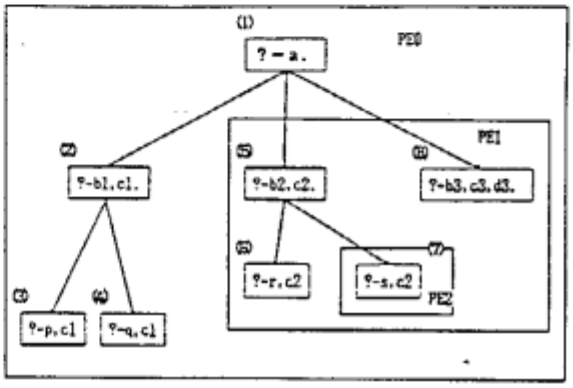


図１　Search-tree の分割

4　枝分け方式の特徴

PE単体の性能

①分割する為に必要な処理時間が少ない

→逐次処理を基本とし、他から要求があった時に初めてタスクを分割する。　これにより、他から要求がないのに実行中にタスクの分割をしておくといったオーバヘッドを無くすことが出来る。

②処理最中のプロセス切り換えが少ない

→他から要求が無い間は、逐次に実行して行くことにより、不必要なプロセスの切り換えをせずにすむ。

分配される仕事について

③分割された個々の仕事が大きい

→他から要求のあった時には、最もトップレベルに近い仕事を分け与える事により、分割する仕事の大きさを従来よりかなり大きく出来る。

㉔暇なプロセッサに仕事を与える。
→暇なプロセッサがいれば自分の仕事を分割して分け与えることにする。従って新たな仕事は必ず暇なプロセッサに分け与えられる。
㉕仕事の数はプロセッサ台数以下である。
→プロセッサ数よりも多くの仕事を生成しないので、不必要なメモリスペースがいらない。

PE間の通信について

㉖仕事の送受は暇なプロセッサがいる時だけにする。
→全てのプロセッサが稼働している時には、プロセッサ間の通信が行われない。
㉗仕事の要求は仕事を持っているプロセッサにする。
→不必要に処理の中断や分割の為の処理を引き起こさない。

5. 実現法

以上述べた様な特徴をもつ様分け方式による並列推論処理系を実現する為に次の様な方法を採った。

5.1 Sub-treeの切り出し方

Sub-treeの切り出し方は、2通りの方法が考えられる。分岐出来るノードの、

①或る枝より右側総ての枝を暇なPEに送る。（図1）
㉘或る枝だけを暇なPEに送る。（図2）

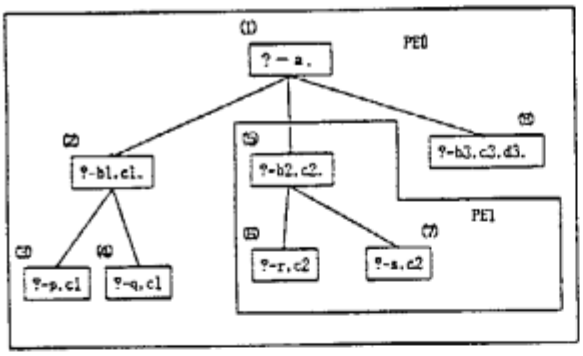


図2　別の戦略によるSearch-tree の分割

①の方法では、分割される仕事が㉘より大きい。図1の場合、述語aに関する3つの定義内の2番目からの実行が必要になる。単一化を行わずに分割されたサブゴールが作れるので処理の分散がより速く行われる。

㉘では、分割される仕事が小さい。転送されるサブゴールは、図2では、?-b2,c2.であり、通常人間が入力するqueryと同じ形をしている。

このゴールを生成する為に、PE0は、現在、(1)で自分の実行している定義の次の選択であるa:-b2,c2. の頭部と(1)との単一化を実行する必要がある。しかも、分割を行う為には、この頭部の単一化が成功するまで、aの定義の頭部と単一化を行わなければならない。つまり、忙しいPE0は暇なPE1に仕事を与える為に、更に働かなければならない。

本方式では、処理単位の大きく、分割の為に必要な処理の少ない①の分割法をとった。

この方法を実現する為に、5.4で説明する新しい述語を作成した。

5.2 変数束縛の一時解除

treeを上方で分割する時、分割されたSub-treeに含まれている変数の値が、分割を行うノードを実行した時点よりも後で束縛された場合には、その変数値を、一時的に未定義状態に戻す事(undo)が必要がある。

この為、各変数に何時束縛されたかを記録しておく領域を設け、Sub-treeを分割する際に、既に束縛されていたかどうかを判定出来る様にした。この方法によれば、分岐時にゴールの中に存在する変数の数に比例した時間でundoが可能となる。

これを、Trail stack を用いて行う事も出来るが、Trail stack を用いると、undoするのに分岐時点から現在迄に束縛された変数の量に比例した時間がかかる。

5.3 転送するデータの形式

Sub-treeを分割し転送する場合には、その時点のサブゴールの形で転送する。図1の場合、PE1 が実行するゴールは述語a（の一部）なので、次に述べるrno とともにa. と転送される。

5.4 ルール番号の導入

図1から分かる様にPE1 からの要求によりtreeを分割する場合、PE0 が述語aの第一番目の定義を実行しているので、PE1 はAの定義の第2番目以降だけを実行する必要がある。その為に、直後に続く述語の定義の任意番目から実行させる事の出来る rno(数) という特殊な述語を作った。

これを用いることにより、PE0 がPE1 に転送するゴールは ?-rno(2),a. という形になる。

6　今後の課題

様分け方式は、Sub-treeの分割方法の自由度が高い為、積極的に分割に関する制御を行うことにより、処理の高速化を図ることが出来る。また、逐次型処理系の持つ制御構造を持たせる事も可能である。しかし、並列処理系に、どのような制御構造を持たせるかは、言語との関係で考慮しなければならない。また、転送されるゴールのデータ構造も転送量、並びに処理速度の点から今後検討しなければならないと、考えている。

なお本研究は、第5世代計算機プロジェクトの一環として、ＩＣＯＴの委託で行ったものである。

参考文献

(1)佐藤ほか「節単位処理モデルの提案」
情報処理第28全国大会5H-8 pp.1141
(2)板敷ほか「PROLOGの並列処理システム」
情報処理第29全国大会7B-4 pp. 249
(3)板敷ほか「並列推論処理システム ―改良型節単位処理方式の実験 ―」 情報処理第30全国大会7C-7

関係データベースエンジンの基本演算の性能検討

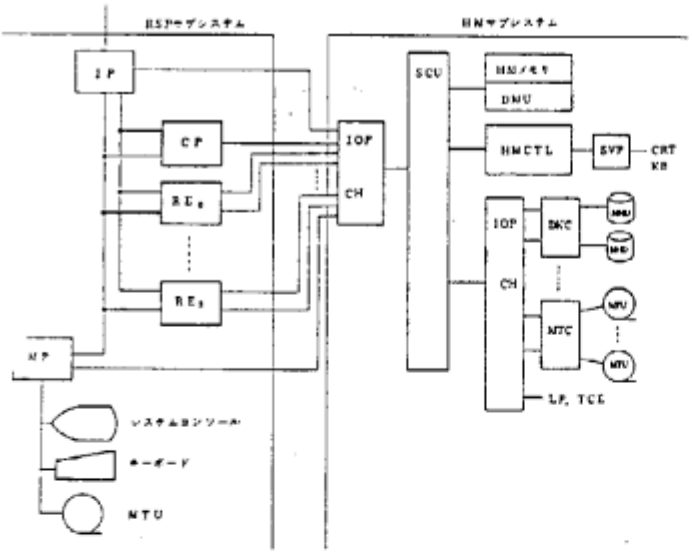
酒井 浩　　岩田和秀　　安部公朗　　柴山茂樹　　村上国男
（株）東芝　総合研究所　　新世代コンピュータ技術開発機構

1　はじめに

著者らは，通産省第５世代プロジェクトの一環として，関係データベースマシンＤｅｌｔａの開発に携わっている。

Ｄｅｌｔａは，図１のように機能分散型システムであり，フロントエンドプロセサの役割を担うインターフェース・プロセサ（ＩＰ），コマンド解析やデータベースの制御を司どるコントロール・プロセサ（ＣＰ），関係代数演算を高速処理する関係データベース・エンジン（ＲＥ）４台，データベース・データを格納する階層構造メモリ部（ＨＭ）サービス・プロセサの役割を果たすメインテナンス・プロセサからなる。

本稿では，ＲＥが処理する各種演算のうち，エンジン・コアという専用ハードウェアだけで処理できる演算（以下，基本演算と呼ぶ）についての性能検討の結果を報告する。なお，基本演算は，ソート（並べかえ），ジョイン（結合），セレクション（選択）など重要な演算を含む。



2　処理概要と性能測定の方法

ここでは，基本演算に伴うＲＥとＨＭの処理概要と性能測定の方法について述べる。

2．1　ＲＥによる基本演算の処理

ＲＥの基本演算は，ＣＰからの指示に基づいて行われる。その概要は次のとおりである。

①ＣＰからＧＰＩＢ経由で送られてきた関係代数演算用のコマンドを解析し，ＨＭにＨＭアダプタを介して，演算対象となるデータの送受信を要求するコマンドを送る。

②ＨＭから送られてきたデータに対して，エンジン・コアという専用ハードウェアで関係代数演算を施し，結果を再びＨＭに送る。

③演算結果として生成したアイテム数などをレスポンスとして，ＧＰＩＢ経由でＣＰに送る。

2．2　ＨＭ内部の処理

ＨＭでは，演算対象となるデータ（ストリーム）ごとに異なるタスクを対応させている。従って，例えばジョイン演算では，３つのタスクが起動される。あるデータ転送を行なうためにＨＭが行う処理は，次のとおりである。

①ＲＥからのアテンションを契機として，タスクの起動し，ＲＥからの要求を受取るためのチャネル・プログラムを実行する。

②ＨＭ→ＲＥのデータ転送では，ＨＭの仮想記憶装置上に格納されているデータをＲＥに転送するためのチャネル・プログラムを作成・実行する。

ＨＭ←ＲＥのデータ転送では，ＲＥから転送されてくるデータの格納領域を確保して，データをそこに格納するためのチャネル・プログラムを作成・実行する。

③データ転送が終了すると，レスポンスを作成し，ＲＥに転送するためのチャネル・プログラムを実行する。

なお，データ量が大きい場合には②の処理が繰返される。

2．3　測定方法

ＲＥの基本演算の性能測定の方法は次のとおりである。

システム構成：　ＨＭとＲＥ１台だけを使用する。ＣＰやＩＰは使用しない。

演算対象：　演算の対象となるデータは，ＲＥで作成してＨＭの半導体メモリ上に格納する。アイテムの長さは，すべて１６バイトとする。

時間の測定：　ＲＥによるコマンドの解釈からレスポンスの作成までの所要時間を，ＲＥの計時モジュールを用いて測定する。また，ＲＥもしくはＨＭで認識できる重要なイベントについては，その発生時刻もあわせて記録し，それも利用する。

エンジン・コアによる処理の計算値：　ＲＥの専用ハードウェアであるエンジン・コアが処理に要する時間は，ほぼ正確に見積ることができる[3]。そこで，これと実測値を比較して，予想通りの性能が得られているかどうか判定する。

測定する演算の種類：　ＲＥでの演算は入力データの数が１つのものと２つのものに大別できるので，それぞれを代表するソート演算とジョイン演算について行う。

3　結果

ソート演算とジョイン演算について所要時間を測定し，表１，表２の結果を得た。また，ＲＥとＨＭの主要処理の所要時間を表３に示す。

データの件数		計算値	実測値
入　力	出　力	相対値	相対値
1	1	1	30
4095	4095	43	75
8190	8190	107	137

表１　ソート演算の所要時間

データの件数			計算値	実測値
入力1	入力2	出　力	相対値	相対値
1	1	1	1	48
4095	4095	1	85	108
4095	4095	4095	107	133
4095	65520	4095	747	830

表２　ジョイン演算の所要時間

レスポンス作成	ＲＥ主体	1
コマンド解釈	ＲＥ主体	3
アテンションからコマンド読取りまで	ＨＭ主体	2
４ＫＢのデータの転送と完了報告	ＨＭ主体	9

表３　ＲＥとＨＭの主要処理の所要時間

4　考察

4．1　基本演算の性能

表１と表２から，ＲＥとＨＭは基本演算を期待どおりの時間で処理することがわかった。

表２のジョイン演算では，データの量がある程度大きい時の方が，ごく小さい時よりオーバヘッドが小さくなっている。これは２つの入力データに関する処理がＲＥとＨＭの内部で多重化されるため考えられる。

4．2　ＲＥ制御プログラム

基本演算の場合，ＲＥのマイクロプロセサは，ＨＭアダプタとエンジン・コアの制御を主に行っている。ジョイン演算の場合，ハードウェアの起動と割込みの回数，およびI/O 命令の実行回数は，典型的には，次のとおりである。

機器名称	起動	割込	I/O 命令
ＨＭアダプタ	48	65	570
エンジン・コア	2	2	105

これらの値は，通常の汎用コンピュータの場合と比べて格段に大きいといえる。そのため，ＲＥ制御プログラムは，ハードウェア制御の高速化に留意して開発した。その結果，ハードウェアから割込み処理を，１回当たり平均 0.3ミリ秒以下で行うことができた。これは，マイクロプロセサで制御していることを考慮すると十分高速であるといえる。

4．3　ＨＭ

表３からもわかるよに，ＨＭは高速に動作しているといえる。これはＨＭハードウェアのデータ処理能力とデータ転送能力の大きさだけでなく，ＨＭ制御プログラムもダイナミック・ステップ数を小さくする努力が払われた結果としてもたらされたものである。

なお，実際のデータ転送は，当初の予定どおり最大で３ＭＢ／秒で行われていることが，ここに述べたのとは別の方法でわかっている。

4．4　未検討の項目

今回の検討では，Ｄｅｌｔａ全体の性能に関して下記の項目が抜けている。従って，今後それらを含めた全体的な評価をする必要がある。

①ＲＥのマイクロプロセサを用いて行う演算
②ＣＰ→ＲＥの通信に要する時間
③ＨＭ内部でディスク上に格納されているリレーションを半導体メモリにステージングする時間
④アトリビュート方向に格納しているための得失
⑤ＩＰ，ＣＰでの処理時間

5　結論

関係データベースマシンＤｅｌｔａの中で関係代数演算を行なう関係データベースエンジン（ＲＥ）の基本演算の性能について検討した。この結果，ＲＥおよびＨＭが行なう種々の処理を考慮すると，ＲＥの専用ハードウェアの処理能力の大きさが十分に発揮されていることがわかった。これは，ＲＥ制御プログラムの品質，ＨＭハードウェアのデータ処理能力と転送能力，ＨＭ制御プログラムの品質といった関連するものすべてについて，性能向上を目指して改良がなされた結果である。

著者らは，今後Ｄｅｌｔａ全体について性能評価を実施したいと考えている。

参考文献

[1] 角田，柴山，横田他，「RDBM Delta（Ⅰ）～（Ⅲ）」，情報処理学会第26回全国大会予稿

[2] 酒井他，「関係データベースエンジンの開発（その１～６）」情報処理学会第29回全国大会予稿

[3] Sakai,H., et al: Design and Implementaion of the Relational Database Engine, Proceedings of the International Conference on Fifth Generation Computer Systems, 1984

ＰＳＩの性能評価　（２）

中島　浩　　三石　彰純　　瀧　和男
（三菱電機）　　（新世代コンピュータ技術開発機構）

１．はじめに

逐次型推論マシンψ(PSI) のハードウェア・アーキテクチャの良否を検討し，将来の改良のための基礎データを得るために，ハードウェアの評価を行っている。　ここでは，ハードウェア評価用のツールと，現在までに得られた評価データに基づいた　考察を述べる。

２．評価用ツール

ハードウェアの評価の対象として，マイクロ命令のパターン解析と，キャッシュ・メモリの動的特性を選んだ。　これらの項目についての評価作業を行うために，３つの評価用ツールを作成した。　以下，各ツールについて簡単に説明する。

(1)ＣＯＬＬＥＣＴ

評価データ収集用ツールであり，ψのコンソール・システム上にインプリメントされている。　データ収集のための操作は，ψの主記憶上に置かれたデータ収集用の命令列を解釈実行することにより行われる。　データ収集用命令には，ψの内部レジスタの読出し／書込み，ステップ実行などの実行制御，収集データのディスクへの書出し，１６個の評価用変数に対する演算，条件分岐，などがあり，様々な評価データを容易に収集することができる。

(2)ＭＡＰ (Micro Instruction Analizer for PSI)

マイクロ命令パターンの解析用ツールであり，特定のパターンを持つマイクロ命令の出現頻度を，静的／動的にカウントする。　マイクロ命令のパターン指定のために，任意のフィールドの抽出や抽出したフィールドの値に対する一致／大小の判定を行うことができる。　また，判定結果に対して論理演算を施すことにより，複数のフィールドにまたがった複雑なパターンを指定することができる。　動的解析は，静的解析の結果と，COLLECT によって得られるマイクロ命令アドレスのトレース結果を組み合わせて行う。

(3)ＰＭＭＳ (PSI Memory Module Simulator)

キャッシュ・メモリを中心とするメモリ・モジュールの動的特性解析用ツールであり，COLLECT によって得られる，マイクロ命令アドレス及び主記憶の論理アドレスのトレース結果を用いて，メモリ・モジュールのシミュレーションを行うことにより，その動的な特性を解析する。　解析結果として，キャッシュ・メモリのヒット率やメモリ・アクセスによるオーバヘッドについて，キャッシュ・コマンドの種類やオーバヘッドの要因なども加味した詳細なデータを得ることができる。　また，キャッシュ・メモリの容量，セット数，書込み方式，などのキャッシュ・メモリ・アーキテクチャのパラメータを任意に変更することができ，改良の方向を検討するために有用である。

３．マイクロ命令パターン解析

ψのCPU の特徴として，柔軟なアクセスが可能な高速レジスタ・ファイル(WF)の採用が挙げられる。ここでは，WFがどのように使用されているかを，マイクロ命令の関連するフィールドのパターンを解析することによって評価した。　なお，評価データとしては，ψのオペレーティング・システム (SIMPOS) の一部分であるウィンドウ・システムを約150Kstep 実行した結果を用いた。

WFのアクセス方式としては，以下に示すものがある。

(1)直接アクセス：先頭64w はマイクロ命令内のアドレスにより直接アクセスが可能。　また，先頭16w は２ポート読出しが可能。

(2)定数アクセス：末尾64W は定数領域として用いられ，マイクロ命令内のアドレスにより直接アクセスが可能（但し読出しのみ）。

(3)間接アクセス：主記憶のデータ・レジスタであるPDR/CDR や，WF専用のアドレス・レジスタであるWFAR1/WFAR2/WFCBR を用いた間接アクセスが可能。なお，PDR.CDR.WFAR1 による間接アクセスはローカル・スタックのバファリングに，WFAR2 による間接アクセスはトレール・スタックのバファリングに，それぞれ用いられる。

上記のアクセス方式の実行頻度をTable 1 に示す。表に示されるように，WFは高い頻度でアクセスさ

Table 1

Type	Source 1	Source 2	Destination
WF00~0F	19.4/10.1	100.0/29.7	40.4/15.2
WF10~3F	56.9/29.5	—	52.5/19.7
Constant	16.1/ 8.4	—	—
@PDR/CDR	1.8/ 0.9	—	0.7/ 0.3
@WFAR1	5.1/ 2.7	—	3.9/ 1.5
@WFAR2	0.3/ 0.2	—	0.2/ 0.1
@WFCBR	0.4/ 0.2	—	0.1/ 0.0
Total	100.0/51.9	100.0/29.7	100.0/37.5

a/b : a ···· WFアクセス命令中の頻度
b ···· 全命令中の頻度

れ，その大部分は直接アクセスと定数アクセスが占めている。 従って，マイクロ命令の多くのビットをWFのアドレスに割り当てたことは，正しい方針であったと言える。 但し，直接アクセス可能な領域の半数に対するアクセスが95% 以上を占めていることが明らかになっており，アドレスのビット数を若干削減できる可能性もある。

間接アクセスの大部分は，ローカル・スタックのバッファに対するものであり，バッファ・アクセスの高速化のために設けたアドレス生成機構の有効性を示唆している。 それに対し，他の領域（トレール・スタックのバッファなど）は，容量面ではWF全体の80% 以上を占めるにもかかわらず，アクセス頻度は１%以下と極めて低い。 これは，WFの容量の大幅な削減が可能であることを示唆している。

4，キャッシュ・メモリの動的特性

前述のウィンドウ・システムを約90Kstep 実行した結果を用いて，キャッシュ・メモリの動的特性を測定した。 測定結果のうち主なものをTable 2 に示す。

表から明らかなように，キャッシュ・メモリのヒット率は非常に高く，特にwrite-stack 命令（スタックの伸長時に用いる）においては顕著である。 これは，スタックの伸縮が多く，スタック・トップのローカリティが高いことを示唆している。 また，キャッシュ・メモリのミス・ヒットに起因するオーバヘッドが非常に小さい。 これは，メモリのアクセス頻度が比較的小さいために，メモリ・モジュールがビジー状態のときに，CPU が次のアクセスを行ってまたされることが少ないことを示している。 即ち，メモリ・アクセスを低減するための，WFを用いたバファリングが有効であることを示唆している。

Table 2

ヒット率	read	92.3 %
	write	95.5 %
	write-stack	99.0 %
	Total	94.0 %
オーバヘッド	read	2.05%
	write	0.04%
	write-stack	0.00%
	Total	2.09%
アクセス頻度		13.4 %

次に，Fig.1 に，キャッシュ・メモリの容量を変化させた場合に，性能にどのような影響があるかを示す。 なお，性能の指標は，キャッシュ・メモリがない場合からの実行速度の向上率を用いた。 図から，0.5Kw 付近で性能が飽和しつつあること，即ちφのキャッシュ・メモリ容量が若干オーバ・スペック気味であることが読取られる。

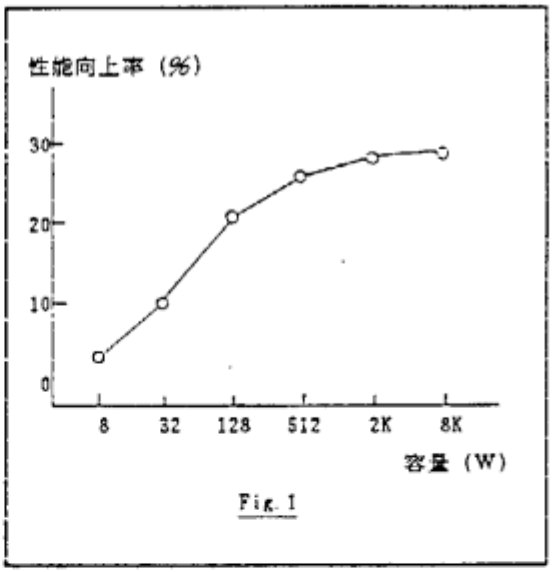


Fig.1

5，おわりに

以上述べた評価結果が，測定用のプログラムの特性に依存することは充分予想される。 特に，今回用いたプログラムはＯＳの一部分であり，一般的な論理型プログラムの特性を持っているか否かについては疑問が多い。 従って，現在他のプログラムを対象とした測定を続行しており，その結果によりさらに精度の高いデータが得られると思われる。 また，評価項目についても，さらに拡大することを予定している。

日本語校正支援用ＯＡＮＥＲＳ

4L-5　空閑 茂紀　中村 信夫　森下 太朗　大崎 幹雄　石井 暁　古川 康一

シャープ（株）　産機研究所　新世代コンピュータ技術開発機構

1．はじめに

近年、日本語の機械化が著しく進展する中で、起稿に先駆けた資料の収集・整理、文章の考案、校正などの作業は従来通り人手によらざるをえない状況が続き、合理化が遅れている。我々はＩＣＯＴ(新世代コンピュータ技術開発機構)の受託業務の一環としてＩＣＯＴで作成中の日本語校正支援システムに使用する知識ベースの作成及びその評価を行なっている。知識ベースの評価を効率的に遂行するため社内で開発したＯＡＮＥＲＳ[１],[２]を評価システムとして活用し、朝日新聞の社説を対象にテストした結果が出たので、その概要について報告する。

2．評価システムの構成

本評価システムは以下のソフトウエアモジュールから構成されている。使用したハードウェアはＶＡＸ／１１　７８０である。

2．1　ファイルの編集

本システムはＷＰで作成した、あるいは既存のデータベースの文章を入力文として校正を行うことを念頭においている。これらの入力文の形式はお互いに異なるので選択する入力文に応じて自動的に統一的な内部形式に編集する。現在使用可能な入力文は朝日新聞の記事、社内外のオフィス文例、評価文の三種の形式である。

2．2　文章抽出

文章に付けられたヘッダーの情報を利用して校正したい文章を選択する。複数の文章を一度に選択することも可能である。

2．3　単語抽出

カナ漢字変換プログラムで用いている約８万語の自立語辞書やテーブル類を漢字コードから引けるように編集し、入力文との比較により単語及び文節を抽出する。自立語検索用のインデックスは出現する見出しの統計データを利用して作成した。

2．4　エラー表示

自立語辞書との照合が取れなかった単語は校正する可能性が高いものとしてファイル化し表示する。この時、見易さを向上させるためエラー部をキーとしたＫＷＩＣを作成する。

2．5　対話的修正

ＥＭＡＣＳ風のマルチウインドウを持った日本語エディタを利用し対話的に修正を行う。未登録語の登録も行なえる辞書エディタの機能も有する。

2．6　データベース

入力文と辞書のデータベースに分かれる。

3．評価システムの処理及び評価

一般的に校正の作業の中には次のものが含まれる。

１)誤字、当て字、脱字の確認と修正
２)送り仮名、表記の確認と修正
３)常用漢字外の振り仮名の有無の確認と修正
４)忌み言葉、言い替えたほうが良い言葉の確認と修正
５)固有名詞の表記及び内容の確認と修正
６)制限字数の確認と字詰め、字埋め
７)構文上、意味上の適切さの確認と修正
８)より分かりやすい、美しい文への変換
９)文意と事実関係の確認

本システムは１)～５)を初期の開発目標としており、本稿では朝日新聞の社説記事を対象に２)の送り仮名、表記の機械の支援による校正の可能性を中心に検討した。校正の判断は「朝日新聞の用語の手びき」、常用漢字表などを基準とした。

3．1　入力文

朝日新聞記事
昭和59年7月1日～昭和59年7月10日
紙面上では翌日の朝刊に掲載

各記事はヘッダーが付いており、記事の選択が可能である。ヘッダーのキーには日付、面コード、版数、記事連番がある。

ヘッダーのキーを自由に組み合わせて目的とする記事を選択するユーティリティ・プログラムを用いて、社説のみを抽出した。

3．2　辞書

単語照合用の辞書と文節切り出し用の辞書、テーブル類からなる。

単語照合用の辞書:朝日新聞の用語の手びき
　＋当社カナ漢字変換用８万語自立語辞書

文節切り出し用辞書:付属語、接辞、活用語尾テーブル、接続行列テーブル

「朝日新聞の用語の手びき」は朝日新聞の記者が誰にでも分かる平易な文章で記事を書くためのよりどころにしているもので、文章で記述された部分を除いて使用した。その結果は次の通りである。

NO	項　　目	件　数
1	表記の基準	------
2	音訓引き漢字表	9,904
3	現代仮名遣い要領	------
4	送り仮名の付け方	8,010
5	用字用語	15,430
6	誤りやすい慣用句	820
7	外来語の書き方	1,277
8	外国地名の書き方	915

3．3　校正候補の抽出

本評価システムは入力された漢字仮名交りの文章を上記２種類の辞書と照合し、文節単位に切り出す方法を取っている。文節を切り出すために次のアルゴリズムを用いた。

１)字種による切断

２)最長一致、頻度などによる切断

これらの文節切り出しアルゴリズムを通じ、次の２つの校正の可能性の高い文節が抽出される。

１)未登録語

２)解析エラー

1)は接続条件を満たすが単語照合で失敗した文節であり、２)は接続条件を満たさない文節である。

3．4　結果及び評価

現在、3．1～3．3を用いた実験を繰り返しているところであり、一例として朝日新聞７月２日朝刊の社説の結果の一部を図１に示す。

上段は評価した原文、中段はシステムにより文節切断した結果、下段はシステムが判断した校正の可能性の高い候補文節のＫＷＩＣであり、u:,e:はそれぞれ次のキーワードが未登録語、解析エラーであることを表している。

―昨年、中国、韓国から抗議を受けて、改定を～ どのような記述の変更がおこなわれたかであろう。 これまでより検定がゆるやかになったとは～ 妥当かどうか最検討すべきだと思う。
―▲昨年、／中国、／韓国から／抗議を／受けて どのような／記述の／変更が／お／こな／われ これまでより／検定が／ゆる／やかになったとは 妥当かどうか／最検討す／べ／き／だと／思う。
中国　u:韓国　から抗議を受けて 変更が　e:お　こなわれたかであろ おこな　e:わ　れたかであろう。 検定が　e:ゆる　やかになったとは 最検討す　u:べ　きだと思う。 焦点は　u:アルゼンチン　の動向だ 経営に　u:ヒビ　が入りはせぬか

図１朝日新聞社説記事の評価結果の例

図１に示すように校正評価プログムを通して抽出された言葉、文節には次のものが含まれる。

１)未登録語・・・韓国、アルゼンチンなど

２)常用漢字では一般に漢字表記がとられところをかな(カナ)表記にした言葉

・・・おこなわれる、ゆるやかになど

3)文字コードの間違い。

・・・検討すべきのべはカタカナコード

２)は分かりやすい文章との兼合いがあるものと思われる。３)は記事は正しいのでコード変換上の誤りと思われる。今後評価データを増やし詳細な校正システムの効果について検討していく予定である。

4．おわりに

日本語校正システム用の知識ベースは、人間では気付かなかったり、見落としがちな間違いを抽出するのに効果のあることが分かった。

本作業を行うにあたりＩＣＯＴに新聞記事の提供、「朝日新聞の用語の手びき」の使用を許諾して戴いた朝日新聞社に感謝します。

[参考文献]

[1]森下他 "OANERSの日本語パーザ"
情報処理学会第30回全国大会 6K-6 1985

[2]中村他 "OANERSの質問応答機構"
情報処理学会第30回全国大会 4L-6 1985

[3]石井 "新聞における校正・校閲の実データによる調査"ICOTテクニカルレポートTR－039 1984

第五世代コンピュータ・プロジェクトと TELLプロジェクト

横井 俊夫　黒川 利明　渕 一博
（財）新世代コンピュータ技術開発機構

1．はじめに

第五世代コンピュータは、新しい機能を提供し、そのための新しい仕組・構造を持つコンピュータである[1]。新しい機能は知識情報処理という言葉で表される。この新しい機能を効率良くしかも整った構造のシステムとして実現するためには、コンピュータ自身がソフトウェア面、アーキテクチャ面とも整った仕組・構造を持ち、整った構造のソフトウェアを作成しうる開発環境を持たねばならない。第五世代コンピュータ・プロジェクトの中で知的プログラミング・システムと称される研究開発課題は、この開発環境を作り出すこと、と同時に新しい機能である知識情報処理の代表例ともなる、非常に重要な課題である。

この知的プログラミング・システムへの取組みとして、様々な方向、様々なレベルからのアプローチが行われているが、その中の大きな柱として東工大（横本教授）のTELLプロジェクトへの協力という活動がある。この協力形態は、プロジェクト前期（57年～59年度）においてはまだ予備的なものであり、ICOT側の体制も充分とはいえないが、中期（60年～63年度）からは、本格的なものとして進める予定である。

TELLプロジェクトについては、すでに詳しく説明されているので、ここでは第五世代プロジェクトにおける知的プログラミング・システムへの取組みの基本的な考え方を仕様記述の観点から説明する。この基本的な考えと、TELLプロジェクトの考え方とが互いの軌を一にするものであることが、両プロジェクトを近付けた要因である。

2．知識情報処理システムへの指針

まず、すべてに先立って、第五世代コンピュータにおける知識情報処理システムの有り方、指針を述べる。大きく3つからなる。(a) 問題に対する適切な記述・表現のための言語の設定：対象となる問題領域の知識を扱うためには、それに適した知識の記述や表現のための言語を上手に設定することが、まず肝要である。知識処理のためには、問題領域の指定だけでは不十分で、その領域の知識をどのようなものとして見るか、どのレベルで扱うかを明確にしてはじめて議論がはじまる。この知識のとらえ方、レベルを明示するのが、言語の設定である。(b) 知識間の対応付けの容易性：言語を用いて知識の内部構造が表現される。知識処理は、2つの知識が同一であるか否かを判定したり、同一になる条件を見つけ出したりする対応づけの操作が基本となる。あまりにも掛け離れた構造を持つ知識間での複雑な操作は、まだまだ機械化は不可能である。簡単な操作、すなわち構造が非常に類似していることが重要である。問題解決の立場からいえば、解決のための一つ一つのステップが小さいこと、少なくとも機械が行うステップは小さく、大きな解決手順の流れに対する対応づけは人間側にまかせることを意味する。(c) 簡明で柔軟なシステム構成：システム全体としての知識ベース、データベースを持つ対話型のシステムとして構成される。(b) の指針にそった人間－機械系としての知識情報処理システムは、柔軟な対話を通じてはじめて有効に働く。この対話を効率良く、簡明なものとするのが知識ベース、データベースの存在である。システム全体を見通しの良い整理されたものとするのもこの対話と知識ベースの機構である。この時留意すべきことは、この機構も、やはり対象とする問題領域の特殊性を十分に取り入れて作らねばならないということである。

3．ソフトウェアの問題への取組みの指針

ソフトウェア（プログラム）の作成、保守、運用を問題領域とした場合、2．で述べた指針がどう具体化されるかを述べる。

(a) 言語の設定：プログラムという知識を表現・記述するために必要な言語として、大きく3種類のものが考えられる。①人間に理解しやすく、広く使われており、しかも表現上の高いユニバーサリティを持つ言語、②理論的に厳密な表現が可能な言語、③マシン上で効率良く実行可能な言語、の3種類である。この3つの性質をすべて兼ね備えた一つの言語という設定は、先の将来の課題であり、ここでは3種の別々の言語を設定するところからはじめる。①に対しては自然言語（日本語、英語）である。当然、簡単な図、表や論理式や数式その他が表現力を増すために混在することは大前提である。人間は自然言語で考えるといわれる程、いかなる問題領域に対しても知識表現のある枠組が自然言語によって定められる。②に対しては、形式的言語である。述語論理や時間論理等形式論理による言語や形式的仕様記述言語といわれているものなどが対応する。形式的仕様については様々の議論があるが、ヨーロッパを中心に着実な進展が見られ、またGougen等のOBJ2[2]やEQLOG[3]等も注目に値する。③に対してはプログラム言語で

ある。論理型言語を母体に、関数型、オブジェクト指向型等の特徴を組上げたものを想定する。この想定のもとに多数の具体的なプログラム言語が設計されて実装される。ESP[4]という現実的なものから、KL-1[5]、QUTE[6]等それぞれの特色を持ったものが実験対象となる。プログラムの対象分野を限定することにより、知識の把握がより具体化、精密化される。例えば、プロセス・コントロール、交換機用プログラム、スプレッドシート等が考えられる。これらに対しては、それぞれの特徴を上述の3つの言語に盛り込むことになる。

(b) 対応付けの容易性：ある基準に沿って交換する、2つの表現が等価であること等を証明する、与えられたある表現と同じ別の表現を合成する、これらが代表的な操作である。自然言語で表現されたものをn、形式的言語で表現されたものをf、プログラム言語で表現されたものをpとし、X-YをXからYへの操作とすると、n-n、f-f、p-p、n-f、n-p、f-n、f-p、p-f、p-nが考えられる。n-nは、日本語仕様と英語仕様間の（機械）翻訳も含まれる。p-pには、狭義のプログラム変換が含まれる。p-f、p-n、f-n等は、難しくはプログラムや仕様の抽象化、易しくは説明の生成に対応する。このような操作が容易である、あるいは適切な部分を機械化するためには、次の2つの事が重要である。まず、すべての言語においてソフトウェア（プログラム）記述の基本的な構造が同一、あるいは類似していることである。構造としては、プログラム構造（モジュール構造、ブロック構造、クラス構造）、データ構造、制御構造等全般にわたる。次に、機械化の対象とする操作のステップを小さくすることである。大仕掛の検証、合成の自動化より、プログラム変換やプルーフ・チェッカを主にするということである。

(c) システム構成：知識（データ）ベースとしては、用語辞書、モジュール・ライブラリ、言語自身に対する知識ベース等多くのものが用意される。どの言語を用いようとも、作成、編集、変換、検証、検索を対話をしながら支援する機能が用意され、各機能を自由に使いこなすための統合的なワーク・ステーションが開発される。当然、プロトタイピングの考え方が大きな要素の一つとなる。

以上、3つの指針にそって説明をした。具体的にこれらの研究開発を進める土台となるのは、開発されつつあるSIM(PSI)[7]である。プログラム言語としてESPを取り上げ、SIMPOS[8]のプログラミング・システム[9]をシステム構成の基礎とする。また、SIMPOSのESPによる約10万行に及ぶプログラムは貴重な実験対象でもある。

また、具体的なシステム・イメージは、Literate Programming (Knuth)[10]、Inferential Programming (Scott & Sherlis)[11]、Parameterized Programming (Gougen)[12]等が参考になる。それぞれ個性に富む提案であるが、我々の基本的な考え方に素直にそうものである。

4．おわりに

ソフトウェアの問題は、様々の方向、様々のレベルからの総合的な取組みによって少しづつ解決されていくものと思われる。自然言語処理技術、定理証明・数式処理技術、エキスパート・システムの技術、高機能ワーク・ステーション技術等、第五世代プロジェクトの成果を統合しての取組みが必要である。また、第五世代プロジェクトでは、現在のソフトウェア生産活動にしばられることなく、新しい土台の上でソフトウェアの問題に取組もうとしている。しかし、得られる成果は、目前の問題にも適用できる、あるいは解決の方向付けをする等、広く活用できるものである。

謝辞：本稿をとりまとめるに際して、議論に参加された当機構の坂井公、安川秀樹の両君に感謝する。

［参考文献］

[1] 横井俊夫、“第５世代コンピューター研究開発の現状と展開”、テレビジョン学会誌、Vol. 38, No.11, 1984
[2] Futatsugi, K. et al., "Principles of OBJ2", Proc. POPL, 1985
[3] Goguen, A.J. and Meseguer, J., "Equality, Types, Modules and (Why not?) Generics for Logic Programming", J. Logic Programming, Vol.1, No.2, 1984
[4] Chikayama, T., "Unique Features of ESP", Proc. FGCS'84, 1984
[5] Furukawa, K. et al., "The Conceptual Specification of the Kernel Language Version 1", ICOT TR-054, 1984
[6] Sato, M. and Sakurai, T., "Qute : A Prolog / Lisp Type Language for Logic Programming." ICOT TR-016, 1983
[7] Uchida, S. and Yokoi, T., "Sequential Inference Machine : SIM Progress Report", Proc. FGCS'84, 1984
[8] Yokoi, T. and Uchida, S., "Sequential Inference Machine : SIM Its Programming and Operating System", Proc. FGCS'84, 1984
[9] 黒川利明、近山隆、高木茂行、坂井公、内田俊一、横井俊夫、“SIMPOSのプログラミング・システム－概要－”、情処学会第30回全国大会予稿集、1985 その他
[10] Knuth, D.E., "Literate Programming", The Computer Journal, Vol.27, No.2, 1984
[11] Scherlis, W.L. and Scott, D.S., "First Steps towards Inferential Programming", IFIP-83, 1983
[12] Goguen, J.A., "Parameterized Programming", IEEE SE-10, No.5, 1984

Prolog処理系の試作

3A-7

伊東　松幸　　向山　浩靖　　藤村　茂　　山本　仁
（シャープ株式会社）

1. はじめに

Prolog は人工知能用言語として注目されている。しかし一部の大型コンピュータ、スーパー・ミニコンを除き、パーソナル・コンピュータや小型のワークステーションでは、インタプリタ、コンパイラが揃い使用に耐え得るようなメモリ容量のある処理系は少ない。またＬＩＳＰ処理系のようにプログラミング環境の優れた使い易いProlog の処理系は極めて少ない。そこで我々は知識情報処理ワークステーションの構築に先立ち、シャープの小型ワークステーション ＯＡ－９０（ＭＣ６８０００，ＵＮＩＸ*・System Ⅲ）上にこれらを具備した処理系《Ｓ－ＰＲＯＬＯＧ》を試作したので報告する。（注＊ ＵＮＩＸは米国ベル研究所で開発されたＯＳです。）

2. Ｓ－ＰＲＯＬＯＧの概要

Prolog 処理系の外部仕様はＤＥＣ－１０Prolog に準拠した。ただしデータ・タイプとして実数・文字列・配列(heap・vector)を追加し、 atom・functor名・変数名には日本語が使用可能である。また使いやすくするために文節かな漢字変換機能付き日本語スクリーン・エディタも試作した。処理系全体としては インタプリタ(含デバッガ)・コンパイラをＵＮＩＸ*上の1つのプロセス、編集を行なうエディタを別のプロセスとして割り当て、プロセス間通信を行なう。これによりProlog 実行中にすぐにソース・ファイルのあるエディタに移れ、部分的なソースのconsult, reconsult, compile が可能であるとともに、ロギング機能でいつでも以前の実行結果を見ることが可能である。

シャープのＯＡ－９０は小型のワークステーションにもかかわらず主記憶１．７５Ｍバイト(最大４Ｍバイト)でＭＣ６８０００(１０ＭＨz)を使用しているため、メモリ容量・処理速度がミニコン並である。このワークステーション上にかなりのパフォーマンスを持ったProlog の処理系を試作することができた。

3. インタプリタ

インタプリタは structure・share 方式で、ほとんどをＣ言語でインプリメントした。処理系のメモリ・エリアはヒープ、グローバル・スタック、ローカル・スタック、トレイル・スタック、ハッシュ・テーブル等からなる。 structure・share 方式としたためグローバル／ローカルの各スタックは図１で示すような構造となっている。整数・実数は３２ビットである。

デバッガはＤＥＣ－１０Prolog に準拠した。ただしデバッガや listing等で変数名が表示できるようにした。

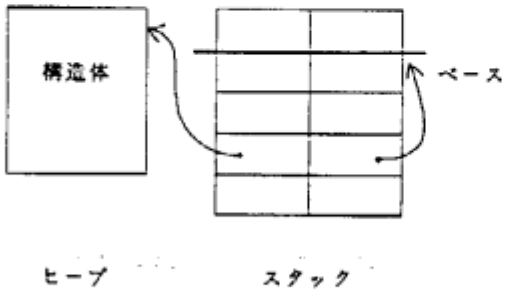


図１

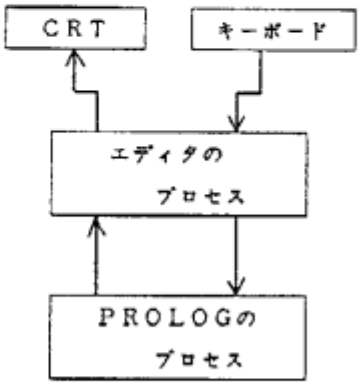


図２

4. コンパイラ

コンパイラはC言語により記述した．インプリメント方法は[1]による．コンパイラが生成するオブジェクトの主な部分はユニフィケーション部と基本制御部で、組込述語のほとんどはインタプリタのものを流用している．コンパイラは以下の順序でMC68000機械語列を生成する．

①ソース・プログラムを構文解析し内部コードに変換する．

②内部コードから[1]に相当するPLMコードを生成する．

③PLMコードを各々対応した機械語列に変換する．

コンパイラはDEC-10 Prolog の使用方法に準拠している．つまり部分的なコンパイル(incremental compile)や、インタプリタ・コンパイラ間の相互呼び出しが可能である．

表1に処理速度を示す[1]．処理系はインタプリタを中心に作成したため、また大半の組込述語はインタプリタと同様のものを使用しているので、コンパイラの処理速度はそれほど速くない．今後は組込述語の一部もコンパイル化し、高速化を図る予定である．

	reverse 30	quick sort 50
インタプリタ	470 ms	720 ms
コンパイラ	124 ms	260 ms

表1

5. エディタ

EMACS風の文節かな漢字変換機能付き日本語スクリーン・エディタを試作した．主な機能として2分割表示、マルチ・テキスト・バッファ、マクロ定義、省略入力(abbreviation)等約80のコマンドがある．なお termcapを見ているので各種の端末から利用可能である．汎用のテキスト・エディタとして使用できるが、Prologモードを設け、1つのウィンドウにProlog の実行環境を割り当てることが可能である．従って上下2つのウィンドウを同一あるいは別のテキスト・ファイルに割り当てたり、上のウィンドウをテキスト・ファイル、下のウィンドウはProlog の実行環境を割り当てたりすることが可能である．Prolog の処理系をエディタとは別のプロセスとして動作させ、プロセス間通信を行なうことにより使いやすいプログラミング環境が可能となった．図2に概略を示す．

Prolog からの出力は必ずエディタのプロセスを通りCRTに表示され、Prolog への入力は必ずエディタのプロセスを通して行なう．したがってProlog への入力はほとんどの場合エディタにより編集されてから送られるため、入力ミスによるキーの再入力が省かれる．更に2つのウィンドウで一方をテキスト・バッファ、他方をProlog にし、テキスト・バッファのウィンドウでプログラムの一部を修正して、そのプログラムを reconsult することが、簡単なエディタのコマンドで可能である．ユーザからはProlog のウィンドウはテキスト・バッファとほとんど同様に見なせ、Prolog とテキスト・バッファの両ウィンドウのカーソル移動は簡単に行なわれる．さらにProlog からの出力はエディタを通してProlog のウィンドウに表示され、それがすべて編集の対象となっているため、以前の結果をさかのぼって見ることが出来たり、編集したりすることが可能である．

6. おわりに

本研究は，第5世代コンピュータ研究開発プロジェクトの逐次型推論マシン・入出力前処理部の研究開発の一環として行ったものである．ここでは，主に，Prolog処理系とソフトウェアについて報告した．

参考文献

[1] David H. D. Warren : INPLEMENTING PROLOG , D. A. I. Research Report No.39 , No.40 (1977)

並列推論マシンＰＩＭ－Ｒの アーキテクチャ

尾内　理紀夫　　麻生　盛敏　　清水　肇　　益田　嘉直　　松本　明
（財団法人　新世代コンピュータ技術開発機構）

1．はじめに

ICOTは述語論理型言語をベースにした知識情報処理システムの研究開発を行なっている。述語論理の基本操作は推論であるから，そのハードウェアは推論マシンと呼ばれ，また，推論が基本操作であるから，このマシンは並列動作が基本となる。並列推論方式あるいは述語論理型言語として，現在いくつかのものが提案されているが，本マシンの対象言語としては，ICOTが核言語第１版のベース言語として位置づけているPrologとConcurrent Prolog [1] を選択した。並列推論方式としてはreduction 概念に基づきPrologをOR並列に，Concurrent Prolog をAND 並列に処理する方式を採用した。

次にreduction ベースのマシンを考えるに至った動機について簡単に述べる。 式 7+3のreduction を考えた時，この式は加算に関するruleを用いて自らをmodifyし，10となる。そして式10はこれ以上reduction(modify) できない（既約）ので解となる。すなわち、reduction は、self-modify とみなせる[2] 。一方，PrologあるいはConcurrent Prolog の実行過程は、親clause (goalとclause) からのresolvent の生成過程であり，resolvent として空節が導出されれば，それが解となる。これは、goalが、clause群という一種のruleを用いてみずからをmodifyする過程とみなせる。このように、PrologあるいはConcurrent Prolog プログラムの実行過程と、reduction との間に親和性の良さを見出すことができる。これがreduction 概念に基づく並列推論マシンPIM-R (Parallel Inference Machine based on Reduction concept) の研究開発の動機である。

PIM-R はstructure-copy方式を採用しているが，copyおよびそれに伴う処理量の低減およびnetwork 通過packet数とpacket 幅の低減のためreverse compaction[3]，only-reducible-goal-copy[5]，独特のprocess 構成法[5] を採用している。アーキテクチャとしては，Concurrent Prolog のための分散化共有メモリ(Message Board) の導入，効率的packet分配のためのIntelligent Network Nodeの導入，Ground instance である長い構造体データ格納のための構造体メモリ[4] の導入をその特徴としている。本稿では、PIM-R におけるPrologとConcurrent Prolog の並列実行方式，PIM-R アーキテクチャについて述べる。

2．PrologとConcurrent Prolog の並列実行方式

PIM-R では、PrologをOR並列(問題(ex. BUP等) によって高い並列度が期待できる[8]) に、Concurrent Prolog をAND 並列（ただしgoalが並列AND operatorで結合された時）に実行する。PIM-R においては，goal(body 側リテラル) が複数あった時（それら複数goal全体を親プロセスと呼ぶ．ただし，並列AND 関係にあるgoalはAND forkして別々のプロセスとなる．）は，そのうちのreducible なgoal(どれがreducible かは各種operatorにより指示される）のみをreduceし，生成されたresolvent(子プロセス）から親プロセスへポインタが張られる。このポインタにより子から親へ解が返される。すなわちPIM-R でのProlog，Concurrent Prolog の処理過程はプロセス木の伸張，縮小の過程であり，処理が終了した時，木は論理的に消滅する。

Concurrent Prolog の場合は、Prologの場合と異なり、OR関係にある子プロセスは異なる処理要素に分配されるのではなく、まず同一処理要素(IM)内に格納して，そのguard 処理をする。一方，並列AND operatorで結合されたgoalは異なるIMへ分配し並列実行する（AND 並列実行）。またConcurrent Prolog では，１つのプロセスがcommit処理に成功した時に、兄弟プロセスをkillしにはいかずに，遅れて成功した時に自殺する方法をとる。

3．PIM-R アーキテクチャ（図１）[5]

PIM-R は二種類のNodule(Inference Module とStructure Memory Nodule)とそれらを結ぶNetwork から構成される。

3.1 Inference Module (IM)（図２）

Unification UnitとProcess Pool Unit から構成される。

3.1.1 Unification Unit(UU)

(1)Clause Pool

各Clause Pool は同一のclause群を格納する。

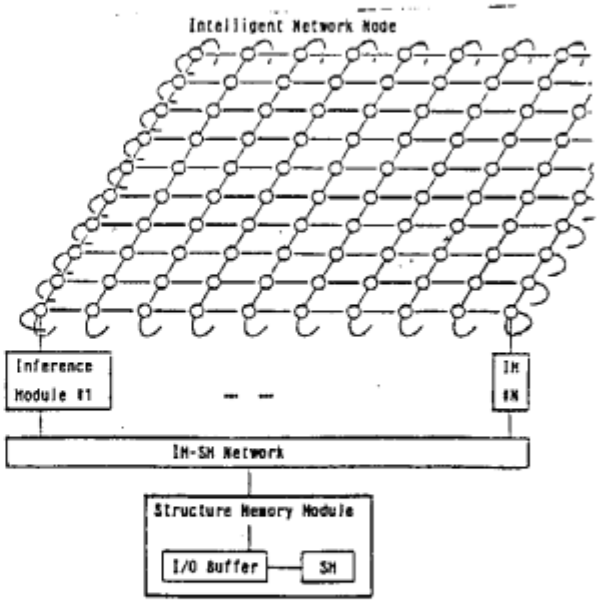


図1．PIM-R構成図

(2)Matcher

PPU から送られてきたgoalの第一引数のデータタイプによりunifiable なclause候補の絞り込みを行なう。

(3)Unifier

Matcher から送られてきたgoalとClause Pool からコピーしたClauseとの間のunification あるいは組込み述語の実行を行ない，結果をPPU へ返す。 PrologはOR並列実行するので，OR fork 数が２以上の時は新たな子プロセスを ，分配方式に従って自IH内PPU あるいは，ネットワーク経由で他IHへ分散させる。 Concurrent Prolog の場合，unification の結果は，常にいったん自分IH内PPU へ戻す。またサスペンドした場合には，将来の再unification のために，unification しようとしたclauseの格納場所，サスペンドの原因となったgoal側のチャネル等の情報を自分IH内PPへ返す。

3.1.2 Process Pool Unit(PPU)

PPU は，二種類のメモリ（Process PoolとHessage Board)と，二種類のコントローラ（Process Pool Controller とHessage Board Controller）から構成される。

(1)Process Pool Controller(PPC)

PPC は以下の各処理を行なう。

- fork処理（OR/AND fork 数の設定，OR/AND fork 失敗処理）
- 子プロセス処理（子プロセスの成功，失敗，生成，サスペンド処理および，親プロセスへの結果報告処理）
- commit処理
- reducible プロセスのUnification Unitへの転送処理
- deadプロセス処理

(2)Hessage Board Controller(HBC)

HBC は次の各処理を行なう。

- チャネルのためのセルの確保
- チャネル値の読み出し処理とチャネル値の格納処理
- サスペンドプロセスへのactivate処理

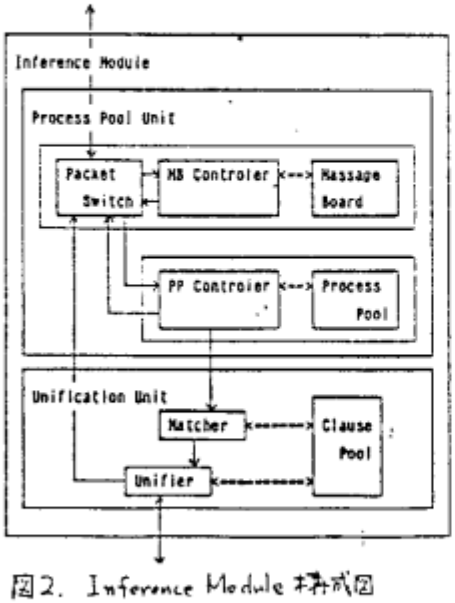


図2. Inference Module 構成図

(3)Process Pool(PP)

PPはプロセスを格納するメモリであり，１語32ビットである。structure-copy方式に起因するcopy量とIH間の通信をなるべく少なくするようなプロセス構成法[5] とプロセス内部形式[3] を採用している。reducible なプロセスはready キューに繋がれ，先頭からPPC により取り出され，UUへ送られる。また，各プロセスはreduction レベル(プロセス木の根からの深さ）を持ち，これをもとにready キュー内のプロセスを入れかえることによりIH内depth-first あるいは，breadth-first の処理を実現できる。

(4)Hessage Board(HB) [6],[7]

Concurrent Prolog におけるチャネル変数用の分散化共有メモリである。

3.2 Structure Memory Module(SMM) [4]

SMM は，Ground Instance である長いリストやベクタ等の構造体データを格納し，UUからの要求に対し，unification に必要な構造体データをUUへ転送する。

3.3 Network

IH間Network は，近傍PPU 内packet switch の入力バッファ長等の情報により子プロセスの各IHへの分配を動的に制御できるIntelligent Network Nodeを格子状に配置した構成である。IH-SMM間Network は共有バスによる実現を考えている。

4．おわりに

PrologをOR並列に，Concurrent Prolog をAND 並列に実行する並列推論マシンPIM-R アーキテクチャについて述べた。

最後に，御討論いただいた日立中央研究所　杉江，米山，坂部，岩崎各氏と，御指導いただいた村上第一研究室長に謝意する。

［参考文献］

[1] E.Y.Shapiro:A Subset of Concurrent Prolog and its Interpreter, ICOT Technical Report TR-003,1983

[2] D.A.Turner:A New Implementation Technique for Applicative L anguages,Software-Practice and Experience,No.1,Vol.9,1979

[3] 清水ﾓﾘ: 並列推論マシンPIM-R のプロセス内部表現，本情処全国大会予稿集 6C-7，1985

[4] 益田ﾓﾘ: 並列推論マシンの構造体メモリの一構成法，本情処全国大会予稿集 6C-5，1985

[5] 尾内ﾓﾘ: 並列推論マシンPIM-R アーキテクチャとソフトウェアシミュレーション，ICOT TR-77，1985

[6] 尾内，麻生: 並列推論マシンにおけるGuard と入力annotationの制御機構，第27回情処全国大会，1983

[7] 尾内，麻生: 並列環境におけるConcurrent Prolog の実現法，第29回情処全国大会，1984

[8] 尾内，清水ﾓﾘ: 逐次型Prologプログラムの解析，Logic Programming Conference 84' 東京，1984

並列推論マシンＰＩＭ－Ｒの ソフトウェアシミュレーション

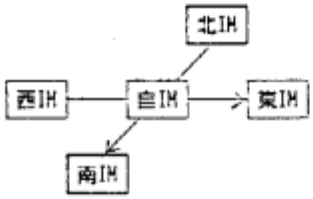
尾内　理紀夫　　麻生　盛敏　　清水　肇
（財団法人　新世代コンピュータ技術開発機構）

1．はじめに

並列推論マシンPIM-R[1]における並列処理効果，アーキテクチャ等の検証のためにソフトウェアシミュレータを開発し，テストプログラムを走行させ，各種データを収集したので報告する。なお，本シミュレータはDEC-10 Prolog/C-Prologで記述されており，その実行はDEC2060/VAX-11上で行なわれる。

2．シミュレーション条件

①ネットワーク上では転送パケットの衝突はないとする。
②各ユニットの入出力バッファは十分大きいとする。
③PPU/UUの処理時間比は通常のアセンブラ命令で記述した場合の，おおよそのステップ数よりその比を決めた。
④PrologではOR-fork 数が２以上の時に，Concurrent Prolog ではAND-fork時に，新しい子プロセスを各IM(Inference Module)に分散させるが，分散方式は自IM→東IM→南IM→自IM→…の繰り返しとする方式を採用した。（下図）



3．シミュレーション結果

3.1 Prolog プログラム

4Queens プログラム（付録1)を走行させデータを収集した。

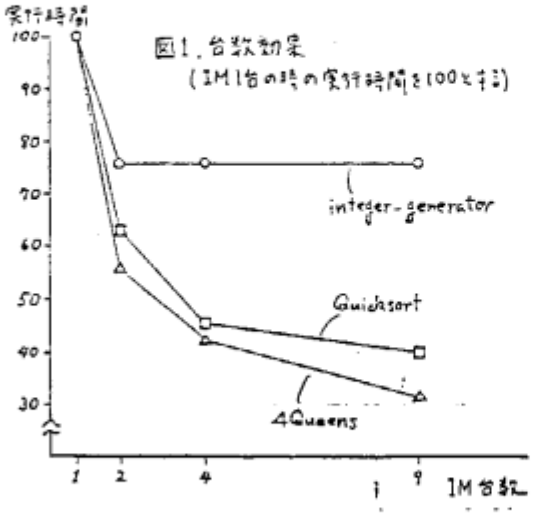


図1．台数効果
（IM1台の時の実行時間を100とする）

(1) 並列処理効果（図１）

4Queens では，IH台数増加に伴ない処理性能向上がみられ，8～ 9台頃から飽和状態に近づく。これは4Queens の動的な平均OR並列度 6.2[2] とほぼ対応する。この結果は，PIM-R がPrologプログラムの持つ並列性を引き出していることを示している。

(2) ネットワーク

ネットワーク通過パケットの種類別個数を次表１に示す。

表１

	IH 2台	IH 4台
総パケット数	162	204
true return パケット数	40	44
OR fork パケット数	61	80
fork down パケット	61	80

fork down パケットは，子プロセスがdeadになったことを親プロセスに伝えるためのものであり，これは解を求めることには直接は関係なく，garbage collectionにかかわることである。そこで，Process Poolに余裕があれば，Process Pool Controller(PPC)におけるdeadプロセスへのマーク付け処理およびfork down パケットの生成，転送処理の優先度を下げ，解を求めるために必要な処理とパケット転送を優先させることができる。これにより，ネットワーク転送パケット個数を，IH 2台 ,IH 4台の場合で，それぞれ約40% 減少させることができる。また，これにより，PPC の処理が軽くなった結果，IH 4台の場合で，1個目の解および2 個目の解が求まるまでの処理時間が，それぞれ16%,18% 短縮される。

(3) 稼動率

IH 4台の場合のPPC の稼動率は，各々46%,58%,54%,76% であり，OR fork 時の子プロセスの分配方式を固定的にした割にはバランスしている。なおIntelligent Network NodeによりPacket Switch の入力バッファ長の最も短かいIHに子プロセスを動的に分配する方式をとると，IH 4台の場合で１個目の解および２個目の解が求まるまでの処理時間をそれぞれ10%,14% 短縮することができる。この時PPC の稼働率は各々69%,72%,62%,65% とバランスしている。

3.2 Concurrent Prologプログラム

integer-generator(付録2)とQuicksort プログラム(付録3)

を走行させ各種データを収集した。

(1) 並列処理効果（図１）

もともと並列度が低いinteger-generator(プログラムからもわかるように２以下）に比べQuicksort では台数増加に伴ない性能向上がみられ，6〜7 台頃から飽和状態に達する。この例の場合の並列度は約 4であるから，これらの結果は，PIM-R がConcurrent Prolog プログラムの持つ並列性を引き出していることを示している。

(2) ネットワーク

リダクション等の回数を表２に示す。

表２

リダクション回数	284
成功回数	196
失敗回数	19
サスペンド回数	69

またQuicksort 実行中のネットワーク通過パケットの種類別個数を次表３に示す。

表３

	IH 2台	IH 4台
総パケット数	141	211
true return パケット数	17	17
AND forkパケット数	21	25
fork down パケット数	21	25
HB関連パケット数	82	140

これらより，リダクションが 284回起り，そのうち196回が成功する間にIH 2台の時で141 個，IH 4台の時で211個のパケット転送が起る。つまり，このままでも，リダクションが約 2〜 1.4回起る間に，1 個のパケット転送が起る。表３からもわかるように，Prologの場合と違ってHBに関連するパケットがIH 2台の時で58%，IH 4台の時で66% もあるから，Prologプログラムの場合のように，プロセスのdead処理とfork down パケットの生成，転送を止めてもネットワーク転送個数をIH 2台の時で12%，IH 4台の時で15% しか減少させることができない。このHBに関連するパケットは，fork down パケットと異なり，転送の優先度を下げる訳にいかない（ 下げたら解が求まらない）のでPrologの場合以上にパケット転送の高速化が重要になってくる。

(3)Hessage Board Controller(HBC) 導入効果

HBへのチャネルセルの確保は，UUにおいてユニフィケーションが成功し，新しい子プロセスがPPU に帰ってきた際に行なわれる。HBへ格納されるチャネルの個数は88であり，表２より，ユニフィケーション成功回数は 196回であるから，約 2.2回成功するたびに，一つのチャネルのためのセルをHBに確保しなければならない。よって，HB内にチャネルセルを確保する速度は，PIM-R の処理速度に影響を及ぼす。そこでHBC が導入され，HBに関する処理を担当させている。もしHBC がなく，HBに関連する処理をPPC が担ったとすると，IH 1台の場合で14%，IH 4台の場合で12% ほど処理時間が増加する。

4．おわりに

本ソフトウェアシミュレーションにより，PIM-R がProlog，Concurrent Prologに内在する並列性を引き出すこと，HBC の導入効果，Intelligent Network Nodeによる子プロセスの動的分配効果，PPC におけるdead処理とfork down パケット生成，転送処理休止の効果，Concurrent Prolog 実行時のパケット転送の高速化の必要性を確認した。現在，詳細ソフトウェアシミュレータの開発およびマイクロコンピュータによるシミュレーション専用装置の開発を行なっているが，これについては機会をあらためて報告する。最後に，御討論いただいた日立中央研究所　杉江，米山，坂部，岩崎各氏と，御指導いただいた村上第一研究室長に謝辞する。

[参考文献]

[1] 尾内ﾓﾝ：並列推論マシンPIM-R のアーキテクチャ，本情処全国大会予稿集　6C-6　，1985

[2] 尾内，清水ﾓﾝ：逐次型Prologプログラムの解析，Logic Programming Conference 84'　東京，1984

(付録１)　4Queens プログラム

```
go:-queens([1,2,3,4],X).
queens([],Y,Y).
queens(X,Y,Z) :-
        select(U,X,V),safe(U,Y,1),queens(V,[U | Y],Z).
select(X,[X | Y],Y).
select(X1,[X | Y],[X | Z]):-select(X1,Y,Z).
safe(U,[],_) .
safe(U,[P | Q],N) :-
        nodiag(U,P,N),M is N+1, safe(U,Q,M).
nodiag(U,P,N):-
        I1 is P+N,I2 is P-N,I1 =\= U,I2 =\= U.
```

(付録2)integer-generator プログラム

```
go:-true | integer(0,X),outstream(X?).
integer(X,[X | N]):-Y is X+1 | integer(Y,N).
outstream([X | N]):-display(X) | outstream(N?).
```

(付録3)Quicksort プログラム

```
go:-true |
   quicksort([5,9,2,7,3,6,10,4,1,8],X),screen (X?).
screen([]):-display([]) | true.
screen([X | Y]):-display(X) | screen(Y?).
quicksort(X,Y):-true | qsort(X,Y).
qsort([X | Y],R):-true |
  partition (Y?,X,S,L),qsort(S? ,S1) ,
  qsort(L?,L1),lappend(S1?,[X | L1],R).
  ( partition ,lappendは省略)
```

並列推論マシンＰＩＭ－Ｒにおける プロセス内部表現

6c-7

清水 肇　麻生 盛敏　益田 嘉直　尾内 理紀夫
（財団法人 新世代コンピュータ技術開発機構）

1．はじめに

現在我々は、structure-copy方式を採用したリダクション概念に基づく並列推論マシン:PIM-Rの検討、ならびに、シミュレーションによる評価を進めている[1][2]。

structure-copy方式は、個々のプロセスの独立性を高め、構造体の共有に起因するネットワーク・トラフィックを低減するが、コピーのためのオーバーヘッドが増加する。

本稿では、このオーバーヘッド低減のための、プロセスの内部形式と逆順コンパクションについて述べる。

2．プロセス内部形式

内部形式について述べる前に内部形式内のデータタイプ一覧を表 1に示す。

図１に示すように、Process Pool [1]におけるプロセス内のgoal列のテンプレートを格納するProcess テンプレート部は、ヘッダー、リテラルヘッダー、リテラルエリア、ストラクチャエリアからなっている。なお、clauseの定義を格納するClause Pool [1] 内のClause定義部もこの形式に従っている。

ヘッダーは、clause長、ストラクチャエリア先頭番地、リテラルヘッダー先頭番地からなり、変数エリアには、変数個数と各変数のバインド情報が格納される。

リテラルヘッダーには、リテラル数（逐次AND 関係にあるボディリテラルの数+1）、ヘッドリテラル、逐次AND 関係にあるボディリテラル（ただし、commit operator は 1つのリテラルとみなす）を逆順に並べて、格納する。

ここで、引数個数が 0であるリテラルは、リテラルヘッダーに（データタイプのPoi,Sym として）格納されるが、引数個数が 1以上であるリテラルや、並列AND 関係にあるリテラルは、データタイプのLit,Paraをそれぞれに用いてリテラルエリアに逆順に格納する。

リテラルエリアに格納されるリテラル（Lit タイプのポインタによって指される）には、引数個数、Clause PoolのClause定義グループ管理部（OR関係にあるclause数と、それぞれのclauseが格納されているClause定義部へのポインタから構成される）へのポインタ、各引数が格納される。

Int	clause長	ヘッダー
Int	ストラクチャエリア先頭番地	
Int	リテラルヘッダー先頭番地	
Int	変数個数	変数エリア
Int	リテラル数	リテラルヘッダー
Type	ヘッドリテラル	
Type	ボディリテラル N	
Type	ボディリテラル 1	
		リテラルエリア
		ストラクチャエリア

注1) Typeは、データタイプの Poi, Lit, Sym, Paraのいずれか

図１ プロセスの内部形式

Type Classification			Abbreviation
Type	SubType1	SubType2	
Variable	Void-variable		Void
	Variable-1'st		Var1
	Variable-ref		VarR
Channel	Undef Channel	1st	UCh1
		ref.	UChR
	Read Channel	1st	RCh1
		ref.	RChR
	Write Channel	1st	WCh1
		ref.	WChR
Atomic	User Defined Atom		Atom
	Nil		Nil
	System Symbol	Type0	Sym0
		Type1	Sym1
		Type2	Sym2
	Integer		Int
	Real		Real
Structure	List		List
	String		Strg
	Vector		Vect
	Channel Information		ChI
	And	Parallel	Para
		Seq	Seq
	Literal		Lit
	Pointer		Poi
Structured Pointer	List		SPGL
	String		SPGS
	Vector		SPGV

表１ データタイプ

このリテラルヘッダーや、リテラルエリアに、リテラルを逆順に格納する方式により、3．に述べる逆順コンパクションをやり易くしている。

構造体データ（リスト、ベクタ、チャネルに関する情報）は、ストラクチャエリアに格納される。ただし、構造体データがない場合はストラクチャエリアは存在しない。

3．逆順コンパクション(reverse compaction)

Process Poolにあるプロセス、

```
p(1,[X | Y]):- q(1,X) & r(X,Y).
```

のProcess テンプレート部を図2の左側に示す。ただし、'&' は逐次AND operatorとする。

ここで、第2語目が 0番地であり、リテラルの格納位置は、リテラルヘッダーの先頭番地からのdisplacementで指される。また、構造体データの値の格納位置は、ストラクチャエリアの先頭番地からのdisplacementで指される。

リテラル数は、リテラルエリアの先頭に格納されている。この時点でリテラル数は、ボディリテラル数+1=3であり、リテラルヘッダーの先頭番地からのdisplacement 3にあるLit 13により、p(1,X)がreducible であることが示され、これをUnification Unitへ送る。

今、Unification Unitで、unit clause とunify して、q(1,2)がProcess Poolへもどってくると仮定する。

Prologの場合、複数の解がreturnされる可能性があるので、まず、このProcess テンプレート部全体をコピーしてから結合情報X-2 を書き込む（この結果変数エリア、ストラクチャエリアが変化する場合がある）。

このままでは、Process Poolのメモリ使用効率が悪いので、不要となったリテラルq に相当する部分をリテラルエリアから抜き、ストラクチャエリアを前に詰め、リテラル数を-1する。この時、リテラルは逆順に格納されているためリテラルエリアの後から抜くだけでよく、構造体データの値の格納位置は、ストラクチャエリアの先頭番地からのdisplacementで指されるために、ポインタのはりかえは必要としない。また、リテラル数を-1することによって、次にreducible であるリテラル r(2,Y) が指される。即ち、リテラル数はreducible あるいはrun しているgoalリテラルを指している。

以上の操作後の状態を図2の右側に示す。

ストラクチャエリアが変化する場合、即ち、新たに構造体データが追加される場合は、ストラクチャエリアの後につなげればよい。

変数エリアが変化する場合、即ち、新たな変数が追加される場合は、リテラル・ヘッダー先頭番地とストラクチャエリア先頭番地を変更する。

組込み述語や、Concurrent Prolog の場合は解は1つだけ（生き残れる兄弟プロセスは1つ）なので、解がreturnされてきた時、Process テンプレート部をコピーする必要はなく、この逆順コンパクションによりclause長が長くならないならば、直接overwrite してよい。

この方式により、コピー量、Process Poolの使用量を低減することができる。

4．おわりに

PIM-R の効率的なstructure-copy方式を支援するプロセス内部形式について述べた。現在、この方式にのっとったPIM-R のソフトウェア・シミュレータをoccam で記述中であり、今後、本方式の評価を進める予定である。

最後に、ご討論いただいた、日立中央研究所　杉江、米山、坂部、岩崎各氏、また、日頃ご指導いただく第1研究室村上室長はじめ並列推論マシングループ諸氏に感謝の意を表する。

［参考文献］

[1] 尾内ｅｔａｌ．“並列推論マシンPIM-R のアーキテクチャ”　本大会予稿集　6C-6

[2] 尾内ｅｔａｌ．“並列推論マシンPIM-R のソフトウェアシミュレーション”　本大会予稿集　6C-9

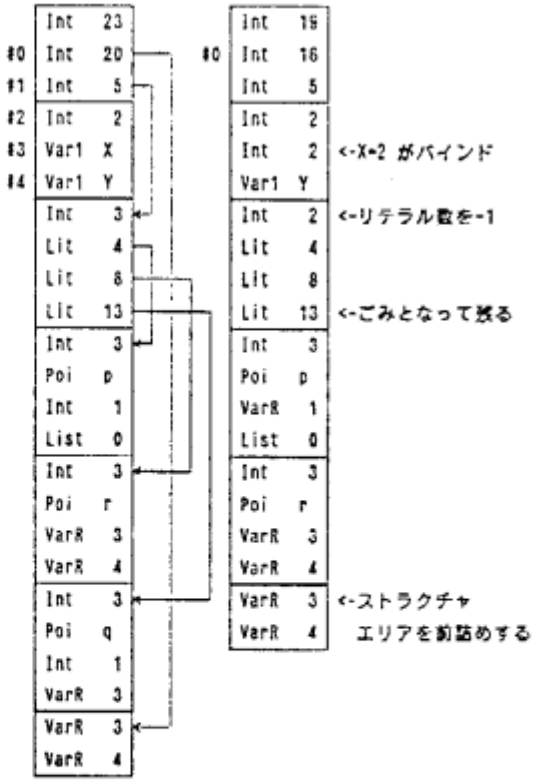


図2　逆順コンパクション処理の前後の状態

ＰＳＩにおけるオブジェクトサポート

/C-/

横田 実　近山 隆　西川 宏　　中島 克人
（　（財）新世代コンピュータ技術開発機構　）　（　三菱電機（株）　）

1．はじめに

逐次型推論マシンPSI[1][2] は第五世代コンピュータ研究開発のためのツールとして開発された論理型プログラム専用マシンであり，現在，そのオペレーティングシステムSIMPOSの開発が述語論理に基づくシステム記述言語ESP[3][4] を用いておこなわれている．ESP の最大の特徴はオブジェクト指向の概念を述語論理の世界に取込んだ点である．本論文ではPSI におけるオブジェクトの実現方法と，マイクロプログラムによるオブジェクトアクセスのサポート機能について論ずる．

2．PSI におけるオブジェクトの導入

ESP はもともと論理型言語として設計されたものであるが，オブジェクト指向型言語としての特徴であるクラスの概念，多重継承やデモンの機構を備えている．これまでのオブジェクト指向型言語との違いはメソッドが述語論理で記述されることであり，オブジェクトに対するメッセージの伝達はユニフィケーションにより達成される．また，設計の方針としてインプリメントの容易さと実用面からの実行効率を考えて，クラス間の関係は静的に決定することとし，コンパイラにより効率の良いオブジェクトコードを作り出すよう考慮した．従来型マシンにおいて最終的にメソッド呼出しがサブルーチンの呼出しに変換されるように，PSI ではそれが述語呼出しに変換される．従って，マシン側からは通常の論理型言語の実行と変らない．違いは呼出すべき述語（メソッド）が引き数として渡されたオブジェクトにより動的に決定される点である．（図１）このための特殊な述語呼出し用組込述語method-call,overridden-method-callが設けられた．また，オブジェクトとは本来，内部状態を有するものであり，それを実現するものがオブジェクトに附随する局所変数(ESPではスロットと呼ぶ) である．PSI ではこの変数をヒープ上にとり，論理変数ではないサイドエフェクトを有する変数として実現している．この変数へのアクセスのために組込述語slot,set-slot を設けた．

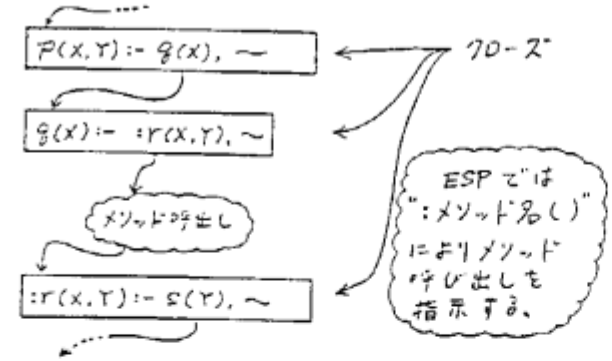


図1　PSI における メソッド呼び出し

3．マイクロプログラムによるインプリメント

図２にオブジェクトの内部データ形式を示す．すべてのオブジェクトはヒープ上のベクタとして生成される．オブジェクトの持つ局所変数はそのベクタ要素として実現される．あるクラスが継承している全てのメソッドはコンパイラにより１つのメソッドテーブルとしてまとめられる．同様に局所変数の名前とスロット位置との対応はスロットテーブルとしてまとめられる．この２つのテーブルはオブジェクト記述子を介してそのクラスに属するすべてのオブジェクトから共有される．

メソッドテーブル，スロットテーブルへのアクセスはメソッド名，スロット名（実際には名前を示すアトム番号）によりハッシュする方法を採用した．もしハッシュ値の衝突が生じた場合には，ハッシュしたエントリ位置以降の最初の空きエントリが格納場所として利用される．

（１）メソッド呼出し

ESP でのメソッド呼出しは機械語レベルで，組込述語me

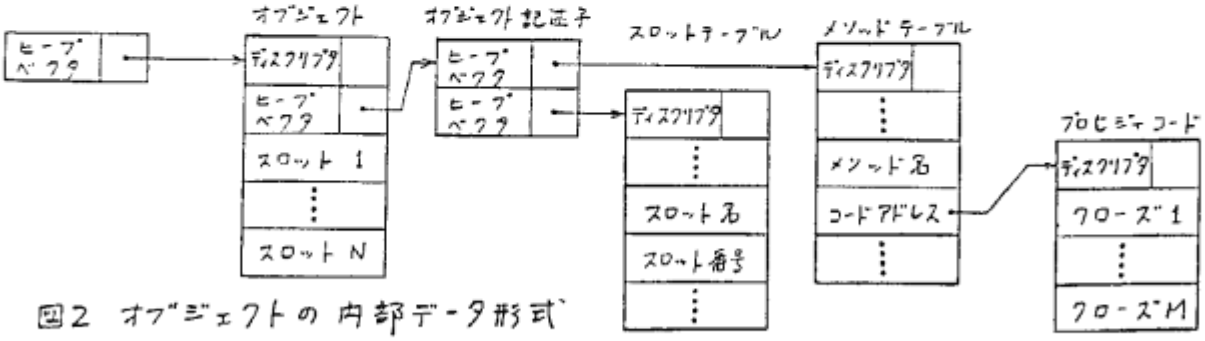


図2　オブジェクトの内部データ形式

thod-call(Object,Method,Arity,Arg1,...,ArgN)を用い作用対象となるオブジェクト，実行すべきメソッド名と引き数の数，およびメソッドに渡す実際の引き数を指定することにより実現される．ファームウェアはまず，指定されたオブジェクトの形式が妥当であるかをチェックした後，オブジェクトの持つメソッドテーブルをメソッド名によりハッシュし目的の述語アドレスを得る．PSI ではクローズの内部形式をプロセジャ単位に生成しており，実際にはこのプロセジャアドレスが得られる．同一のメソッド名がスーパクラスにも存在し，直接それを呼出したい場合にはコンパイラにより識別され，特定のメソッドテーブルを直接指定する組込み述語overridden-method-call(MethodTable,Method,Arity,Arg1,...,ArgN)によってメソッドの呼出しが行なわれる．呼びだすべき述語のコードが求まった後は通常のユニフィケーションが開始される．

（２）スロットアクセス

オブジェクトの持つ局所変数の値を読みだすには組込述語slot(Object,Slot,Value) が，また，局所変数への値の代入にはset-slot(Object,Slot,Value) が使用される．局所変数への代入は履歴を持ち，バックトラックによってもundoされない．ファームウェアはメソッド呼出しの場合と同じ方法でスロットテーブルをアクセスしスロット名に対応するスロット位置を求める．この方式では変数へのアクセスを単なるオフセットとして直接アクセスするよりも効率は悪いが，多重継承を実現するためには同一のスロット名がクラス毎に異なるスロット位置を持つ必要があり，これを実現するための最もシンプルな手法として採用した．

４．評価

ESP の持つオブジェクト指向の特徴がどのくらい利用されているか，また，ファームウェアサポートがどのくらい効を奏しているかについて現在開発中のSIMPOSＯ．７版を用い小さなプログラム（フィボナッチ数の計算）を編集，コンパイルし実行するという一連の操作をマルチウィンドウシステムを利用して行いデータを収集した．（表１）

（１）オブジェクトの利用率

全述語呼出しの９０％は組込述語の呼出しであるが，その１７％がスロットのアクセス，１１％がメソッドの呼出しである．通常のローカル述語呼出しとメソッド呼出しとの比は１：１．２５でありメソッド呼出しがかなり利用されていることが分る．

（２）オブジェクト操作のオーバヘッド

通常の述語呼出しに比べてメソッド呼出しがどのくらいオーバヘッドを持っているかについてマイクロプログラムの実行トレースより検討した結果，１回のメソッド呼出し当たり約５０－６０ステップ(10-12microsec) であった．３０KLIPS という仮定からマイクロプログラムの実行ステップ数は１推論当たり平均１６０ステップとなり，メソッド呼出しのオーバヘッドは３割強となる．内訳の主なものとして引き数の評価に１０ステップ，メソッドテーブルの取り出しに１０ステップ，ハッシングによるテーブル検索時間はハッシュのヒット率に依存するが，評価の結果，１回のメソッド呼出しに要するテーブルへのアクセス回数は３．６回で，これが約２０ステップのオーバヘッドとなっている．メソッドテーブルへのアクセス回数が多い点も検討の余地があるが，ミスヒット時のオーバヘッドはむしろ引き数の数がメソッドテーブルに入っていないことに影響されている．

スロットアクセスの場合，スロットテーブルへのアクセス回数は平均１．３回で全体の実行時間は約４０ステップ(8microsec) であった．これは普通のベクタ要素へのアクセスとほとんど変りない速度である．

５．おわりに

評価の結果，SIMPOSでは予想以上にオブジェクト指向の特徴が利用されていることが分った．ファームウェアのサポートとして４つの組込述語を導入したが，これにより充分高速にオブジェクトを扱うことが可能であることも分った．しかしながら，オブジェクトアクセスのためには連鎖カットの使用などオーバヘッドとなる要因が他にもあるので，現在，実際のユーザからみた場合の影響について評価中である．さらにオブジェクトアドレスのキャッシングなどの改善によりどの程度オーバヘッドを小さくできるかについても評価中である．

ローカル述語呼出し回数	838,398
組込述語の呼出し回数	9,399,595
メソッド呼出し	1,039,150(11%)
スロットアクセス	1,656,297(17%)
メソッドテーブルへの平均アクセス回数	3.6
スロットテーブルへの平均アクセス回数	1.3

表１ SIMPOS でのオブジェクト利用に関する評価結果

参考文献

[1]Uchida,S. et.al., Outline of the Personal Sequential Inference Machine PSI,New Generation Computing,1-1,1983

[2]Yokota,H. et.al., The Design and Implementation of a Personal Sequential Inference Machine:PSI, ICOT Technical Report,TR-045,1984

[3]Chikayama,T., Unique Features of ESP,Proc. of FGCS'84,1984

[4]Chikayama,T., ESP Reference Manual,ICOT Technical Report,TR-044,1984

並列論理型言語のコンパイル技法

ミタ-ゴ

上田 和紀
(日本電気(株) C&C システム研究所)

0. はじめに

最近1年間ほど, Concurrent Prolog[1] (CP)のコンパイル技法の研究を行なってきた。またそれと並行してCPの言語仕様の問題点を考察・検討してきた。それらの成果の概要を報告する。

1. コンパイル技法

並列論理型言語としてCPを取りあげ, 以下の項目を中心に検討を行なった。対象としたマシンは通常の汎用機である。汎用機上での実現法の研究は, 速度や移植性の点で当面最も有利であること, 近未来の並列マシンにおいて各プロセサが多プロセスを扱う方法を提供すること等の理由できわめて重要である。

1-a) ストリーム並列の効率的実現

並列論理型言語を実用的な汎用言語とするためには, ストリーム並列[2]を効率的に実現しなければならない。特に多対1のプロセス間通信の際必要なストリームの併合を効率よく実現する必要がある。筆者らは, n本のストリーム(静的にはリスト)を併合する述語を解析し, コンパイル技法の工夫によってO(1)の遅れをもつ併合器が実現できることを示した[3]。この併合器は, 入力ストリーム数nの動的増減を許す併合器に(O(1)の遅れを保ったままで)拡張できる。実用上は, 併合器は組込述語として定義するのがよい。

さらに3)では, 配列をゴールとして実現すると, O(1)の時間で読み書きができることを示した。ここでは配列は

array(N, S)　%Nは配列の大きさ

というゴールであり, 配列の利用者はストリームSを通じて, read(添字, 変数)・write(添字, 値)といったメッセージを送りこむ。このようにストリームをインターフェースとして用いると、(要素の出し入れのできる)二分木や重つなぎ並びのような, Prologでは実現のむずかしかった動的データ構造も容易に実現できる。

このような, ストリームを多用するプログラミングスタイルが実用的であるためには, ストリーム通信の効率が非常に高くなければならない。文字列ストリームについては4)でべたづめ表現法を提案したが, 一般のストリームの効率的実現についても明るい見通しを得ている。

1-b) 実用処理系

CPのプログラムをDEC-10 Prologのプログラムに変換するコンパイラを作成した[5]。DEC-10 Prologはコンパイラを持っているから, CPプログラムは結局機械語になって走る。

速度は予想以上に高く, モード宣言を与えたappendプログラムで11.7 kLIPSを得た(DEC2060)。ちなみにDEC-10 Prologでのappendの実行速度は, コンパイラで42 kLIPS, インタプリタで2.7 kLIPSであった。高効率の源は, CPのユニファイア(Prolog プログラムとして定義する必要がある)を可能な限り最適化したこと, およびCPのtail recursiveなプログラムがPrologのtail recursiveなプログラムに変換されるようにしたことにある。Tail recursiveなプログラムの最適化は, ストリームを多用するプロ

グラムにとってきわめて重要である。

記述言語とターゲット言語をPrologにしたことにより，処理系は短期間のうちに完成した。これにはソース・ターゲット・記述言語間の類似性と，それに伴う実行時支援作成作業の軽減が大きく寄与している。実現を容易にするために「失敗」の概念を導入せず，また動的待合せを用いたりもしているが，実用上の不都合は今のところない。

2. CPの言語仕様の問題とその解決

CPの言語仕様を，並列実行という観点から詳細に検討した[6]。以下にその主な結果を要約する。

a) 述語頭部の単一化およびガード部の実行は，すべて並列に行なうべきである。

b) ガード部の局所環境[7~9]と大域環境とが矛盾する場合，その矛盾の発生のしかたによって，commit前に検出しなければならない場合と，(実現の困難さ故) commit後の検出を許さなければならない場合とがある。

c) Xとf(Y?)との単一化の意味を，Xとf(Z)，ZとY? というふたつの単一化に分けてとらえてはいけない。

aはCPの場合，実現上の負担を重くする。bとcは実用上の不都合は少ないように見えるが，意味記述が複雑化し，プログラム変換のような操作をむずかしくする。

上記の問題点はすべて，CPのread-only annotationと多環境の存在に起因している。そこで，それらを廃止した単純な言語を提案した[10]。CPとの主要な相違は次の一点である。CPでは，commit前に，ゴール側の変数を他のゴール側変数や非変数項と単一化する場合，そのユニファイアを局所的に保存し，commit時に外に公開する。新しい言語では，そのような単一化は，自らユニファイアを生成せずに成功するようになるまで中断させる。この変更により，必要な記述力を保ったままで，CPの種々の問題点を解決し，CPよりもPARLOG[11]よりも単純な言語とすることができた。

なお，本研究は第5世代コンピュータプロジェクトの一環として行なった。

参考文献

1) Shapiro, E., A Subset of Concurrent Prolog and Its Interpreter, ICOT Tech. Report TR-003 (1983).

2) Conery, J. and Kibler, D., Parallel Interpretation of Logic Programs, Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture, pp. 163-170 (1981).

3) Ueda, K. and Chikayama, T., Efficient Stream/Array Processing in Logic Programming Languages, Proc. FGCS '84, pp. 317-326 (1984).

4) 上田他，論理型言語による文字列処理の記述について，情報処理学会記号処理研究会資料 26-1 (1983).

5) 上田・近山，並列論理型言語の実用処理系，日本ソフトウェア科学会第1回大会 3D-5 (1984).

6) Ueda, K., Concurrent Prolog Re-Examined, to appear as ICOT TR.

7~9) 宮崎他，佐藤他[8]，田中他[9]，Concurrent Prologのシーケンシャルインプリメンテーション——Shallow binding (Deep binding[8], Copy[9]) 方式による多環境の実現——日本ソフトウェア科学会第1回大会 3D-2-4 (1984).

10) Ueda, K., Guarded Horn Clauses, to appear as ICOT TR.

11) Clark, K. and Gregory S., PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept. of Computing, Imperial College (1984).

SIMPOSのプログラミング・システム

— エラー処理／ヘルプ機能 —

渡瀬 哲朗　東条 敏　吉田 紀彦（㈱三菱総合研究所）
溝口 徹夫（三菱電機㈱）　黒川 利明　近山 隆（(財) ICOT）

1．はじめに

例外事象はソフトウェア・システムにおいて重要な概念のひとつである．ここで例外事象とは以下のようなことである．

・ファームウェアが検出した zero division，
・装置が検出した入出力エラー，
・ソフトウェアが検出したエラー，
・複雑に入れ子になった手続き中の大域脱出，
・作業中の利用者に対するヘルプ
　　等々．

本稿では，PSI(Personal Sequential Inference machine) 上に開発中のプログラミング／オペレーティング・システムであるSIMPOSにおける例外処理の基本概念と枠組について述べる．

2．方針

例外事象はソフトウェア／ファームウェアの一部により検出され，引き起こされる．この処理実行部をdetectorと呼ぶ．例外事象が起こるとその例外事象に対応した処理が始まる．この処理実行部をhandler と呼ぶ．例を示す．

"エディタを利用して文書編集を行っている．ファイル処理部を呼び出し，ファイルFへ文書を書込もうとした．ところがファイルFが存在しなかった．このようなエラーが生じた場合は，エディタのコマンド・ループに戻る．"

以上の処理を対応付ける．まずエディタ中かそれより前の時点で，エディタのコマンド・ループへ戻すhandler H を設定する．エラーが起こったとき，エディタはあくまでファイル処理部の呼出し元であり，ファイル処理部がdetectorとなる．例外事象がdetectorにより引き起こされてhandler Hがはたらき出す．（図１）

同じ例外事象でも発生する環境により，様々な意味をもつ（この環境設定については後述する），また各例外事象の発生は，その意味に従って別々に処理される．

3．基本的枠組

(1) situation

例外処理法の文脈をsituation として定義する．situation は，例外事象とそれに対応したhandler の組を要素にもつ．これをsituation component と呼ぶ．situation はスタックに似た構造をもつ．プロセスはあるプログラム部分を実行させるために，必要分のsituation component をsituation に付け加える．実行終了と同時(success, failどちらでも) にsituation を元の状態に戻す．situation は一時的にのみ変化する．つまり，例外処理のための(ある意味で副作用のない) 動的環境になっている．

もちろん，default のsituation が存在する．例外事象に対する常識的処理と考えられる．

(2) 例外事象の分類

各例外事象は階層付けられ，数種に分類される．例えば①file error，②file not found　の２つの例外事象を考える．②は①のひとつと考えられるので，①を親，②を子と階層付けられる．例外事象②が起こったとき，②を含むsituation component だけではなく①を含むsituation component も探索される．

(3) 大域的な制御の移行

detector-handler機構にsituation を導入することにより，大域的な制御の移行が実現される．Lispのcatch and throw の処理は "detectorがthrow を起こし，situationからcatcher を取出し，catcher がその登録時のレベル付近に制御を移行する" といった具合に実現される．Lispのerrorsetやerror 回復付きのコマンド・ループなども実現される．

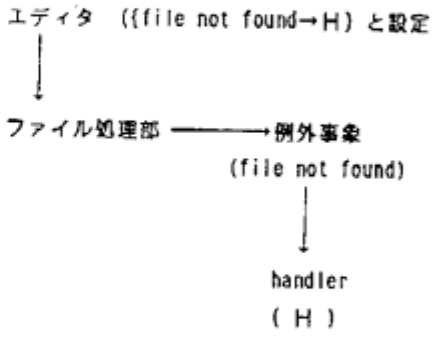


図　1

4．実現法

(1)event

例外事象はESP のクラスによって分類される．その継承の頂点に来るのがevent である．event のサブクラスとしてerror とhelpを用意する．さらに各サブシステムによって，それぞれの例外事象が継承により階層付けられて実現される(図２)．また警告，致命的エラー，通常エラーなどのサブクラスにも分類される．このように例外事象は多重継承を用いて表現される．SIHPOSのような発展中のシステムでは，この階層がさらに拡がる可能性をもつ．

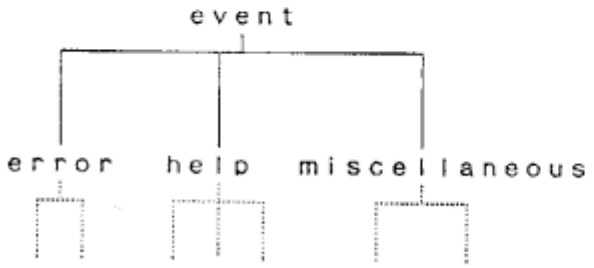


図 2

(2) 例外事象の発生

detectorは例外事象を見い出すと，次のようにして例外事象を起こす．

・見い出された例外事象に相当するインスタンス・オブジェクトを作成する．

・このインスタンス・オブジェクトに必要な情報を与える．

・インスタンス・オブジェクトに例外事象を引き起こすメッセージを送る．

例外事象が発生すると，そのインスタンス・オブジェクト自身がhandler をみつけ，実行させる．

(3) handler の探索

situation は上述のsituation component で構成される．situation component は例外事象名に相当するクラスのクラス・オブジェクトとhandler のインスタンス・オブジェクトの組で構成される．

handler の探索には例外事象の継承関係を利用する．例えば図３のような例外事象の継承関係が存在したとする．error-p とその処理者handler から成るsituation component をsituation に加えてプログラム部分を実行したとする．error-p が起こればhandler が探索され，起動される．もしerror-c が起これば，継承関係によりerror-p を含んだsituation component のhandler が探索され，そのhandler が処理を任される．

一般に以下のような利用が可能となる．default situation では総の例外事象に対し普通の処理を設定する．あるプログラム部分の実行時のみ，子の例外事象に対し特別処理を設定する．

また，最初にみつかったhandler がfailすると次のhandler を探索される(例外事象がfailするのではなく，handler の実行がfailすることに注意せよ)．さらにfailした場合には次のhandler が探索される．failを続ければ，最終的に "event " のhandler が起動される(defaultで設定される)．

実行される可能性があるhandler 項目をメニュに出力し，利用者に選択させることもある．

図 3

(4) 制御機構

situation のような動的環境や，大域的な制御をPrologで実現するのは極めて難しい．我々はKL0 におけるPrologの拡張機能［１］を利用した．catch and throw,errorsetなどは遠隔cut を利用してfailさせることとする．そして遠隔cut and failによるsituation の状態の回復には，on backtrack機能を用いる．

(5) situation

先程も述べたようにsituation はsituation componentのスタックと考えて構わない．situation は各プロセスに１つずつ独立に存在する．

5．問題点

現在，situation がプロセス単位に独立であるため，複数プロセスをまたがる例外処理は特にサポートされていない．各プロセスで，例外が起こったことをポートを通して受信し，その受信者がdetectorとなって例外を引き起こすことになる．

SIHPOS自体，実験的要素が多い．この例外処理に関してもさらに経験を積んで改良する予定である．

参考文献

［１］高木　他：拡張制御構造のPrologへの導入，
昭和58年度前期情報処理学会全国大会論文集，
pp.37-38．

SIMPOSのプログラミング・システム
---ウインドウ・マニピュレータ---

中沢 修　　飯間 豊　　榎本 章二　　辻 順一郎
（沖電気工業（株））　（松下電器（株））　（(財) ICOT）

1. はじめに

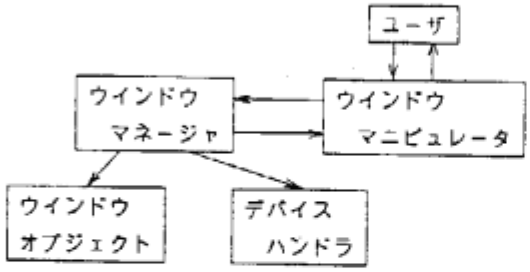
SIMPOSのウインドウ・サブシステム[1]は，一つの物理端末（スクリーン）上に多数の論理端末（ウインドウ）を構築し，複数プロセス間の端末の共有化を図る様に構成されている。そのウインドウ・サブシステムの基で作動するウインドウ・マニピュレータ（以下，単にマニピュレータと呼ぶ）は，ウインドウを利用するユーザとのインタフェースの向上を目的とし，マウスを用いてウインドウの作成／移動／状態の変更等を行なうものである。

本稿では，PSI（逐次型推論マシン）上でのプログラミング環境の向上に重要な役割りを果すマニピュレータに関して 1)その位置付け，2)機能及び特徴，3)マニピュレータの動作に関係のあるウインドウの状態等について述べる。

2. ウインドウ・マニピュレータの位置付け

ウインドウ・サブシステム（ウインドウ・マネージャ）とマニピュレータとの関係を図．1に示す。

マニピュレータは，ウインドウの作成／移動／状態の変更等を行なうことができるが，これらはすべてウインドウ・マネージャプロセスによって制御されている。
つまり，マニピュレータは，ユーザとの対話によって得られた情報を基に種々の評価を行なった後，実際の画面上での変更，ウインドウの追加等をウインドウ・マネージャに依頼する。



図．1　マニピュレータの位置付け

3. ウインドウの状態およびその遷移

ウインドウ・サブシステムが管理するウインドウオブジェクトは，階層を構成する。階層のルートには，スクリーンと呼ばれるシステムウインドウがあり，一般のウインドウは，すべてこのスクリーンの下に作られる。
あるウインドウから見て階層のすぐ上のウインドウを親ウインドウ，すぐ下を子ウインドウといい，同一の親を持つウインドウを兄弟ウインドウという。

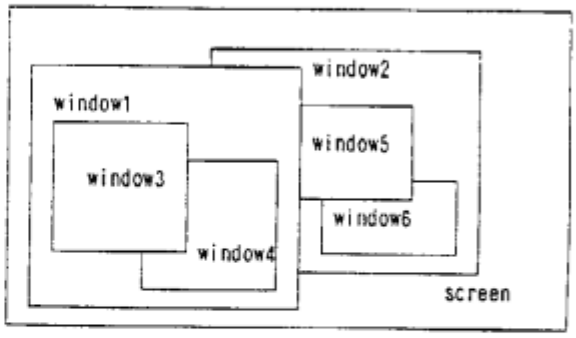
3．1　ウインドウの状態

1）　exposed / deexposed

ウインドウは，図．2に示す様に互いに重なり合うことができる。重なり合いのために他の兄弟ウインドウに隠されたウインドウをdeexposed といい，全く隠されていないウインドウをexposed という。又，兄弟ウインドウの中でexposed であるものは一つとは限らず，重なりがない限り，いくつでもexposed になり得る。
図．2においては，exposed なウインドウはwindow1,window3,window5 で，deexposed なウインドウはwindow2,window4,window6 である。

2）　shown / unshown

exposed なウインドウがすべて人間に見えるとは限らず，自分自身がexposed で親もexposed で，その親もすべてexposed なウインドウだけが完全に見える。この様なウインドウをshown といい，それ以外のウインドウをunshown という。
図．2においては，shown なウインドウはwindow1,window3 で，それ以外がunshown である。



screen の子ウインドウ　：　window1,window2
window1 の子ウインドウ　：　window3,window4
window2 の子ウインドウ　：　window5,window6

図．2　ウインドウの状態

3．2　表示状態の遷移

ウインドウの表示状態には，3.1で述べた様にexposed/deexposed とshown/unshown があるが，各ウインドウが取り得る状態は，1)exposed でshown ／ 2)exposed でunshown ／ 3)deexposed でunshown の三通りである。
ウインドウは，常に上記の数の小さい状態になろうとしている。状態が2)又は3)に止まっているのは，他のウインドウに妨害されているためである。

ウインドウの状態は以下の場合に変わる。

・ユーザプロセスのプログラムからそのウインドウの表示状態を変える要求が出された時
・マニピュレータによって変更が指示された時
・他のウインドウの状態が変わったための影響を受けた時

4．ウインドウ・マニピュレータの機能

マニピュレータは，ウインドウの作成及びサイズ／位置その他の表示状態をユーザとの対話によって変更するプログラムである。これによって，ウインドウの状態等を単にユーザプロセスのプログラムにより変えるだけでなく，マウスにより指定できるため，必要な時すぐに

・見たいウインドウを一番上に出す
・必要のなくなったウインドウを消す
・テキストを見易くするために，ウインドウの大きさを変える

等，使用環境の向上を図ることができる。

マニピュレータは，以下の様な手順で動作する。

1. コーディネータ（別稿参照）によって起動される
2. 図．3に示す様なコマンド選択のためのメニューが現われる
3. ユーザによって，このメニューの中から実行すべき項目が選ばれるのを待つ
4. ユーザとの対話によって必要な情報を取り込む
5. 種々の評価を行ない，ウインドウ・マネージャに要求する処理を決め，それを依頼する

以下にマニピュレータ項目の概要を示す。

1）　SHOW

スクリーン上に部分的に見えているウインドウを完全に見える状態にする。

2）　HIDE

スクリーン上に見えているウインドウをウインドウの重なりの最下層に移す。

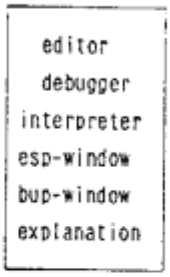
3）　SHOW(MENU)

現在システム中に存在するウインドウをサブメニュー（図．4）を用いて選択し，SHOWする。

4）　SELECT(MENU)

現在システム中に存在するウインドウをサブメニューを用いて選択し，selectedなウインドウにする。

ここで，selectedなウインドウというのは，キーボード入力の行なわれるウインドウのことである



図．3　コマンドメニュー　図．4　サブメニューの例

5）　RESHAPE

スクリーン上でマウスによりウインドウを選択し，そのウインドウの左上端と右下端を再指定することにより，そのウインドウの位置，サイズを変える。

6）　MOVE

スクリーン上でマウスによりウインドウを選択し，その後のマウスの移動によりウインドウの位置を変える。

7）　EXPAND

スクリーン上でマウスによりウインドウを選択し，そのウインドウを他の完全に見えているウインドウを隠さない程度で最大のサイズを持つ様に拡大する。

又，ウインドウの作成に関しては，ウインドウ・サブシステムを利用するユーザプロセスのプログラムから起動される。作成の種類としては，

・ウインドウの位置のみの設定
・ウインドウの右下端の指定によるサイズのみの設定
・ウインドウの左上端と右下端の指定による位置・サイズの設定

があり，この種の設定を行なった後，実際の作成作業をウインドウ・マネージャに依頼する。

5．　おわりに

本稿で説明したマニピュレータは，コーディネータ等との結合を終え，その動作は検証済みである。しかし，マニピュレータはマンマシンインタフェースに関わるソフトウェアであるため，今後は，使用経験を基に，使い易さ，処理効率の向上，及び機能拡充等について検討を行なう必要がある。

6．　参考文献

1)飯間，中沢，樫本，辻
ＳＩＭＰＯＳのウインドウ・サブシステム
情報処理学会第29回全国大会予稿４Ｅ－６

SIMPOSのプログラミング・システム
— コンパイラ —

4E-7

高木 茂行　近山 隆
((財) ICOT)

1. はじめに

逐次型推論マシンφのオペレーティング・システム／プログラミング・システムであるSIMPOSの実行管理系の一部であるコンパイラについて述べる。

コンパイラは，ソース・プログラムをテンプレートに変換するESP コンパイラ，テンプレートからオブジェクトを作成するコンポーザ，テンプレートの持つ中間コードを機械語(KL0) に変換するKL0 コンパイラの3つの独立したモジュールから成る。以下各々について説明する。

2. ライブラリのモジュール管理

実行管理を行なうライブラリは，φ上のプログラムである個々のクラスをソース，テンプレート，オブジェクトの3種の形式で管理する。ユーザが作成したソース・プログラムはライブラリに登録することによって，はじめてライブラリに管理されるようになる。

ソースはユーザがエディタ(Edips) で作成したJIS コードのテキスト・ファイルである。ライブラリはこのソース・ファイルの別名として，ライブラリのディレクトリの下に"クラス名．ESP"というファイル名を登録し，以後はこの名でソース・プログラムを管理する。

テンプレートは，1個のクラス定義からわかるクラスの単体情報をまとめたものである。これはソース・プログラムからESP コンパイラによって作成され，そのクラス自身のスロット情報，述語情報，及びそのクラスが直接継承・参照する他のクラスの情報が含まれる。述語情報には，解釈実行コード，コンパイル済コードの両方が存在する。ユーザはどちらのコードを使用してオブジェクトを作成することも可能である。テンプレートは主記憶上に作成されるが，同時にディスク上のファイルとしても書き出される。以後はソースではなく，このテンプレートのファイルから主記憶上に読み込まれて使用される。

オブジェクトは，汎用計算機での実行形式のプログラムに相当するものである。これはテンプレートではまだ未解決の他クラスへのリンケージ・ポインタを設定し，継承しているクラスの情報を全て集めて1個の完結したクラスを主記憶上に作成したものである。

3. ESPコンパイラ

ESP コンパイラは，ソース・プログラムからテンプレートを作成する。この処理の結果，解釈実行コードを持つテンプレートが主記憶上にできあがる。同時にテンプレートはディスク・ファイルに出力され，後の使用に用いることができる。

ESP コンパイラは1個のクラス定義を読み込み，その字面から解析できる情報を取り扱う。すなわち，そのクラスが直接に継承・参照するクラスの表の作成，そのクラスが持つスロットの表の作成，そのクラスで定義された述語の解釈実行コードの作成が主たる役割である。当該クラスで定義される全ての述語（スロットの初期化，メソッドのデモン・主処理，局所述語）は，ここで処理して解釈実行コードを作るためのタームに変換する。このタームをKL0 コンパイラによって，インタプリタの処理する形式に変換したものが，テンプレートの持つ解釈実行コードである。

SIMPOSのインタプリタは，タームを直接解釈実行するのではなく，タームをKL0 コンパイラによって処理した解釈実行コードを解釈実行する。これはヘッド部，ボディ部，及び変数名表を引数に持つユニット・クローズの形式でテンプレートに格納する。これによって中間コードをファイルの中に格納し，再利用が可能となる。解釈実行用にタームの形式に戻すには，単にこのユニット・クローズを呼出せば良い。

もう1つのESP コンパイラの仕事はソース・プログラムのマクロ展開である。ESP の言語仕様では，処理の略記法としてメソッドの呼出しを　：メソッド名（オブジェクト，引数…）　のように書いたり，演算を　X＋5　のように書いたりできる。これらの記法をKL0 機械語で処理できる命令（列）に置きかえることが必要であるが，この処理はソース・プログラムを読み込んだ直後に行ない，以後は展開後の結果を用いて処理する。

4. KL0コンパイラ

KL0 コンパイラは，ESP コンパイラによって作成されたテンプレートの解釈実行コードからφの機械語であるKL0 のコードを作成する。汎用機でのコンパイルにあたる部分を主に担当する。従って，クラスの継承・参照等の情報には関与しない。また，φにはインタプリタがあり，解釈実行が可能である。そのため，KL0 コンパイラによるコンパイルはプログラムの実行に関しては，実行速度の向上とスタックの消費量を小さくするという効果をもたらす付加的

機能であると言うことができる。

KL0 コンパイラによるコンパイルはクラス毎に行なわれる。従って，複数のクラスで定義される述語はあるクローズは解釈実行され，別のクローズはKL0 のコードで実行されるという場合がありうる。

KL0 コンパイラの第２の機能は，タームからインタプリタの処理する解釈実行コードを生成することである。ESP コンパイラは，ソース・プログラムを読込み，マクロ展開を行なった後，個々のクローズをまとめて述語を作る。この時の処理は全てタームに対して行なわれ，KL0 のコードや解釈実行コードにはなっていない。ここでまとめあげられたクローズ群は，個々のクローズがヘッドとボディを持つタームであり，述語はクローズのPrologリストである。従ってこれらは全てスタック上に存在し，ヒープ上のコード実体にはなっていない。ESP コンパイラは個々のクローズをKL0 コンパイラに渡し，KL0 コンパイラが解釈実行コードに変換する。

KL0 コンパイラの第３の機能は，ライブラリが必要とする一部のKL0 コードを生成することである。これは，コンパイル済コードからインタプリタを呼出すコードと，例外処理を要求するコード（exception をraise するコード）がある。ライブラリの処理に伴って，存在しなくなった述語や，まだコンパイルされていない述語へのリンク・ポインタを扱う時にこれらのコードが必要となる。

5．コンポーザ

コンポーザは，テンプレートに格納された個々のクラスの単体情報をもとに，１個の実行可能なクラス・オブジェクトを作成する。これによってクラス・オブジェクトが作成され，インスタンス・オブジェクトはこのクラス・オブジェクトに新規作成要求(:new)を発行することによって作成される。

実行可能なオブジェクトの作成のためには，継承・参照しているクラスへのリンク・ポインタの設定の他に，全ての継承を解析してできあがったメソッド述語の呼出しコードの作成，及び，全クラスに共通して存在するメソッド述語（クラスclass のメソッド）のコードの付加が必要である。

このうち，継承解析後のメソッド述語の作成は，そのメソッドのデモンの組合せと主処理の組合せを１つの述語として呼出すコードの作成であり，この呼出しコードのタームを作成してKL0 コンパイラによって作成する。このメソッド述語はライブラリに登録されたコードであり，テンプレートには含まれないものである。このコードはKL0 コンパイラによって作成されるので，このメソッド述語を解釈実行コードとするかコンパイル済コードとするかはコンポーザが選択することができる。これはユーザからの指定によって，どちらにすることも可能であり，実行コード本体はコンパイルされているが，それを呼出すメソッド述語は解釈実行コードであるといった組合せが可能である。

6．最適化

コンパイラの行なう最適化は各フェーズにそれぞれ分担される。ESP コンパイラの部分では，ある述語が継承によって影響されるかどうかを判定し，その情報を後のフェーズに渡す。これによってコンポーザでは余分なメソッド呼出しコードを作らずにすむ。メソッド述語自体を作らずにすむ場合もある。

KL0 コンパイラでは，継承に無関係な部分の最適化を行なう。現在行なっている最適化は，tro (tail recursion optimization) のための変数の分類とclause indexing 及び若干のpeephole最適化がある。tail recursiveな呼出しによってlocal stack を節約できることはよく知られており，DEC-10 prologではmode宣言によってこれをより効率化している。KL0 での実行はファームウェアで行なわれており，DEC-10 prologよりも複雑な実行制御が可能となっているので，変数の分類の条件もそれによって複雑化している。また，mode宣言がないため変数のlocal 化が難しい。KL0 コンパイラでは組込述語の引数の入出力関係に着目して変数のlocal 化の効率を向上させた。

clause indexing はファームウェアのレベルでサポートされているので，コンパイラは対応するコードを出せば良く，実行速度の向上に役立っている。

7．おわりに

SIMPOSの実行管理系のうち，コンパイラについて報告した。コンパイラは，昨年12月にリリースされ，φの実機上で稼働中である。今後さらに改良を加え，より完成したものにしていく予定である。

参考文献

［近山　84］　Unique features of ESP, Proc. FGCS'84, ICOT, 1984, pp.292-298.

［萩尾　85］　SIMPOSのプログラミング・システム－ライブラリー，情報処理学会第30回全国大会，1985.

SIMPOSのプログラミング・システム
ーーライブラリーー

4E-6

荻尾　弦一郎　　小山　幸久　　小長谷　明彦　　高木　茂行　　近山　隆
（ビーコンシステム㈱）　　（日本電気㈱）　　（(財) ICOT）

1．はじめに

SIMPOSは，逐次型推論マシンのための，プログラミング／オペレーティング・システムである．

SIMPOSは，対象指向機能を論理型言語に融合したプログラム言語ESPで記述され，クラスを基本単位として構成されている．ライブラリ・サブシステムは，SIMPOSで定義されたクラス，ならびに利用者が定義したクラスの管理を行う．

2．背景

クラス・システムにおいては，各クラスは，継承と呼ばれる関係において他のクラスの性質を受けつぎ，又，単に他のクラスに対しメソッドを送るために，他のクラスの参照を行っている．新規クラスは，基本的な性質を持つクラスを継承し，それに，必要な機能の付加や修正を行って作成される．

この様なシステムにおいては，存在する全てのクラスを管理し，必要な性質を持つクラスの検索を行ったり，新規クラスを作成する時に継承解析を行うためのサブシステムが必要となる．

ライブラリ・サブシステムは，そのような要求を満たすために全てのクラスを管理し，各クラスに関する情報を保持すると共に，クラスに対する種々の操作を提供している．以下において，ライブラリ・サブシステムのクラス管理方式について説明する．

3．クラスの表現形式

ライブラリ・サブシステムは，個々のクラスを以下の3種の表現形式に分けて管理する．

a）　ソースは，ユーザがエディタにより作成したプログラムの内部形式である．これは，ターム形式の構造で表現される．

b）　テンプレートは，クラス単体に関する情報を持つ．具体的には，継承クラス名，参照クラス名，スロット情報，及び，ソースで定義されている述語のインタプリティブ・コードとマシン・コードである．

c）　オブジェクト記述子は，実行に直接必要なクラス・オブジェクトと，継承クラス名，参照クラス名，被継承クラス名，被参照クラス名をもつ．これらのクラス名の情報は，クラスが修正された時に，関連クラスの修正のために，使用される．

4．クラスの表現形式の変換

ライブラリ・サブシステムは，クラスの3種の表現形式の変換を行い，はじめに存在するソースより，最終物として，クラス・オブジェクトを作成する．

ソースをコンパイルすることにより，テンプレートを作成する．このときは，ソースに記述してある情報のみを使用してテンプレートを作成し，継承の解析は行わない．

テンプレートを継承解析することにより，クラス・オブジェクトを作成する．このときは，全ての継承するクラスのテンプレートが継承解析のために使用され，対象指向呼出しの実行のために最適化されたクラス・オブジェクトを作成する．

5．構成

本システムのクラス管理の構成を図－1に示す．

ライブラリ・サブシステムは，クラスの情報を保持するプール（オブジェクト格納のための容器）としてライブラリを持つ．

ライブラリは，登録されたクラスに1対1に対応するクラス記述子を持ち，個々のクラスの情報は，クラス記述子の中に，前述の3種の表現形式により保持される．

クラスの3種の表現形式は，対応するファイルへの格納形式を持つ．このため，対応するファイルが作成された後では，コンパイルや継承解析を行わずに，テンプレート・ファイルやオブジェクト・ファイルから直接，テンプレートやクラス・オブジェクトを作成できる．

ライブラリに登録されたクラスは，必ず対応するクラス記述子とオブジェクト記述子を持つが，テンプレートやクラス・オブジェクトは，必要に応じてファイルからロード，又は，新規に作成される．

6．クラス管理の自動化

以上のようなクラス管理を行うにあたって，ライブラリ・サブシステムは以下のような自動化を行っている．

a）　関連クラスの自動ロード処理．

一つのクラスを実行するためには，そのクラスのみではなく，そのクラスが直接・間接に使用するクラスを全てロードしなくてはならない．

本システムでは，クラスがもつ継承・参照情報を使用し，与えられたクラスの実行に必要なクラスのロードを自動的に行う．

b） 関連クラスの自動更新．

ソース・ファイルの修正が行われた場合，それに伴い，クラス・テンプレート，及び，クラス・オブジェクトを修正する必要がある．SIHPOSでは，対象指向呼出しの高速化のためにクラスの継承の解析を静的に行なう，そのためクラスの修正に際しては，そのクラスを継承する子クラスの継承解析を再度行う必要がある．従って，クラス・オブジェクトについては，修正したクラスのみではなく，その子クラスのクラス・オブジェクトも修正しなくてはならない．

本システムでは，各ファイルの，作成又は更新日時をバージョンとして持ち，そのバージョンの比較により，修正の自動化を行っている．

即ち，テンプレートを作成する時は，ソース・ファイルとテンプレート・ファイルのバージョンの比較を行い，ソース・ファイルが新しい場合は，ソース・ファイルをコンパイルして，テンプレートを改めて作成する．

同様に，クラス・オブジェクトを作成する時は，自分自身及び親クラスのテンプレートのバージョンと，オブジェクト・ファイルのバージョンの比較が行われる．テンプレートに新しいものが存在するときは，継承解析により，クラス・オブジェクトを改めて作成する．

7．ライブラリアン

ライブラリ・サブシステムは，全てのクラスを管理するため，ユーザは，ライブラリアンと呼ばれるユーザ・インタフェースを通じて，各種のクラスの情報を取り出すことができる．

クラスの情報の一部を以下に示す．

- メソッド名／引数個数．
- スロット名．
- 参照・被参照クラス名．
- 継承・被継承クラス名．

この機能により，例えば，以下のようなことが可能になる．

- あるクラスで使用可能なメソッドを，全て求める．
- 特定メソッドを持つクラスや，特定クラスを継承するクラスを求める．
- クラスを修正する場合に，その影響をうける被継承クラス，被参照クラスを求める．

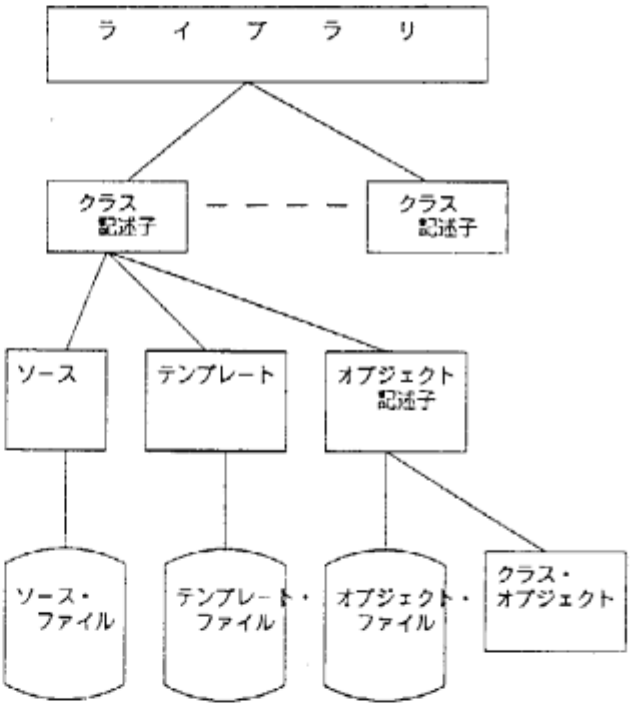
このような機能により，必要な機能を持つクラスの検索や，クラスの修正に対する影響のチェックを，容易に行うことができる．

8．おわりに

SIHPOSのライブラリ・サブシステムにおける，クラスの管理方式について述べた．現在，ライブラリ・サブシステムは，一部を除き，ＰＳＩ上で稼動中である．今後の課題としては，システムの拡充，及び，ユーザ・インターフェースの改良等が残されている．

参考文献

1． [近山 84] Unique Features of ESP.
Proc. of FGCS'84, ICOT, PP. 292-298,1984 (英文).



図－1 ライブラリ・サブシステムの構成．

ＳＩＭＰＯＳのプログラミング・システム

―― デバッガ／インタプリタ ――

石橋 弘義
（松下電器産業（株））

近山 隆
（(財) ＩＣＯＴ）

１．はじめに

逐次型推論マシン（ＰＳＩ）上に開発されたプログラミング／オペレーティングシステム（ＳＩＭＰＯＳ）の“デバッガ／インタプリタ・サブシステム”について、その基本概念と実現法について述べる。

２．概要

デバッガ／インタプリタ・サブシステムは、次の２つの部分から成る。

①ＥＳＰプログラムを解釈実行する“インタプリタ”

②プログラム実行時の動的な情報を表示し、ユーザとの応答によりプログラムのデバッグを行う“デバッガ”

本サブシステムには次の機能がある。

①インタプリティブ・コードの実行　(interpreter)

②コンパイル済コードの呼出し　(interpreter)

③プログラム実行時のデバッグ情報の提供　(debugger)

④プログラムの実行制御と繰り返し試行　(interpreter + debugger)

これらの基本機能は DEC-10 Prologのデバッガと同じである。 以下に主な相違点を挙げる。

①デバッグの単位をクローズ単位にすることにより、より詳細な情報の提供や実行制御を可能にした。

②マルチウィンドウを用いることにより、高度なユーザ・インタフェイスを実現した。

３．インタプリタ

(1)インタプリティブ・コードとコンパイル済コード

ＥＳＰプログラムのオブジェクト・コードとしては、コンパイル済コードとインタプリティブ・コードの２つがある。 コンパイル済コードは、解釈プロセスなしに直接実行されるので高速に実行できる。 しかし、インタプリタではコンパイル済コード全体を１つの述語のように扱うので、デバッガによるトレースやダイナミックなプログラムの変更は出来ない。 これに対して、インタプリティブ・コードはデバッグ情報の収集に適した構造をしており、実行時のトレースやダイナミックなプログラムの変更も可能である。

また、ゴールを実行する場合に、そのゴールのコードがインタプリティブ・コードであってもコンパイル済コードであっても次のように、同じ呼び出し形式でよい。

```
:interpret( #interpreter, :goal(#example,Args) );
```

(2)インタプリタの基本方式

実行速度を上げるためにできるだけハードウェア，ファームウェアを利用し、ソフトウェアではデバッグ情報の収集と実行制御や繰り返し試行が柔軟に出来るようにした。例えば、ユーザ・プログラム中の ",", ";" の処理は次のようになる。

```
interpret((X,Y)):-!, interpret(X), interpret(Y);
interpret((X;Y)):-!,(interpret(X) ; interpret(Y));
interpret(Pred) :-!, execute(Pred);
```

この場合、実際の ",", ";" の実行はファームウェアに任されている。 また、KL0 組込述語の実行は次のようにすればよい。

```
execute( unify(X,Y) ) :-!, unify(X,Y);
execute( first(X,Y) ) :-!, first(X,Y);
execute(  ……     ) :-!,  ……   ;
```

なお、ユーザ定義述語の実行メカニズムに関してはここでは省略するが、述語コードの管理はライブラリ・サブシステムが行っている。

(3)on_backtrack, bind_hook 等の実行制御

これらのＥＳＰ固有の実行制御は次のようになる。

```
execute( on_backtrack(X) ) :-!,
        on_backtrack( interpret(X) );
execute( bind_hook(X,Y) )  :-!,
        bind_hook(X, interpret(Y));
```

つまり、on_backtrackやbind_hook の実行は、その引数のゴール列をインタプリットすればよい。

(4)マルチレベル・カットのインプリメント

インタプリタでマルチレベル・カット等を正しく動作させるには、実行中のユーザ・プログラムのレベルを管理する必要がある。 つまり、インタプリタはユーザ・プログラムの呼び出しが深くなる毎に、インタプリット中の実際のレベルをスタックに積む。 そして、インタプリティブ・コードで実行中のユーザ・プログラムでは、仮想レベル(pseudo level)が使われる。 この値はユーザ・プログラムの呼び出しの深さと同じであり、コンパイル済コードで実行中のレベルとは異なる。 しかし、ユーザ・プログラムでは仮想レベルを意識する必要はない。

これに関係する組込み述語としては "level","absolute_cut","relative cut","succeed" 等がある。

4．デバッガ

(1)P&Cモデル

Procedure & Clause Box Control Flow Model（P&Cモデル，図1）は，DEC-10 プロローグの Procedure Box Control Flow Model（Pモデル，図2）を拡張したものである。 つまり、実行の最小単位をプロセジャ単位からクローズ単位に変えた。 このモデルを採用することにより、ユーザの要求に応じてより細かいトレースや実行制御が可能になった。 この拡張はＥＳＰ特有のものではなくプロローグにも応用できる。

以下で各ゲートの説明をする。

call：プロセジャの最初の呼び出しを表わす．このゲートの実行はユニファイ可能なクローズが存在するかどうかに無関係である．

unify：クローズ・ヘッドのユニフィケーションの試みを表わす．このゲートの実行時にクローズを表示すれば、このプロセジャの全クローズを見ることができる．

pick：クローズ・ヘッドのユニフィケーションの成功とクローズ・ボディの実行の始まりを表わす．

exit：クローズの成功、つまりプロセジャの成功を表わす．

redo：バックトラックによるプロセジャ，クローズの再実行を表わす．

miss：クローズ・ボディの失敗を表わす． ESP/KLO ではヘッドのユニフィケーションに別解があり、再び同じクローズが pick される可能性もある．

next：クローズの失敗を表わす．この後、他の選択肢があればそのクローズのユニファイを試み、なければそのプロセジャは失敗する．

fail：プロセジャの失敗を表わす．つまり、このプロセジャを呼び出したクローズにバックトラックする．

(2)マルチウインドウ・インタフェイス

高度のユーザ・インタフェイスを提供するために、デバッガはマウスとマルチウインドウ・システムを使用する。トップ・レベルのデバッガ・ウインドウは最小限のトレース情報を出力するだけで、他の情報はそれぞれ別のサブウインドウを使う。 これにより必要なウインドウのみを表示しながら作業ができる。 例えば、デバッガのゴール入力時にユーザが使った変数の結果などはデバッガ・サブウインドウに表示される。 また、デバッガの内部状態はステイト・ウインドウにあり、状態の変更などもこのウインドウからマウスによるメニュー選択で行う。 さらに、デバッガ・コマンドもメニュー選択で入力できる。

(3)デバッグ機能

プログラムの実行制御用コマンドをあげる。

①ステップ実行
②次のスパイ・ポイントまで中断なしで実行
③述語やクローズ単位でスキップ
④中断なしで実行
⑤述語，クローズの再実行や失敗
⑥指示した所まで制御を戻す

5．デバッガ／インタプリタ・サブシステム

本サブシステムの実行手順は以下のようになる。

①ユーザはデバッガに対して解釈実行を要求する．
②それに従って，デバッガはインタプリタに解釈実行を要求する．
③インタプリタは解釈実行中、各ゲート毎にデバッガにデバッグ情報を渡す．
④デバッガは各ゲートで中断条件に合うかチェックし、それに従ってユーザに応答を求め、インタプリタに実行制御を指示する．
⑤インタプリタは指示に従って解釈実行を続ける．

6．おわりに

現在、ＳＩＭＰＯＳのプログラミング・システム自体ユーザに使われ始めたばかりであり、まだ使い易いシステムとは言い難い。 今後、ユーザからのフィードバックを取り入れてよりよいものにしていく予定である。

参考文献

D.L.Bowen, L.Byrd, F.C.N.Pereira, L.M.Pereira, D.H.D.Warren： DECsystem-10 PROLOG USER'S MANUAL (Dept. AI., Univ. of Edinburgh , 1983)

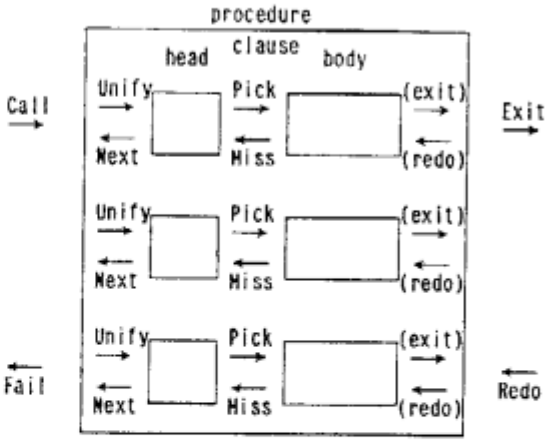


図1．Procedure & Clause Box Control Flow Model

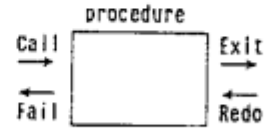


図2．Procedure Box Control Flow Model

ＳＩＭＰＯＳのプログラミング・システム －トランスデューサー

4E-4

坂井　公　（（財）ｉＣＯＴ）
堀　敬史　（（株）三菱総合研究所）
佐藤　裕幸　（三菱電機（株））

１．はじめに

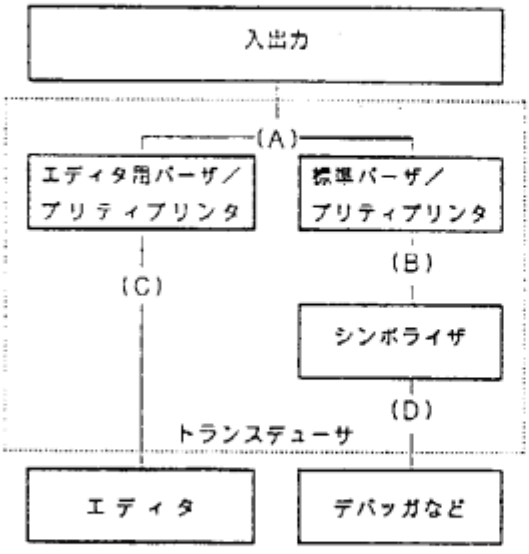
パーソナル逐次型推論マシン(PSI) のプログラミング・システム(SIMPOS)のうち、トランスデューサの概要について述べる。トランスデューサは、PSI 内部のデータ構造とその表示イメージとの変換を行なうユーティリティで、ユーザが定義した文法に従って、構造を持ったデータを構文解析したりプリティプリンティングしたりする。

２．構成と機能

図－１の破線に囲まれた部分が、トランスデューサの主な構成要素である。各構成要素は次のような変換を行なう。

- 標準パーザ／プリティプリンタ…表示イメージと構造化データ間の変換
- エディタ用パーザ／プリティプリンタ…表示イメージとエディタ用構造化データ間の変換
- シンボライザ…構造化データとPSI 内部表現間の変換

従って（Ａ）、（Ｂ）、（Ｃ）、（Ｄ）は、それぞれ、表示イメージ、構造化データ、エディタ用構造化データ、PSI 内部表現である。それぞれがどのような形式のデータであるかを図－２に具体例によって示す。

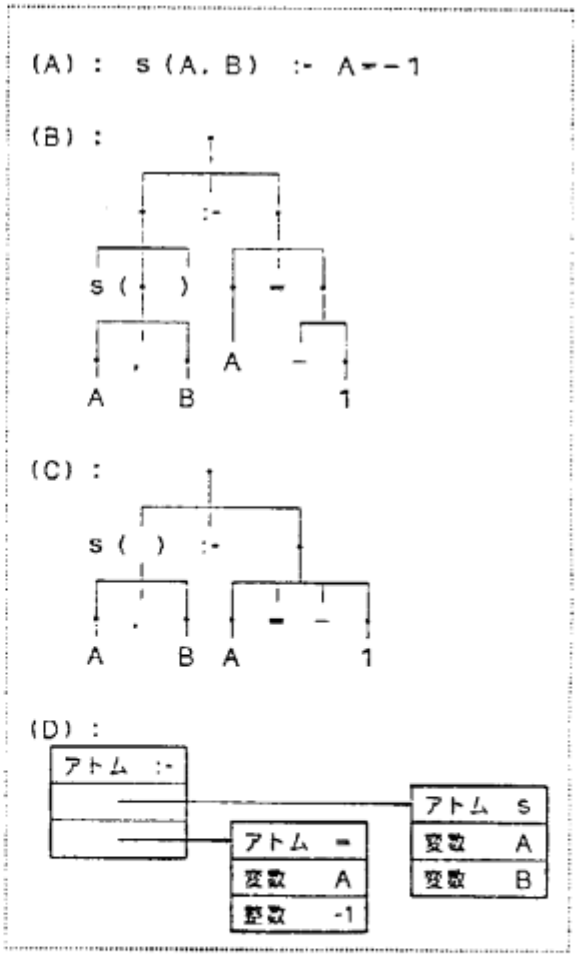


図－１　トランスデューサの構成

３．文法

ユーザ定義の文法により構文解析を行なうシステムとして提案されているものは、文脈自由文法に基づいたものが多いようであるが、我々は、PSI のシステム言語であるESP との親和性とエディタ・コマンドとの対応のとりやすさから、演算子順位文法を採用した。文法は構文要素を定義する部分と演算子順位を定義する部分からなる。

構文要素(token) の定義には、BNF を用いる。構文規則の右辺には正規表現が書けるようになっているが、再帰的



図－２　データ表現

規則は許されないので、定義できる構文要素は正規集合になる。 浮動小数点データを表現する構文要素の定義を例として挙げる。

```
digit ::= "0" | "1" | … | "9"
real ::= digit, {digit}, ".", {digit}
      ["^","+" | "-"], digit, {digit}
```

右辺で正規表現のために使用できる記号とその意味は次のとおりである。

−	（負符号）	… 差集合
,	（コンマ）	… 連接
|	（縦線）	… 和集合
" … "	（引用符で囲む）	… 終端記号列
[…]	（大括弧で囲む）	… 省略可
{ … }	（中括弧で囲む）	… 0個以上の繰返し
(…)	（小括弧で囲む）	… 補助記号

構文要素は、下記の6種類に分類される。

- アトム…定数、変数など単独で意味を持つデータ
- 前置演算子…負符号の "−" など引数の前に置かれる1項演算子
- 間置演算子…減法の "−" など2つの引数の間に置かれる2項演算子
- 後置演算子…階乗記号 "!" など引数の後に置かれる1項演算子
- 左括弧… "(", "begin" など
- 右括弧… ")", "end" など

これらのうち、上の4つのグループ間には、重なりがあってもよいが、左括弧や右括弧として定義されたものが他のグループにも属している場合は、構文解析の結果は保証されない。

前置、間置、後置各演算子には、引数との結合に関する優先順が定義される。DEC-10 Prolog などでは、各演算子に自然数を割り当て、その大小で優先順を定めるが、これは人間には直観的にわかりにくいので、我々は次のような記法を採用した。

```
relation>operator
(operator<operator)<relation
```

relationとoperatorは、ユーザが定義した構文要素名である。1行目はrelationが左にoperatorが右にある場合はoperatorが先に引数と結合することを意味する。2行目は左右が逆転しても、operatorが優先することと、operator同志では左のものが優先することを意味する。

例えば、構文要素定義で "−" がoperatorに "=" がrelationになっていれば、図−2（A）のA=−1は（B）のように解析される。

エディタを使用している時は、人間は、論理的な構造よりも、目に見える構造に着目しがちである。又、プリティプリンティングを行なう場合、深い構造を忠実に表現しようとすると大きなスペースを必要とする。そこで、演算子順位は、エディタやプリティプリンタには一般に無視されるようにした。エディタやプリティプリンタにも意識させたい優先順を指定するには ">", "<" の代りに ">>", "<<" を用いる。

例えば、

```
logicalsymbol>>relation
```

と定められていて、":-"が logicalsymbolであれば、図−2（A）はエディタ用パーザによって（C）のように解析される。

4．内部表現との変換

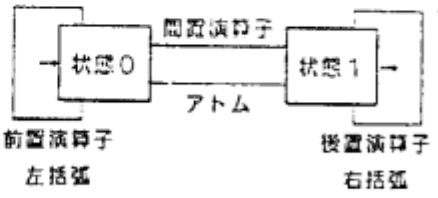
図−2の（B）と（D）の間の変換は、一定の方式で行なう。この変換文法もユーザに定義させることが考えられるが、ユーザがPSI の内部構造に精通する必要があり、文法を書くこと自体複雑な作業になりそうなので、そのような作業をするだけの利点があるかどうか疑問もあり、現在は見あわせている。

5．インプリメンテーション

パーザについて簡単に述べる。文字列を構文要素に分割することは、構文要素定義を読み込んだ時に生成した有限オートマントンを用いて行なう。構文要素の列から構造化データを組み上げるのは、2状態のプッシュダウン・オートマトンで可能である（実際には、エラーメッセージの出力や演算子順位の処理のために数状態を持つ）。スタックの動きは無視して、状態遷移だけを図−3に示す。

6．おわりに

トランスデューサには、外にキャラクタ・シートなど重要なコンポーネントがあるが、それらについては別の機会に述べたい。



図−3 プッシュダウン・オートマトン

ＳＩＭＰＯＳのプログラミング・システム
－　エディタ　－

4E-3

佐藤　裕幸　（三菱電機（株））
堀　敦史　（(株）三菱総合研究所）
酒口　徹夫　（三菱電機（株））
坂井　公　（(財）ＩＣＯＴ）

1．はじめに

我々はProlog高速実行マシンである逐次型推論マシンψ(PSI) のプログラミング／オペレーティング・システムSIMPOSを開発中である。　本稿では、SIMPOSのプログラミング・システムの一つであるエディタ(Edips) の特徴及び機能について報告する。

2．Edips の特徴

Edips は以下のような特徴を持っている。

○マルチ・ウィンドウを使用したスクリーン・エディタである

マウスによるカーソルの指定、メニューによるコマンドの指定やテキストの表示領域，コマンドのエコーバック領域等をマルチ・ウィンドウによって分割している。

○構造エディタである

編集されるデータの中に構造を持ち込んだ構造指向型エディタである。これにより以下のような利点が得られる。

★人間の考えている論理的な単位と編集操作の単位との一致を計れる

★従来のエディタでは、文字，単語，行等のように構造の大きさ毎に類似のコマンドをたくさん持つことがあったが、それらを一元化できる

★編集対象を巨視的に眺めたり、微視的に眺めたりするホロフラスティング機能を利用できる

○ユーザが構文規則を定義できる

テキストをEdips が扱える構造に変換するのはトランスデューサ[1] が行っているが、この規則をユーザが定義できるようになっている（詳しくは[1] を参照）。従ってEdips は汎用構造エディタとなっている。

○表層構造を重視したテキスト・エディタである

人間の編集作業は、本来持っている構造（深層構造）よりもそれを表現しているテキストとしての構造（表層構造）に着目して行われる場合が多い。

○extensibleである

Edips のコマンドの数は必要最少限になっており、ユーザがマクロにより自分の好みに合わせて新しいコマンドを作ることができる。

3．構造編集

Edips は通常、しかるべき構文規則にのっとった構造を持ったデータとして編集対象を扱っており、常に構文規則に適合したものを編集対象として保持する。　ところが、ユーザが編集対象に対して削除や移動などの操作を行うと文法的に正しくない状況に陥る場合がある。　Edips ではそれを避けるために操作が行なわれた編集対象の前後の要素を同時に削除や移動することがある。

Edips における編集操作はすべてターゲットと呼ばれる領域に対して行われる。　ターゲットとは一般のエディタにおけるカーソルに対応するが、連続した複数の構文要素をターゲットとすることも可能である（マルチ・ターゲット）。　今、図１－(A) のようなテキストがあるとする（▒…ターゲット）。　これは図１－(B) のような構造をしている（表層構造であることに注意）。　ここで単純に削除を行うと図１－(C) のように文法的に正しくない状態になってしまう。　そのためEdips では、　＊　もいっしょに削除してしまい図１－(D) のような結果となる。　これは右方向へ削除する例で、左方向への削除もあり、このときは図１－(E) のようになる。　そのほか、複写，移動についても同様な機能を持っている。

これらの機能は慣れないと思わぬ動きをしてしまうことがあり戸惑うかもしれない（もちろん　元に戻すコマンドundoは用意されている）。　しかし、少し慣れてくるとこの機能をうまく利用すれば操作を著しく減少させることができるであろう。

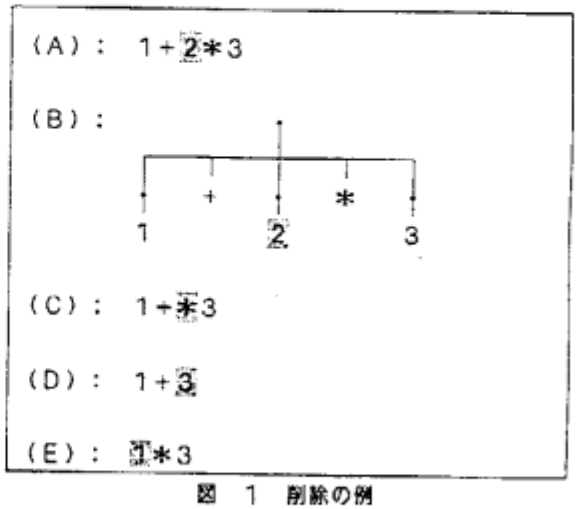


図　1　削除の例

4．文字列編集

3章で示したような構文規則にのっとって編集を行う操作モードを構造編集モードという。 しかし、テキストを入力したり、一時的に文法からはずれるデータ構造を作ったほうが編集効率がよい場合がある。 Edips ではこのことを考慮にいれて、構造を無視した、ただの文字列としての編集が可能であり、このような操作モードを文字列編集モードという。

いわゆる構文エディタのテキスト入力方式は、テンプレート方式とパージング方式に分けることができる。 テンプレート方式では、次に何を入力すべきかをエディタがユーザに指示をするため柔軟性に欠けることもあり、Edips ではパージング方式を採用した。 Edips で編集作業中に文字を入力すると文字列編集モードになり、テキストの入力が終り構造編集モードに戻ると入力されたテキストがトランスデューサによりパージングされる。

5．マクロ・コマンド

ユーザは多用する一連の操作を束にして、一つのコマンドとして取扱うことができる。 このような束として定義されたコマンドをマクロ・コマンドという。 マクロ定義はESP やPrologのプログラムと同様の構文によって記述される。 組込みの述語としてエディタが基本として持っているコマンドのうちundoやマクロ定義コマンド，モード切換コマンド等のメタコマンドを除いたものを記述できる。

例 アトム'aaa' を渡しユーザが入力したもので置換えることを 'aaa' がなくなるまで繰り返す。

```
aaa_replace:-
   search(´aaa´), !,
   replace(@),
   aaa_replace.
aaa_replace.
```

@はマクロ実行中にユーザが入力する引数を表わしており、

```
replace(@)
```

は

```
read(X), replace(X)
```

という一連のコマンドのマクロと考えることもできる。

6．モジュール構成

Edips 及びその周辺の構成を図2に示す。 パーサ，プリティプリンタはトランスデューサの一部で、それぞれ文字列と構造（ターム）との相互変換を行っている。 キーボードやマウスからの入力はウィンドウから送られ、パーサによりタームに変換される。 編集作業はエディタ本体が内部的に保持している構文木を操作することにより行われる。 キャラクタ・シートとは、編集中のテキストを文字列として保持しているシートである。 構文木の変更された部分を文字列としてキャラクタ・シートに送ってやることによりウィンドウの表示が変わる。 つまり画面管理は、すべてキャラクタ・シートが行っていることになる。

文字列編集モードの時は構文木での編集は行わず、直接キャラクタ・シートを操作することにより編集作業を行っている。

7．まとめ

現在Edips は、文字列編集モードに対応する部分が稼動中でなんとかPSI 上でテキストの編集が可能であるという状態である。 今後は、構造編集モードの早期実現を行い、その後実際に使用してみて、仕様の改良ならびに再検討を行う予定である。

＜参考文献＞

[1] 坂井他：SIMPOSのプログラミング・システム
－ トランスデューサ －，本大会4E－4

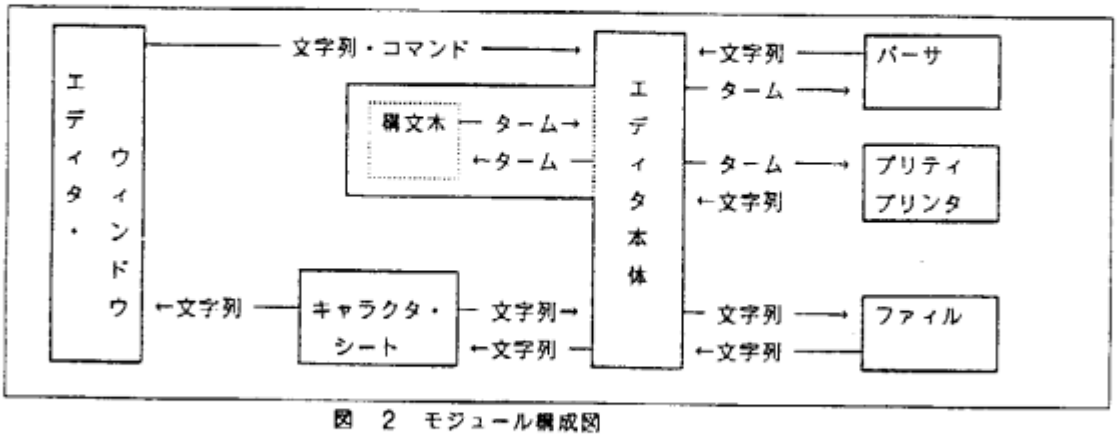


図 2 モジュール構成図

SIMPOSのプログラミング・システム
— コーディネータ —

4E-2

東条 敏 (三菱総合研究所(株))
清口 徹夫 (三菱電機(株))
黒川 利明 ((財)ICOT)

1. はじめに

我々は逐次型推論マシン(PSI)のプログラミング/オペレーティング・システム(SIMPOS)を開発中である．ここではそのプログラミング・システムという環境をサポートするコーディネータについて説明する．

2. 基本概念

ユーザから見たSIMPOSのプログラミング・システムとは，エキスパート・プロセスの集合である．エキスパート・プロセスとは，独立の通信ウインドウ(e_ウインドウと呼ばれる)を持っているプロセスである．エキスパート・プロセスはユーザ要求に対しエキスパートとしての特定の処理を実行する．

SIMPOSではUNIXにおけるShellのような明示的な監視プロセスはない．代わりに，暗にコーディネータと呼ばれるプロセスを置き，エキスパート・プロセスの集合を管理する．

通常ユーザはe_ウインドウを介してエキスパート・プロセスと対話を続けるが，必要に応じてこのe_ウインドウから，もしくはバックスクリーンからコーディネータにコマンドを送り，次節以降に述べるその機能を運用させる(図1)．

3. コーディネータの機能

3.1. エキスパートの管理

コーディネータは現在SIMPOS上で動いているエキスパート・プロセスのをプールにして持っている．すなわち，新にエキスパート・プロセスを生成，消滅するたびにこのプールへ登録・抹消を行う．

コーディネータはユーザから，e_ウインドウを通じて，または後述するシステム・メニューからコマンドを受けとって，以下の処理を行うことができる．

・プロセスの生成

システム・メニューよりエキスパート・プログラム名を選択して，そのプロセスを作る．できたプロセスは命名されてコーディネータのプールに登録される．

コーディネータはプロセス生成と同時にe_ウインドウをつくる．すなわち最初ユーザにマウスを使ってe_ウインドウの位置と大きさを決めるよう促す．

・プロセスの終了

・プロセスの訪問

下に隠れてしまったe_ウインドウのプロセスを訪問する．

・プロセスのサスペンド及び再起動

・プロセスの訪問履歴管理

スタックがひとつ用意されており，そこに随意にエキスパート・プロセス訪問の履歴をつけることができる．ここに，後に戻ってきたいプロセスを指定することができる．

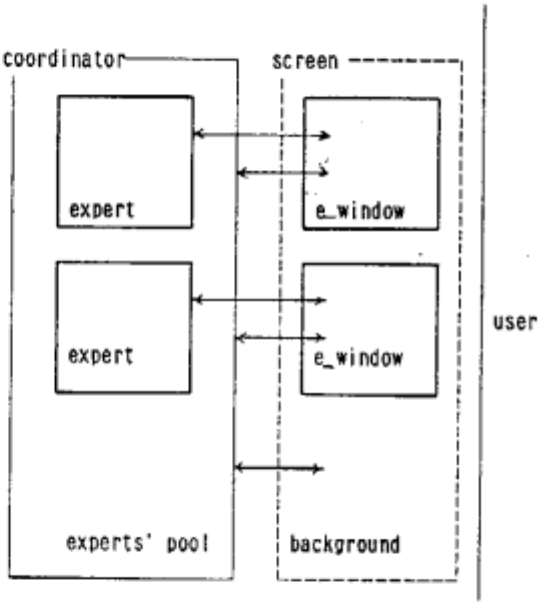


図1．ユーザとエキスパートの対話

3.2. エキスパート・プロセス間通信

コーディネータはエキスパート・プロセス間の通信に関与する．

3.2.1. e_ポート

コーディネータはコーディネータ・プロセスと各エキスパート・プロセスとの間に専用ポートを設け，オブジェクトの交換を可能にする．このポートをe_ポートという．

基本的にe_ポートの活用のしかたはプロセス側の自由であるが，次のような用法が考えられる．

・コマンドのバイパス送信

もしプロセス側に通常のウインドウ入出力ではなく，e_ポートの回路からコマンドの割り込み処理ができるようにしておけば，本来プロセス内部で処理される

コマンドをコーディネータ経由で送信して，優先処理が可能になる．

・プロセス側からコーディネータへのコマンド

e_ポートの回線を今度は逆向きに使って，コーディネータにコマンドを送ることも可能である．また，コーディネータを通じて他のプロセスへコマンドを送ることも可能である．コーディネータの持っているポートは一つなのでどこから来てもウインドウから入ってきたのと全く同様な処理が行われる．

3．2．2．ホワイトボード

原理的にはエキスパート間の通信問題はe_ポートによって解決がつくはずであるが，前述したとおり，e_ポートの用法というのは，多分にソフィステケートされたものである．ごく通常のプロセス間通信，たとえば単なるオブジェクトの交換等に関する限り，送るオブジェクトと送信相手をユーザからもう少し容易に指示できる手段が望まれる．

異なるプロセス間でオブジェクトを引き渡すため，ホワイトボードというエリアが解放されている．あるエキスパートがここに他のエキスパートに対するメッセージを書き込み，相手のエキスパートはユーザ命令によりそのメッセージを受け取る（図2）．

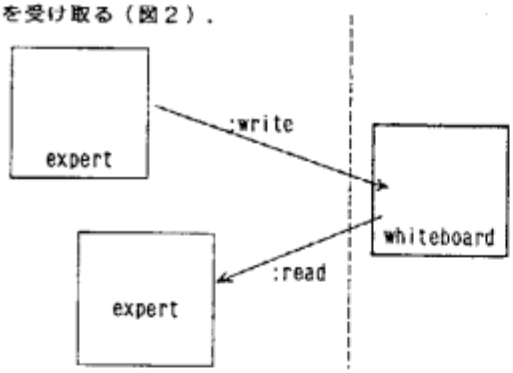


図2．ホワイトボード

活用のしかたとして，たとえば，エディタであるテキストをエディトした結果を，インタープリタへのメッセージとしてホワイトボード経由で渡して，そこでそれを実行させるというようなことも可能になる．

3．3．システム．メニュー

コーディネータは，エキスパート・プログラムを含めてSIMPOSの他のシステム・プログラムとの仲介をするためにシステム・メニューを提供する．ユーザはシステム・メニューの項目を選択することにより，エキスパート・プロセスの生成，選択，終了等の他，ウインドウ・マニピュレータの呼び出し，ログイン処理等，直接自分のプロセスを作らないシステム・プログラムも実行させることができる（図3）．

system menu		
[experts]	[media systems]	[etceteras]
debugger	window-manipulator	login
editor	network-manager	help
file-manager		

図3．システム・メニューの例

ログイン処理では，前のユーザのエキスパート・プロセスをすべて終了させ，新しいユーザの持っているエキスパート・プログラムのディレクトリをみて，そのユーザ固有のシステム・メニューを作る．

4．ウインドウ・システムとのインタフェース

（1）トランスレーション・テーブル

通常ユーザがエキスパートと対話をするのに用いられるe_ウインドウには，コーディネータへの入力コマンドの表がセットされている．すなわち，ある特殊キー／マウス入力に対しては，そのキー／マウス・コードはそこのエキスパートには渡らず，コーディネータに渡って，コマンドとして処理される．この表は，キー／マウス入力をどちらのプロセスに渡すのか振り分け，場合によりコマンドを生成することから，トランスレーション・テーブルと呼ばれる．テーブルはウインドウに固有なものとして，ウインドウごとにその内容を変えることができる．

（2）ウインドウ・マネージャ・アクセッサ

コーディネータでは，特にウインドウ・マネージャと密接にインタフェースをとるアクセッサを設ける．ここではウインドウからのコマンドを受信／送信するチャネルがあり，ウインドウ入力が優先して処理されるようになっている．

5．おわりに

SIMPOSのプログラミング・システムの核としてのコーディネータを，その機能を中心に説明した．但しコーディネータの機能仕様を現段階で固定して考えるべきではなく，今後ユーザの使い勝手等の評価を待ってその機能向上がなされていく必要がある．

参考文献

Kurokawa,T. and Tojo,S., "Coordinator - the Kernel of the Programming System for the Personal Sequential Inference Machine (PSI)", ICOT TR-061 (1984).

SIMPOSのプログラミング・システム

— BUPのPSIへの移植経験 —

奥西 稔幸
（シャープ(株)）

平川 秀樹
（(財)ICOT）

1. はじめに

ICOTでは，東京工業大学，電子総合技術研究所と協力して，論理型言語を用いた構文解析システムBUP(Bottom Up Parsing)システム[1] を開発してきた。BUP システムは，DCG (Definite Clause Grammar)[2]に基づき，文をボトム・アップに解析を行なうシステムであり，Dec-10 Prolog 上に開発されているが，今回，このシステムを，PSI(逐次型推論マシン) に移植することにした。

PSI は，ICOTで開発されたProlog高速実行マシンで，そのオペレーティングシステム（SIMPOS）では，マルチウィンドウ，ネットワーク，日本語処理などの高度なマンマシンインターフェイスがサポートされている。

本報告では，実際の移植作業について述べた後，PSI 上のBUP システムの評価も行ったので，あわせて報告する。

2. BUP システム

Dec-10 Prolog 上のBUP システムは，図２のようになっている。変換部は，Epsilon-reducer, Cycle-checker, Translatorからなり，CFG に基づく書き換え規則で与えられる文法規則・辞書項目を，Parserに変換する（図１）。文解析は，この変換されたプログラムで行われるが，これはPrologで記述されているため，バックトラック等の制御機構は，Prologのものが利用でき，さらに，予測制御や途中結果の登録などにより高速化も図られており[3]，効率的な処理が行なわれる。Tracerは，バージングの際のデバッガとなり，文法開発にも有用である。

しかし，これらのシステムは全てDec-10 Prolog 上で，開発されているため，記憶容量や操作性の点で，大規模な文法開発等の実用的な面に問題があったので，PSI に移植することでよりトータルで使い易いシステムとすることにした。

```
sentence --> np,vp.
        ↓
np(G,[],S0,S):-goal(vp,[],S0,S1),
               sentence(G,[],S1,S).
```

図１ 文法規則と変換結果

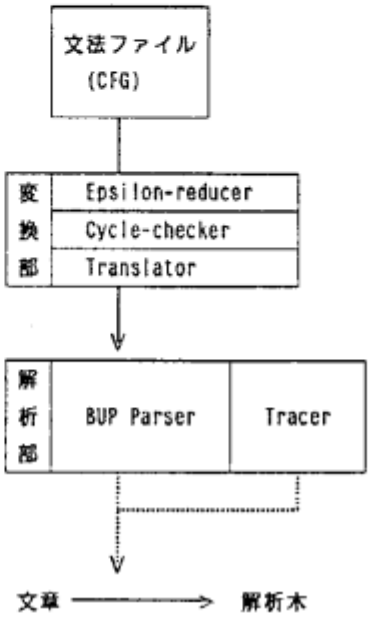


図２ BUP システム

3. PSI への移植

PSI の機械語およびシステム記述言語は，それぞれKL0, ESP[4]と呼ばれ，いずれもPrologを拡張した言語である。特に，ESP には，オブジェクト指向性が取り入れられており，モジュール性に優れている。PSI 上のユーザープログラムは，ESP で記述することになるため，Prologソースプログラムの ESP への変換が，ここでの主な移植作業となる。

以下では，BUP の各サブシステム別に，移植の際に変更を要した箇所を，幾つか

- ESP, SIMPOSの特徴をどう利用したか
- Prologの機能をESP でいかに実現したか

を中心に述べていく。

3.1. 変換部

変換部の３つのサブシステムはいずれも，「ファイルから文法規則を読み，構造的に変換または，走査した後，ファイルに出力する」という処理を行なうため，「データ構造」，「入出力処理」が，ここでの重要なポイントとなる。

PSI の構造体(list,compound-term)は全てベクトルで実現されているので，functor や=.. などのProlog組み込み述語は，KL0 のベクトル操作述語で容易に実現できた（図３）。

```
Prolog: np(a,b,c) =..[Func | Arg].
ESP:    first(np(a,b,c),Func),
        vector_tail(np(a,b,c),1,Arg-vec),
        :vector __to__list(#bup-util,Arg-vec,Arg).
```

図３　構造体操作の例

データ文字列をPSI の内部構造に変換する入出力処理は，SIMPOSの一部であるトランスデューサにより実現される。トランスデューサが提供するクラスをウィンドウやファイルとともに継承することでget,put,read,write等のメソッドが使える。BUP の場合には，PSI のベクトル記法である{}を，文法のextra-condition の記述用として用いるため，一部変更して，図４のような構造変換を行なうクラスを継承した。

```
(NP=VP)  ->  ('{}',NP=VP)
```

図４　BUP 用の構造変換

3.2. 解析部

BUP のパージング・アルゴリズムで同一計算の重複を避けるため，途中結果をassertすることにしているように，Prologでは副作用を用いたプログラミングが処理効率を上げる手法となっている。

BUP の場合には，このParserのほかに，interpreter 的な処理を行なっているTracerでもcut,fail等の操作に副作用を用いており，「副作用の実現」が必要となる。

PSI には，assert・retract 機能がなく，ESP のクラス・スロット，KL0 のヒープ・ベクトル，SIMPOSのプールで副作用が実現できる。

BUP では，「プールを利用して設けたデータベースをクラス・スロットに保持しておき，assertしたいデータをオブジェクトに変換した後，格納する」という方法で全てのassert・retract は実現できた。

以上の他にも，！（カット）の意味，call,clause の実現にも変更を要した。また，ESP のオブジェクト指向性については，デーモン，多重継承等，ESP で新たにシステムを開発するには役立つと思われる機能があるが，今回は，移植という性格からあまり用いることはなかった。なお，移植したBUP システムは約5500行で，移植に要した工程は約２人月であった。

４．PSI 上のBUP システムの評価

PSI 上のBUP システムの評価ということで，ここでは，簡単な英語の文法を用いてDec-10 Prolog 上のシステムとの比較実験を行なった。（文法数は２８）

文1:Alice chases the march-hare who has a watch.
文2:He who falls in love with himself has no rivals.

	ゴール登録による高速化	
	なし	あり
文1	5.6／21.0	85.0／93.0
文2	210／ 650	2012／ 780

（１は最初の解だけ，２は５つの解全てを求めるのに要した時間:PSI／Dec 単位:msec）

図５　PSI とDec の比較実験結果

この評価実験で用いた文法が小さかったため，途中結果の登録による高速化の方が，かえって時間がかかってしまっているが，これは，assertの部分でかなりの時間を要したためと思われる。また，その差は，PSI の方が顕著であり，PSI でのassertの実現にはまだ改良の必要があることがわかる。しかし，文２のように全解を求める場合には，確かに，高速化の効果があらわれていることがわかる。

高速化を用いなかった解析の比較では，PSI の方が４～５倍速いが，これは，パージングの際にPSI では特に高速に実行できるユニフィケーションと=.. を多用しているためと思われる。

現在，この評価とは別に，約２００の文法を用いてBUP システムの評価を行なっているところであり，そこではもう少し詳細なデータが得られるはずである。

５．おわりに

本報告では，BUP システムの移植，および，その経験からのESP によるプログラミングについて述べてきたが，この移植により，BUP はSIMPOSのエディタ，デバッガの利用によるinteractive な対話機能が可能なシステムとなった。

今後は，ネットワークシステムを用いた外部辞書との接続や日本語処理等による機能の拡充，更にESP の特徴を生かしたパージング・アルゴリズムの改良も検討中である。

［参考文献］

[1] 松本・田中：「Prologに埋め込まれたbottom up parser:BUP」 N.L. 34-6,1982

[2] Pereira,F and Warren,D.H.:"Definite Clause for Language Analysis" A.I.13 1980

[3] 松本・清野・田中：「BUP の高速化」N.L. 39-7,1983

[4] 近山：「ESP REFERENCE MANUAL」ICOT 1984/2

高性能ＰＲＯＬＯＧマシン：ＨＰＭ
－オペレーティング・システム SAMPLE－

島津秀雄、　小長谷明彦、　中垣良成、　梅村　馨
（日本電気（株）Ｃ＆Ｃシステム研究所）

1.　はじめに

第５世代コンピュータシステム研究開発プロジェクトの一環として、Ｐｒｏｌｏｇを高速に実行するバックエンド型高速プロセッサモジュールＨＰＭを開発中である[1]。SAMPLE（Shell And Monitor for Prolog Language Environment）はＨＰＭ上の基本ソフトウェアとして新たに設計・開発中のソフトウェアシステムである。本稿ではＳＡＭＰＬＥのオペレーティングシステムの設計方針とシステム構成について述べる。

2.　設計思想

ＳＡＭＰＬＥの目的はＰｒｏｌｏｇをベースとしてそれに適したプログラミング環境を構築することである。そしてＳＡＭＰＬＥのオペレーティングシステムの目的はそのようなプログラミング環境を実現するのに適当なソフトウェアの基本構成要素を提供することにある。

ＳＡＭＰＬＥのオペレーティングシステムの設計方針は以下の通りである。

- ＳＡＭＰＬＥ上のソフトウェアのモジュール化を容易に実現するためマルチプロセス機構を実現する。
- ＳＡＭＰＬＥ上で複数のプロセスによるプロセス協調型システムの構築を柔軟かつ容易に実現するため種々のプロセス間同期・通信機構を提供する。
- ホスト側で管理している種々の入出力装置を入出力ストリームとして一元的なアクセス方法で操作できるようにする。
- Ｐｒｏｌｏｇをベースとした言語ＳＵＰＬＯＧ[2]で記述することでＳＡＭＰＬＥ全体を見通し良くかつ拡張しやすくする。しかしながらオペレーティングシステムは効率が重要であるので、オペレーティングシステムで使うデータ群はあくまでｐｕｒｅ　Ｐｒｏｌｏｇの世界で記述するというのではなく、ヒープ・セグメントを使うことで積極的にｓｉｄｅ－ｅｆｆｅｃｔを利用する。ヒープ・セグメントはすべてのプロセスが共有している。

3.　ＳＡＭＰＬＥの構成

図１にシステムの階層構造を示す。ＳＡＭＰＬＥのオペレーティングシステムは、ハードウェア／ファームウェアによる核の上にカーネル層として基本データ構造とプロセス管理がある。カーネル層では、多重プロセスを実現し、種々の同期・非同期プロセス間通信手段を提供している。これによって利用者はＨＰＭ上でプロセス・ネットワークの世界を柔軟に構築することが可能になる。カーネル層の上位にはジョブ制御、入出力管理、メモリ管理、エラー処理などをつかさどるミドル層がある。一般のオペレーティング・システムに比べて、入出力装置の制御はホスト計算機との通信を除いては不必要であるし、ハードウェア／ファームウェアのサポートにより機械語レベルが向上していることから、オペレーティング・システムの層はかなり軽くなっている。インタプリタ、デバッガ、コンパイラなどのユーティリティ群はＳＡＭＰＬＥミドル層の上位につくられ、それら全体として１つの統一したプログラミング環境を構築していることになる。

4.　オペレーティングシステムの基本構成要素

4.1　カーネル

基本データ構造

ヒープ・セグメントは、アーキテクチャレベルでは単に

メモリセルの集合にすぎない。この層ではヒープ・セグメント上の単なるベクター片を抽象データ化するものである。一般に頻繁に使われるリスト、キュー、スタックなどの構造が実現されている。

プロセス管理

マルチプロセスの実現と種々のプロセス間同期・通信機構を提供している。プロセス間通信はＣＳＰ、ＰＬＩＴＳ、Ｖ－Ｋｅｒｎｅｌのようなメッセージ通信型を基本にしている。更に非同期ソフトウェア・トラップを提供しており、キーボードからの割り込み等は非同期ソフトウェア・トラップに抽象化されている。

4．2　ミドル層

ジョブ制御

プロセス管理の上位層として次のものを用意している。

- プロセス・スタッキング
- フォアグランド／バックグランド・ジョブ制御
- ｆｏｒｋ／ｊｏｉｎ

一般のプログラマはプロセス協調型の構築をするときにこの層をインタフェースとすることになる。

入出力サブシステム

ホスト計算機上の入出力装置－ファイル、キーボード、画面等－を入出力ストリームとして一元的なアクセス手段を提供する。

メモリ管理サブシステム

動的に生成／消去、拡張／縮小されるプロセスの動作に必要なスタック群のメモリ管理およびガーベジコレクションを行う。

エラー処理サブシステム

プログラムが誤動作をして例外事象を発生したときに、エラーモニタが生成され、利用者と会話的に処理を行いながら、必要に応じて機械語単位のトレースが可能なシステム・トレーサが起動される。

4．3　プログラミング・システム

プログラミング環境の使い勝手の良し悪しはこの層にかなり左右される。現在インタプリタ、デバッガ、プリプロセサ／コンパイラ、ローダを開発中である。今後、プログラミング環境としてさらに充実させていくためには、日本語処理ユーティリティやエディタ等プログラミング・システム層の一層の拡充が必要である。

謝辞

本研究の進行にあたり、筆者らとともに実装を行っている当研究所木島徳子氏及び日本電気技術情報システム開発㈱丸山、山岸、小柳各氏、ビーコンシステム独立身氏に深謝いたします。

参考文献

［1］山本昌弘他：高性能ＰＲＯＬＯＧマシン：ＨＰＭ－開発目標と設計思想、情報処理学会第30回全国大会論文集、1985　（本稿）

［2］小長谷明彦他：高性能ＰＲＯＬＯＣマシン：ＨＰＭ－基本言語ＳＵＰＬＯＣ、情報処理学会第30回全国大会論文集、1985　（本稿）

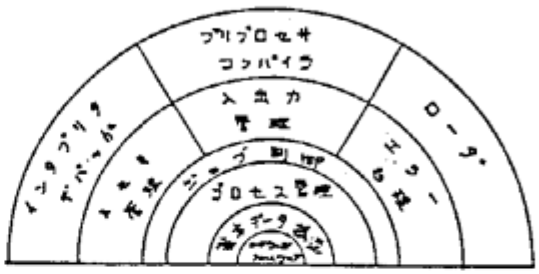


図1　ＳＡＭＰＬＥ

高性能ＰＲＯＬＯＧマシン：ＨＰＭ
――ハードウェア構成――

1C-8

幅田伸一、　中嶋良成、　梅村　護　　（日本電気（株）Ｃ＆Ｃシステム研究所）

1．はじめに　High-Speed PROLOGマシン（以下ＨＰＭと略す）は、第５世代コンピュータ研究開発の一環として開発しているＰＲＯＬＯＧを核言語とした知識情報処理システムを研究開発するためのツールである。ＨＰＭの開発方針は

1）　ＰＲＯＬＯＧで記述したプログラムの高速実行マシンの実現

2）　ＰＳＩで実用的速度で処理不可能な大規模プログラムを実用速度で実行するマシンの実現

である。この方針を満足するためにＨＰＭはバックエンド型プロセッサ構成を採用した。

2．全体構成　高速かつ大規模プログラムの実行可能なマシンを実現するためにＨＰＭが採用したアーキテクチャの特徴を列記する。

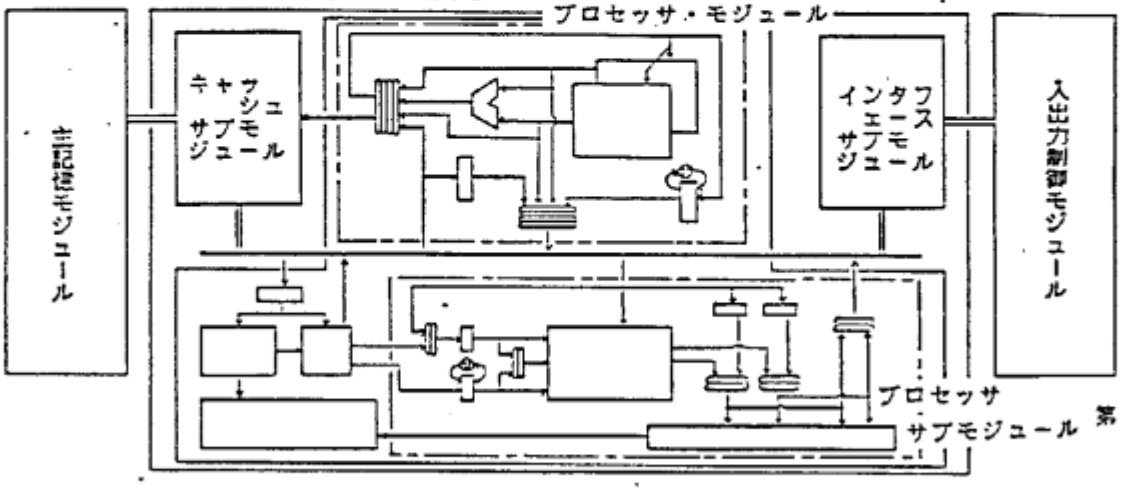
1）　コンパイラによる最適化を最大限に発揮させるレジスタ方式

2）　データタイプをハードウェアで認識するタグ・アーキテクチャ

ハードウェアは３個のモジュールで構成する。ホストプロセッサとのデータ交信制御と入出力装置管理制御を行なう入出力制御モジュール、ＰＲＯＬＯＧの基本処理であるユニファイ処理を行なうプロセッサ・モジュール、大容量主記憶メモリの管理制御を行なう主記憶モジュールで構成する。（第１図）主記憶モジュールは２５６ＭＢの容量を備え、大規模なＰＲＯＬＯＧプログラムを実行するのに必要な作業領域を提供する。

3．プロセッサ・モジュールの構成　プロセッサ・モジュールはＰＲＯＬＯＧの基本処理であるユニファイ処理等を実行するＨＰＭの核となるモジュールである。入出力制御モジュールとのデータ交信制御、ＰＲＯＬＯＧの基本処理であるユニファイ処理の実行、キャッシュメモリの管理制御の３機能を行なう。このためにインタフェース・サブモジュール、プロセッサ・サブモジュール、キャッシュ・サブモジュールの３個のサブモジュールで構成する。この内、プロセッサ・サブモジュールがＰＲＯＬＯＧの基本処理であるユニファイ処理を行なう。

ＰＲＯＬＯＧの基本処理であるユニファイ処理は２つの処理で実行する。１つはＣＡＬＬＥＲとＣＡＬＬＥＥの述語引数のデータタイプを識別し、同一データタイプである時にデータ値の比較を行なう述語引数処理である。他の１つは述語のユニファイ処理過程で生成する実行環境及び制御情報を格納するスタックを制御するポインタの演算を行なうスタック制御ポインタ処理である。プロセッサ・サブモジュールは述語引数処理とスタック制御ポインタ処理



第１図

の２つの処理を並列実行する述語引数処理ユニットとスタック制御ポインタ処理ユニットの２つの専用処理ユニットで構成する。

３．１．述語引数処理ユニット　述語引数処理ユニットは述語引数及び一時変数を保持するデータ・レジスタファイル、メモリ上のデータがロードされる作業用レジスタ（リファレンス・データ・レジスタ）、データタイプを識別して処理フローを制御するタグ解読器で構成する処理ユニットである。レジスタ方式では述語引数処理を高速化するためにデータ・レジスタファイル上に一時的に作成する一時変数を使用してメモリ上のデータを使用する頻度を低くしている。このためデータ・レジスタファイルを使用する処理が増加する。データ・レジスタファイルで行なう処理を高速化するために、述語引数及び一時変数を保持するデータ・レジスタファイルは読み出しポート２個、書込みポート１個を備える。これによりデータ・レジスタファイルは１ステップ中に２セルの読み出しと１セルの書込みが可能である。

ＰＲＯＬＯＧ言語処理方式として採用したレジスタ方式はコンパイラがデータ・レジスタファイル上の述語引数と一時変数を使用する頻度を増加させてメモリ上のデータをアクセスすることによる処理速度の低下を回避することを狙っている。このために述語引数処理ユニットではデータ・レジスタファイル、作業用レジスタとタグ解読器によるデータタイプ解読処理とデータ値比較処理機能を強化している。

３．２．スタック制御ポインタ処理ユニット

ＰＲＯＬＯＧ言語処理系が使用する３本のスタック（グローバル、ローカル、トレイル）とヒープ領域等のメモリ領域上のデータアクセス制御に使用するポインタの演算処理を行なう処理ユニットである。ＨＰＭのＰＲＯＬＯＧ言語処理系はスタックなど８個の領域を使用しているため、メモリアクセスで使用するポインタが約２０個（ＬＣ：LAST CHOICE POINT FRAME POINTER.ＬＥ：LAST ENVIRONMENT FRAME POINTERなど）ある。このためにスタック制御ポインタ処理ユニットは２つのポインタ格納用レジスタファイルとポインタ演算用演算器で構成する。この結果１ステップ中に２個のポインタの演算と１個のポインタの値を退避又はセットする処理が可能である。述語引数処理ユニットが引数の処理中に、スタック制御ポインタ処理ユニットがスタック中の実行環境をアクセスするため実行環境のベースとオフセットを加算してアドレスを求めている。

４．おわりに　ＨＰＭはＰＳＩで処理量またはプログラム規模的に大きなＰＲＯＬＯＧの処理を目的に開発中の大型ＰＲＯＬＯＧ専用マシンである。高速化実現の為に高速デバイスであるＣＭＬ素子を使用し、１００ｎＳｅｃのクロックで動作する。さらに補助記憶の代用となる２５６ＭＢの大容量主記憶を備え、データアクセスの局所性に影響されない高速データアクセスが可能である。処理方式もレジスタ方式を採用しＰＲＯＬＯＧの基本処理であるユニファイ処理を２つの処理に分割して専用の処理ユニットを備え、並列実行を導入している。この並列実行とコンパイラによる最適化、さらに簡単な命令先取り機能によりａｐｐｅｎｄ実行時の処理速度として２８０ＫＬＩＰＳの机上評価結果を得ている。

ＨＰＭが採用した並列実行機能と命令先取り機能の効果を評価し、並列実行機構と命令先取り機構の最適化が今後の課題である。

構　成	マイクロプログラム制御方式 （マイクロコード　８０ビット） アドレス演算用演算器 データ処理用ＡＬＵ
クロック	１００ｎＳｅｃ
データ長	３６ビット （タグ部＋バリュ部）
アドレス長	２９ビット
レジスタ構成	
データ・レジスタ	３２個
ポインタ格納用レジスタ	４８個
その他レジスタ	１６個
キャッシュメモリ	８Ｋワード（４ＷＡＹ） SET ASSOCIATIVE 方式 WRITE THROUGH 方式 論理アドレス・アクセス方式
主記憶	２５６Ｍバイト

第１表　ＨＰＭの諸元

高性能ＰＲＯＬＯＧマシン：ＨＰＭ
1c-7 ー基本アーキテクチャと処理方式ー

中嶋良成、　小長谷明彦、　幡田伸一、　島津秀雄、　梅村　護
（日本電気（株）Ｃ＆Ｃシステム研究所）

1.　はじめに

ＨＰＭはＰＲＯＬＯＧライクな論理型言語を効率よく処理するために設定した機械語を専用のハードウェアとマイクロプログラムによって実行するマシンである。ＨＰＭのアーキテクチャは "D.Warren他のPipelined Prolog Processor"(1)のユニフィケーションに関する方式を基本にして、論理型言語をプログラミング言語として利用し易くする観点と処理の高速化の観点から、データ、機械語および処理方式を拡張している。本論文ではＨＰＭの基本アーキテクチャとして機械語および内部データを報告すると共に、その処理方式の特長と概要を報告する。

2.　内部表現形式

・　タグアーキテクチャ

論理型言語を処理するためには実行時のデータによって決定されるインタプリティブな処理が必須であるので、タグアーキテクチャを採用している。ＨＰＭにおいては語単位でのデータ処理が基本であり、データ、命令全てがタグ付きの１語３６ビットで表現される。

・　５１２Ｍ論理アドレス空間

ＨＰＭの論理空間は２９ビットで表現される５１２Ｍ語である。この論理空間を各種スタック、ヒープ、コード領域など機能別に分けられた８セグメントに分割する。各セグメントは６４Ｋ語を１ページとする１Ｋページまで拡張可能である。

・　最適化可能機械語命令

ＨＰＭの機械語はＰＲＯＬＯＧ言語における処理の基本であるユニフィケーション処理のコンパイラによる最適化を考慮して設定している。この最適化を実現するために一時記憶用のレジスタ群を備え、このレジスタ群を指定可能にした機械語命令を設定した。これらの基本的なユニフィケーションに係わる機械語命令に加え、ｃｕｔ、インデクシングなどの拡張制御機構を導入するための機械語命令を設定した。さらに、加減乗除などの基本演算、入出力処理、プロセス切替え機能を実現するためにマイクロプログラムによって処理される述語を組込み命令として備えている。図１にＰＲＯＬＯＧのクローズがコンパイルされ、命令列に展開された内部表現例を示す。

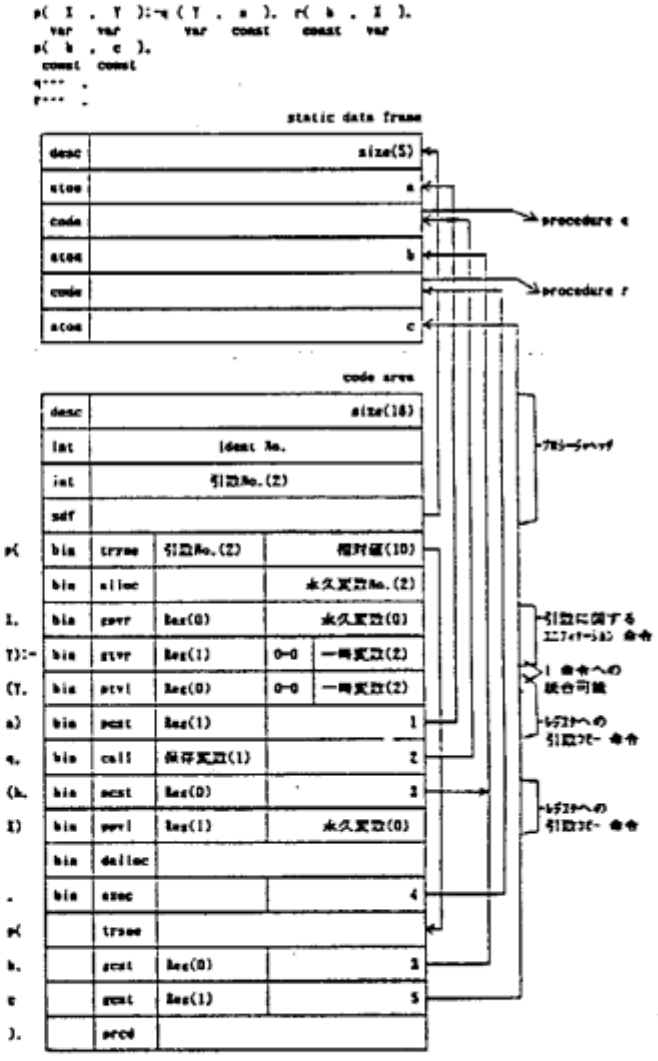


図１　プロシージャの内部表現例

3. 処理方式

- スタック及びヒープ

ＰＲＯＬＯＧのユニフィケーション処理における変数結合、実行順序を制御するために３種のスタックを利用する。

ローカルスタックはクローズ内のゴール間で値を受け渡す間に別の環境の下で処理される可能性がある永久変数とクローズが処理される環境を保存する。

グローバルスタックは可変語数から成り、ユニフィケーション対象である構造体データ，リストデータを格納するためのスタックである。構造体を扱う方式としては構造体コピー方式を採用している。さらに、ローカルスタックを縮小するときに破棄不能な一部の変数を移すためにも利用される。この他ｂｉｎｄ＿ｈｏｏｋ(2) 処理用の環境情報を保持するために利用される。

トレールスタックは一旦結合された変数をバックトラック処理によって未定義状態に戻すｕｎｄｏ処理のために対象変数のアドレスを保持するためのスタックである。

上記の他，主なデータ領域としてヒープを備える。ヒープはプロセス間で共通の領域を提供するための領域として、常にベクトル領域を切出し、利用する。

- レジスタの利用

ＨＰＭの処理方式における特長は機械語命令で指定可能な内部レジスタ群の存在である。ＨＰＭでは、呼び出し元の引数を前述のレジスタに引数をコピーする方式を採用している。この結果、呼び出し元での引数に対する処理はレジスタに引数を出現順にコピーすることが全てである。呼び出し先では各レジスタの引数情報と呼び出し元の引数情報とでユニフィケーションを行う。即ち、呼び出し元では引数をレジスタにコピーする命令列が実行され、次に呼び出し先では、呼び出し先の引数に対応して生成されるユニフィケーション命令列を実行する（図2-(a)）。図2-(b)にレジスタを利用した処理の最適化例を示す。

- 環境の退避

逐次的にＰＲＯＬＯＧプログラムを処理するためには非決定的な処理に備えてクローズの実行環境を退避する必要がある。ＨＰＭでは、この環境の退避を２種のフレームによって効率よく処理する。

エンバイロンメントフレームはクローズのボディ部に複数ゴール存在するときに、前のゴールに関する処理が成功で、制御を次に続くゴール、あるいは最終ゴールの成功では親のクローズへ移すために必要な情報を退避するためのフレームである。

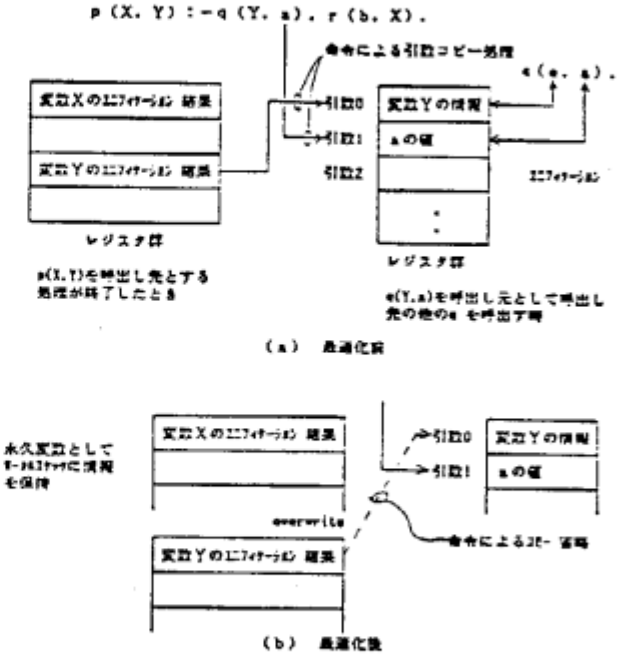


図2 呼出し元と呼出し先での引数処理

チョイスポイントフレームはユニフィケーションが失敗したときに次に処理するために必要な全ての情報を保持する。

このような２種類のフレームを設けることにより、ＯＲクローズがない場合の環境退避を小さな領域から成るエンバイロンメントフレームの作成のみによって十分達成できる。この方式は、ＯＲクローズの少ない決定的なクローズを多用して構成されたプログラムの実行に効果的である。また、表面的には非決定的であっても、コンパイラによるクローズのインデクシングなどによって決定的な形式に最適化される。また、クローズの最終ゴールから呼び出す場合にはそのクローズに対応するエンバイロンメントフレームを消去し、ローカルスタックを縮退することによってテールリカージョンオプティミゼーションを実現する。

4. おわりに

今後、処理性能および実現する機能の観点からＨＰＭのアーキテクチャ、処理方式の評価を進める。 本ＨＰＭのアーキテクチャ、処理方式の開発にあたり、適切な御助言をいただいたＩＣＯＴのメンバーの方々に深謝する。

参考文献

1. E.Tick and D.Warren,"Towards a Pipelined Prolog processor" '84 Int. Symp. on Logic Prog. Feb.'84
2. 高木他、" 拡張制御構造のＰＲＯＬＯＧへの導入" 情処第26回全国大会 P37

高性能ＰＲＯＬＯＧマシン：ＨＰＭ
――基本言語ＳＵＰＬＯＧ――

小長谷明彦、　幡田伸一、　島津秀雄、　中嶋良成、　梅村　恭
（日本電気（株）Ｃ＆Ｃシステム研究所）

1.　はじめに

第５世代計算機への関心が高まると共に、各地でＰＲＯＬＯＧマシンの研究が進められているが、一方で、ＰＲＯＬＯＧの実用性を疑問視する声もある。このような批判が出る背景には、これまでのＰＲＯＬＯＧ言語がきれいさを追求するあまり、本来必要な低レベルなデータ操作機能を排除してきた点にあるのではないかと思われる。ＳＵＰＬＯＧは高性能ＰＲＯＬＯＧマシンＨＰＭ[1]の基本言語として開発されたＰＲＯＬＯＧでＤＥＣ１０ＰＲＯＬＯＧ[2]を包含し、ＰＲＯＬＯＧでシステム・プログラミングを可能とするための種々の拡張が施されている。

2.　ＳＵＰＬＯＧの拡張点

ＳＵＰＬＯＧの主な拡張点は以下の通りである。

（１）　履歴依存性のあるデータ対象、データタイプの拡張を可能とするデータ対象等の導入

（２）　制御構造、例外処理機能の強化

（３）　マルチプロセス機能の導入

（４）　プログラム階層化機能の導入

（５）　マクロ処理機能の導入

2．1　データ対象

ＳＵＰＬＯＧでは整数、アトム、構造体を始めとして16種の基本データ対象を用意している。以下、特徴的なものについてのみ述べる。

制御付変数：
その変数が具象化されるとあらかじめ制約条件として付加されていたハンドラが実行される変数で、制約計算、無限木の計算、文字列ユニフィケーション等に使用される。

エクストラ・アトム：
アトムと同様な属性を持つがアトムと区別可能なデータ対象で、変数を含む項のフリージング（定数化）に使用される。

ロカティブ：
ポインタ型のデータ対象で、各データ対象の内部表現の操作、異なるプロセスのメモリ領域のピーク等に使用される。

ベクトル、シークエンス、コードフレーム、データフレーム：
書き換え可能なデータ対象で、それぞれ種々のシステム情報の格納領域、ビット・ベクトル、プログラム中のコードおよびデータの格納領域として使用される。これらは導出原理との整合性をはかるためにユニフィケーション処理においては要素同士のマッチングではなく要素の格納領域へのポインタ同士でマッチングが行われる。

保護付データ、保護付定数：
利用者定義のデータ対象を実現するために用意されたデータ対象で、例えば、ｂｉｇｎｕｍやｃｏｍｐｌｅｘのようなデータ対象をシステムに追加するときに使用する。保護付データと保護付定数の違いは前者が任意の項を要素として持てるのに対し、後者の要素は定数に制限されている点にある。

2．2　制御構造、例外処理機能

より強力なバックトラック処理、述語呼び出しメカニズムを提供するために、ＳＵＰＬＯＧではＫＬ０[3]で提案された機能を取り込んだ下記の拡張制御構造を備えている。

- レベルカット（任意の手続き呼び出しレベルに対するカット）
- 具象化時実行ハンドラ（ｂｉｎｄｈｏｏｋ）
- バックトラック時実行ハンドラ
- 大域脱出機能（ｃａｔｃｈ　＆　ｔｈｒｏｗ）
- 例外処理機能（組込み述語からの割り出しハンドラ処理および例外事象に対応した割り込みハンドラ処理）

また、条件分岐や繰り返し構造といった手続き的な制御構造の導入も検討しているが、手続き型の制御構造の導入に関してはプログラムの透明性、使い易さ、実行効率等多方面からの研究が今後とも必要と思われる。

2．3　マルチプロセス機能

実用的なソフトウェアシステムを構築するためにマルチプロセス環境は不可欠である。近年、並列ＰＲＯＬＯＧマシンの観点から並列志向のＰＲＯＬＯＧ言語がいくつか提案されている。[4][5][6]　これに対し、ＳＵＰＬＯＧのマルチプロセス機能はＡＮＤ並列やＯＲ並列といったプログラムの並列実行ではなく、入出力処理のように非同期的に動作

するシステムプロセスを記述することを目的としている。このため、各プロセスは独立なスタックを持ち、プロセス間のインタラクションは陽にプロセス間同期通信のための述語を呼び出すことで実現している。AND並列やOR並列に基づくシステム・プログラミングも可能とは思われるが、実用的なオペレーティングシステムを構築するためには、SEND, RECEIVE のような従来型の同期機構を備えたマルチプロセス環境が不可欠ではないかと思われる。

2.4 モジュールおよびパッケージ

大規模なプログラム開発にはプログラムを分割しモジュール化させる方式が不可欠である。SUPLOGではプログラムを階層化させる手段としてモジュールとパッケージの2つを導入した。

モジュールはSUPLOGのコンパイル単位であり、コンパイル時に大域的と宣言しなかった述語は自動的に局所的なものとして扱われてしまう。ただし、アトム名については自動的に大域宣言されたものとして扱う。

これに対し、パッケージはアトム名の階層化を行うものでありパッケージ単位にアトムの属性が与えられる。また外部アトムとして宣言すればパッケージ名：アトム名により別のパッケージから参照することが可能である。さらに、（多重）継承を用いて名前空間を受け継いだパッケージを定義することができる。

2.5 マクロ機能

マクロ機能はユーザ・フレンドリな構文を与えるのに非常に有効な手段である。PROLOGのマクロ機能としてはユニフィケーション機能を使った項レベルのマクロ機能が知られている[7]が、項レベルのマクロ機能ではPROLOGの構文の範囲を越えてのマクロ展開ができないという欠点がある。（例. 変数名の展開、区切り記号を含んだ文字列の展開、等）。SUPLOGでは、より柔軟なマクロ展開を可能とするために、項レベルのマクロ機能ではなく、C－プリプロセッサに見られるような文字列レベルでのマクロ機能を導入した。

3. おわりに

SUPLOGの特徴的な機能を中心にSUPLOGの設計思想について述べた。我々は、PROLOGを発展途上言語と考えており、PROLOGがLISP並になるためにはさらに多くの機能拡張が必要と思われる。しかしながら、真に実用的な処理系はアイデア競争からではなく着実な経験の積み重ねからでしか得られないと我々は確信している。SUPLOGの次のターゲットは関数型言語、対象指向型言語のパラダイムをいかにして取り込むかであるが、我々は応用からの必要性、実行効率、プログラムの透明性等を十分考慮して取り込んでゆきたいと考えている。

謝辞

最後にSUPLOGの設計、開発にあたって、プログラム作成に協力して頂いた櫛田氏、阪野女史、椎崎氏、島田氏、平野氏ならびに本稿をはじめとした多大な資料のドキュメント化に協力して頂いた木島嬢、佐藤氏に感謝の意を表します。

参考文献

1) 山本晶弘他：高性能PROLOGマシン： HPM －開発目標と設計思想－ （本稿）

2) Bowen, D.L. etal "DEC System-10 PROLOG User's Manual, Edinburgh, 1982

3) Chikayama, T. etal "A Draft Proposal of Fifth Generation Kernel Language Version 0.1", ICOT TM-007, 1982

4) Shapiro, E.Y. "System Programming in Concurrent Prolog", Proc. 11th ACM POPL, 1984

5) Clark, K. and S. Gregory "Note on System Programming in PARLOG", FGCS '84

6) Pereira, L. M. "A Distributed Logic Programming Language", FGCS '84

7) Chikayama, T. "ESP Reference Manual", ICOT TR-044, 1984

高性能ＰＲＯＬＯＧマシン：ＨＰＭ
――システムの構成――

1C-5

梅村　護、中崎良成、小長谷明彦、山本昌弘　（日本電気（株）Ｃ＆Ｃシステム研究所）
横田　実、近山　隆、瀧　和男、山本　明、西川　宏　（（財）ＩＣＯＴ）

１．はじめに

高性能ＰＲＯＬＯＧマシン（以下ＨＰＭと略称する）は、第５世代プロジェクトの一環として開発を進めている逐次型推論マシンである[1]。　ＨＰＭの開発目標は、ＰＲＯＬＯＧで書かれた大規模なプログラムを、実用的に充分高速に実行できるようなマシンとすることである[2]。この目標を達成するために、以下の基本方針をたてて設計を行なった。

（１）現状で利用可能な最高速の素子を用いた専用ハードウェアとする。

（２）ＰＲＯＬＯＧの基本メカニズムの高速実行と大規模なプログラム実行のために、大容量の実メモリを備えた構成とする。

（３）コンパイラ技術を利用したきめ細かい最適化が可能な機械語レベルを設定し、ハードウェアによるサポートを行なう。

（４）専用ハードウェアによる高速実行を意図すると共に、システム記述を可能とするＨＰＭ専用の拡張ＰＲＯＬＯＧ言語を設定する。

これらの方針の下にＨＰＭ用ＰＲＯＬＯＧ言語を設定し、バックエンド型の動作を基本とするハードウェアおよび、ユーザ環境を支援するソフトウェアを設計した。システムの構成は図１に示すとおりであり、ファイル管理はホストプロセッサで行ない、本体はプログラム実行に専念する。

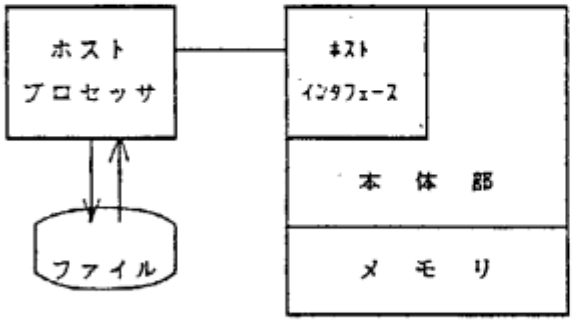


図１．ＨＰＭシステム構成

２．システム構成要素の概要

２．１　言語

ＨＰＭのために設計した拡張ＰＲＯＬＯＧ言語は、現在最も充実したＰＲＯＬＯＧ処理系である、ＤＥＣ－１０版[3]　を基本とし、そのスーパーセットとなっている。主な拡張点は、（１）実行効率、および記述効率／能力を考慮した豊富なデータ対象を備えること、（２）より柔軟、且つ、高能率なプログラム記述を可能とする制御構造を導入すること、の２点である。これらの拡張によって、能率の良いプログラム記述を可能とすると同時に、システム記述までを含めた統一的言語としての位置付けを持つ言語とした。すなわち、この拡張ＰＲＯＬＯＧ言語によって、コンパイラを含むプログラミングシステム機能と、プロセッサ間インタフェースなどのオペレーティングシステム機能とを含むすべてのプログラムを効率良く記述することが出来る。この言語をＳＵＰＬＯＧと称する[4]。

ＨＰＭは、ＳＵＰＬＯＧを効率良く実行するための機械語セットを持つ[5]。　基本方針のとおり、ＨＰＭは高速化達成のため、コンパイラによる最適コード発生を前提としており、機械語レベルはきめ細かい最適化を充分受け入れられるレベルに設定した。例えば、ユニフィケーションにおけるマッチングは原則としてレジスタ間で行なうような命令に展開される。すなわち機械語としてクローズ内の引数特性に応じた適切なレジスタを選択してセットできる命令を備え、コンパイラはこれらの命令を駆使してハードウェアリソースを効率良く利用するコードを発生する。さらにコンパイラによるクローズインデクシングや、テールリカージョンオプティミゼーションによる最適化を考慮したレパートリーを備えている。これらによって、高速処理を疎外するメモリアクセスの頻度も可能な限り抑制される。

２．２　ハードウェア構成

ＨＰＭのハードウェアは基本実行体である本体部、ホストプロセッサとのインタフェース部およびメモリ部から構成される[6]。　本体部とホストインタフェース部は、ＣＭＬ（Current Mode Logic）素子を用いており、クロック速度 100nsで動作する専用ハードウェアである。ホストインタフェース部は、ホストプロセッサとの間のパラレルバスを利用して高速なデータ転送を行なうハードウェアである。本体部およびホストインタフェース部は、８０ビット／ワード、１１キロワードのファームウェアによって制御される。これらは約８０枚のカードに納められる。メモリ部は、２５６キロビット素子で構成される容量２５６メガバイトの実メモリである。

２．３　ソフトウェア構成と動作環境

ＨＰＭのソフトウェアは、ＳＵＰＬＯＧで記述したプログラムをＨＰＭ上で実行するためのプログラミング環境を提供するシステムである[7]。　ソフトウェアシステムの設計方針は以下のとおりとした。

（１）ＳＵＰＬＯＧによってすべてが記述される。

（２）ＨＰＭがバックエンド型マシンであることを利用者がそれほど意識しないで済むような対話的プログラミング環境を実現する。

（３）ソフトウェアシステム自体の構成に際して、モジュール化とマルチプロセス機構を導入し、保守性、および拡張性を高める。

以上の方針の下に設計したソフトウェアシステムは、ホストインタフェースを含む入出力制御およびマルチプロセス機能などをサポートするＯＳ部と、その上層に位置し、ユーザインタフェースとなるプログラミングシステム部とで構成される。

ＨＰＭ上でのソフトウェアシステム動作手順の概念を図２に示す。先ず、ホストシステム上のエディタを用いてＳＵＰＬＯＧプログラムテキストが作られ、ファイル化（ＴＸ）される。ユーザは、ホストシステムのウインドウを介してＨＰＭのインタープリタを呼び出し、テキストファイルをコンパイルするための述語を発行する。ＨＰＭのレジデントコンパイラはＴＸを読み込んでＨＰＭの最適な機械語列への変換を行ない、コンパイルユニット（ＣＵ）を作成する。ＣＵはローダによって内部データベースに登録され、ＨＰＭによる実行が可能となる。以降ユーザはホストシステムのウインドウを介して必要な述語呼び出しを行なうことにより、プログラムが実行される。

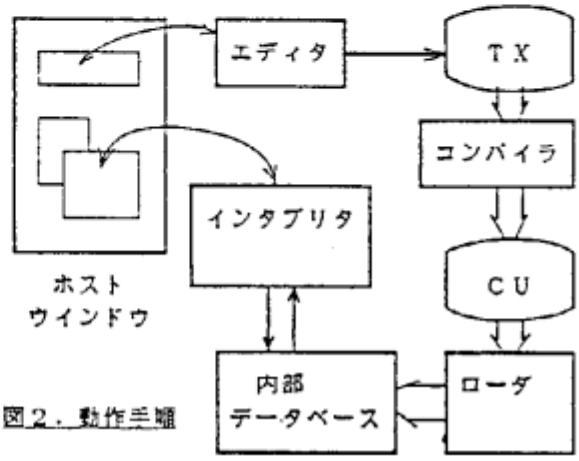


図２．動作三幅

３．おわりに

ＨＰＭはＰＳＩ[1]では規模および速度的に不充分なプログラムの実行を行なうためのシステムとして、第５世代プロジェクト前期内で開発することを目標として設計した。このため実験的要素を多々含んでおり、今後さらに改良／拡張していく必要がある。

参考文献

1) S.Uchida,et.al.:Sequential Inference Machine : SIM Progress Report, Proc.of The International Computer Systems '84, pp.58-69, 1984
2) 山本昌弘他：　高性能PROLOGマシン：ＨＰＭ－開発目標と設計思想－　（本稿）
3) D.L.Bowen: DEC system-10 Prolog User's Manual : Dept.of Artificial Intelligence University of Edinburgh, Dec. 1981
4) 小長谷明彦他：高性能PROLOGマシン：ＨＰＭ－基本言語ＳＵＰＬＯＧ－　（本稿）
5) 中嶋良成他：　高性能PROLOGマシン：ＨＰＭ－基本アーキテクチャと処理方式－　（本稿）
6) 幅田伸一他：　高性能PROLOGマシン：ＨＰＭ－ハードウェア構成－　（本稿）
7) 島津秀雄他：　高性能PROLOGマシン：ＨＰＭ－オペレーティングシステムSAMPLE－（本稿）

高性能ＰＲＯＬＯＧマシン：ＨＰＭ
――開発目標と設計思想――

/C-4

山本喜弘、梅村　護、中嶋良成、小長谷明彦　（日本電気（株）C&Cシステム研究所）
横田　実、近山　隆、内田俊一　（（財）ＩＣＯＴ）

1．　はじめに

知識処理は経験、ルール、規準などに基づく人間の思考作業を支援する情報処理である。即、文字、記号、リスト、イメージなどの非数値データを対象とした非手続き的、非定型処理が中核を占め、我々人間の知的作業の大部分はこの分野に属する(1)。このような新しい情報処理を実現するためには、それに合った新しい言語や計算機の研究開発が必要である。

述語論理に基づくＰＲＯＬＯＧ言語は知識処理用高級言語として、１９８０年代に入りその処理系の開発が活発に進められている(2)。その結果、汎用コンピュータからパソコンにまでＰＲＯＬＯＧを使用できる環境が整備されつつある。又、これらのＰＲＯＬＯＧ言語を用いて非常に広範囲の応用プログラムが作成されつつあり、ＰＲＯＬＯＧの有効性が高まって来ている(2)。

しかしながら、汎用コンピュータを用いたＴＳＳ環境やパソコン等でのＰＲＯＬＯＧ処理系はＰＲＯＬＯＧ言語が内蔵する高度で複雑な推論処理のために高い実行性能を達成することは困難である。ＰＲＯＬＯＧをＬｉｓｐとならび実用言語に育成するためには、高性能且つ使い易いＰＲＯＬＯＧ実行環境を提供することが必須である。このような観点から第５世代プロジェクトの一環として開発を進めている高性能ＰＲＯＬＯＧマシン：ＨＰＭの開発目標及び設計思想について述べる。

2．　開発の狙い及び目標

知識処理におけるプログラミング環境として、人間の知的活動をプログラミング言語で表現し、その表現の検証を容易に且つ高速に行うツールとこれによって作成した応用プログラムを高速に実行するツールとが必要になる。このようなツールを目的とした、ＰＲＯＬＯＧ言語をフレームとして提供するＰＲＯＬＯＧ専用マシンとして、以下の２つのタイプが要求される。

1）　比較的小規模な応用を対象とし、高度なマンマシン・インタフェースを備え、徹底的に使い易さを追求したパーソナル型ＰＲＯＬＯＧマシン、
2）　上記パーソナル型ではカバーしきれない実行速度と記憶容量を要求する大規模な応用プログラムを高速に実行する高性能ＰＲＯＬＯＧマシン。

ＰＳＩ(3)は前者を狙いとしたものであり、本稿の高性能ＰＲＯＬＯＧマシン：ＨＰＭは後者を目的としている(4)。

このために、以下のような開発目標を設定した。

- 実行性能はＰＳＩの５－７倍以上、即２００ＫＬＩＰＳ以上
- 記憶容量はＰＳＩの４倍以上、即実記憶空間は２５６メガバイト以上
- ＰＳＩで作成したプログラムはＰＳＩでは実行性能と記憶容量で不満足な場合に即ＨＰＭで実行できるようにする。このために、ＰＳＩのバックエンド型高速プロセッサ構成にして実行環境の移行をスムーズに行えるようにする。

3．　設計思想

第５世代プロジェクトの前期末に開発を完了し且つ実験ツールとして安定に動作するマシンの開発を前提にして、上記開発目標を最大限に達成するよう設計を進めた。

先ず新しい方式の技術確立を目指し、ＰＳＩと異な

る方式を採用した。

1） 中間言語表現形式ではなくスタック指向機械語命令形式のアーキテクチャ、
2） 構造体、リストの表現形式として、ストラクチャコピー方式。

又、高い実行性能を得るために、ソフト、ハード、デバイスを含めた以下のような総合的アプローチを追求した(5)、(6)、(7)。

1） バックエンド型ＰＲＯＬＯＧマシンとすることによりＰＲＯＬＯＧ処理のみに専用化した構造にする、
2） コンパイラ重視のスタックアーキテクチャの命令仕様とし、コンパイラによる高度な最適化を行う、
3） 高速デバイスＣＭＬ（Ｃｕｒｒｅｎｔ Ｍｏｄｅ Ｌｏｇｉｃ）と高性能実装方式を用いて１００ｎｓｅｃのマシンサイクルを実現する、
4） 内部構造として一部の局所的並列処理方式を採用するとともに、ＰＲＯＬＯＧ実行用命令のマイクロプログラム化（８０ｂｉｔ×１１ＫＷ）を徹底的に進める、
5） ３種類の専用スタックを設け、そのための制御ポインタを高速レジスタファイルに保存する。

更に、高性能且つ大容量記憶空間を提供するために、以下のデバイス及び方式を採用した(8)。

1） ２５６Ｋｂｉｔ ＣＭＯＳチップ、
2） ８ＫＷキャッシュ、
3） ８面の論理的領域に分割した高速管理。

更に、以下の動作環境を提供する(9)。

1） ＰＲＯＬＯＧ言語として、ＤＥＣ１０／ＰＲＯＬＯＧのスーパーセットをサポートする、
2） ＰＳＩ用に作成されたプログラムはセルフコンパイラで再コンパイルを行うことにより実行する、
3） ＰＳＩとのインタフェース処理やＨＰＭの実行制御を行う制御プログラムを備える、
4） 入出力処理はＰＳＩ側で行う、
5） ＰＳＩとＨＰＭ間に高速データ転送インタフェースを設ける、
6） ＨＰＭ実行ジョブはＰＳＩ上ではＰＳＩの１つのプロセスとして実行される。

4. 設計及び評価

前章の設計思想に基づき、５８年４月から９月の予備方式検討を行い、１０月より本格的に設計を開始した。そして、機能設計、詳細設計及び製造を完了し、６０年３月末にはハードウェアおよび中核ファームウェアの検査およびソフトウェアのシミュレーション検査を終了する予定である。又、６０年前半にはソフトウェア検査を完了し、一次リリースの予定である。

現在、ファームウェアの詳細コーディングを完了しているが、これに基づく机上評価では、Ａｐｐｅｎｄが２８０ＫＬＩＰＳで実行できる見通しを得ている。

5. 今後の課題

ＰＲＯＬＯＧが実用言語に育つには使い易い開発環境と知識処理の有効性を示す実験環境の提供が重要である。本ＨＰＭはこの目的を目指したものであり、今後、言語面では文字列処理機能、リレーショナルデータベースやＦｏｒｔｒａｎとのインタフェース機能等のサポート(10)、ハード面ではファームウェア化による高速化やＬＳＩ化によるコンパクト化等を進める予定である。

参考文献

（１） 山本昌弘：推論マシンと将来コンピュータでのその役割、情報処理学会アーキテクチャワークショップインジャパン '84,pp.175-176,1984
（２） 山本昌弘他：言語とその応用 Ｐｒｏｌｏｇ、コンピュートロール No.6,pp.76-94,1984
（３） 西川宏他：パーソナル逐次型推論マシンＰＥＸ－そのハードウェア、情報処理学会第27回全国大会論文集、PP.381-382,1984
（４） S.Uchida,et.al.:Sequential Inference Machine : SIM Progress Report,Proc. of The International Computer Systems '84 ,pp.58-69,1984
（５） 梅村護他：高性能ＰＲＯＬＯＧマシン：ＨＰＭ－システム構成、情報処理学会第30回全国大会論文集、1985 （本稿）
（６） 中嶋良成他：高性能ＰＲＯＬＯＧマシン：ＨＰＭ－基本アーキテクチャと処理方式、情報処理学会第30回全国大会論文集、1985 （本稿）
（７） 小長谷明彦他：高性能ＰＲＯＬＯＧマシン：ＨＰＭ－基本言語ＳＵＰＬＯＧ、情報処理学会第30回全国大会論文集、1985 （本稿）
（８） 幅田伸一他：高性能ＰＲＯＬＯＧマシン：ＨＰＭ－ハードウェア構成、情報処理学会第30回全国大会論文集、1985 （本稿）
（９） 島津秀雄他：高性能ＰＲＯＬＯＧマシン：ＨＰＭ－オペレーティングシステムＳＡＭＰＬＥ、情報処理学会第30回全国大会論文集、1985 （本稿）
（10） 梅村護他：文字列処理を導入したＰｒｏｌｏｇ；ＳｈａｐｅＵｐについて、Proc. of The Logic Programming Conf. '83,1983

ＳＩＭＰＯＳのプログラミング・システム
－　概　要　－

黒川　利明　　近山　隆　　高木　茂行　　坂井　公　　内田　俊一　　横井　俊夫
（（財）ＩＣＯＴ）

1．はじめに

逐次型推論マシン(PSI)のプログラミング・システム／オペレーティング・システムであるSIMPOSのプログラミング・システム部分の概要について述べる。

SIMPOSのプログラミング・システムは、現在コーディネータ，トランスデューサ、デバッガ／インタプリタ、ライブラリから構成されており、SIMPOSオペレーティング・システムの上層に位置している。

本稿ではこのSIMPOSプログラミング・システムの概要を、その設計、実装、現状、将来の拡張について述べる。

2．プログラミング・システムの設計

SIMPOSのプログラミング・システムの設計において考慮したのは次のような点である。

① エンド・ユーザではなく、システム開発者であるユーザを対象とする。
② オペレーティング・システムからプログラミング・システムまでを単一のプログラミング言語で統一する。
③ ユーザとの対話においてはビットマップ・ディスプレイのウィンドウ操作を中心にする。
④ 研究開発の進展に応じて改良，拡張が容易にできるようにする。
⑤ 短期間、少人数でシステムを開発する。

上の①と④とは，「ユーザ自身のシステム開発を可能にする」という意味で同じことである。また⑤の目標を達成するにも、「改良、拡張容易なシステム」が必要なので、これも①と④と同様のこととなる。そして、①、④、⑤の目標を達成するための解として、②の「単一言語システム」があげられる。

したがって、上のような点を考慮した設計方針は次のようなものとなった。

(1) ICOTで開発したシステム・プログラミング言語 ESP[1] をオペレーティング・システム、プログラミング・システムのシステム記述言語とし、さらにユーザが各種システムを開発する場合の言語とする。
(2) この言語ESP にはウィンドウ・システムなどを容易に開発し、ユーザが改良・拡張可能なように、クラスの概念を用いたオブジェクトの機構を取り入れる。
(3) ウィンドウ・システムの設計には、クラスの概念を取り入れたSmalltalk やFlavors [2] を参考にする。

論理型プログラミング言語にクラスの機能を取り込んだということは、単にウィンドウ・システムだけではなく、SIMPOS全体のモジュラリティを高め、「短期間・少人数でシステムを開発する」という⑤の要請を満たすものであった。

プログラミング・システムの最初の版での機能構成要素としては、プログラム作成用のエディタ、プログラムを即時実行するデバッガ／インタプリタを提供することにした。これらを実現するためには、入出力処理を支援するトランスデューサ、コンパイラ機能を含めてプログラムのオブジェクトを管理するライブラリ、プログラミング・システムの管理とウィンドウを通じての対話機能を支援するコーディネータの五要素に分けて、機能設計を行なった。

これは実装の問題にもなるが、コーディネータは常駐プロセスとしてプログラミング・システムで扱う個々の処理単位（エキスパートと呼ぶ）の管理を行なうことにした。エディタやデバッガ／インタプリタはこのエキスパートとして登録され、コーディネータを介していくつも作られたり、消されたりするものとする。

トランスデューサやライブラリはそれ自体をプロセスとするのではなく、処理中に呼び出されるサブルーチンの形式で実現した。

3．プログラミング・システムの実装

SIMPOSプログラミング・システムの実装においては次のような点に注意した。

① クラス／オブジェクトの構成。
② ウィンドウなどを用いたユーザ・インタフェース。
③ 反応時間を含めた応答性能。

クラス構成と各クラスにおける述語を決定することによって、いわゆるモジュール間インタフェース部分が決定された。その後の述語の具体的な処理は各担当者で分担して進められるので、プログラミング・システム開発チームの

作業分担は円滑に行なわれた。

ウィンドウなどを用いたユーザ・インタフェースは最終的には実際にPSI 上でウィンドウを表示しないと実感がでないので、この短期間の開発においては難かしい部分であった。このインタフェースについては、今後も変更／改良の可能性がある。

応答性能に関しては、プログラミング・システム側での対応よりは、オペレーティング・システム側でのプロセス切替えの高速化とか、入出力を含めたウィンドウ・システムの高速化とかの対応が主となった。

4．プログラミング・システムの現状

第五世代プロジェクトのような先進的研究プロジェクトにおいては、成果としてのシステムを次にはツールとして使いこなし、より大きな成果をうるという一種のブート・ストラッピングが不可欠である。

それは同時に、今日のツールが明日には陳腐化し、新しいツールが必要となるということを意味している。

プログラミング・システムの現状は、このツールとしてやっと使えるようになってきたというところであるが、これまでのわずかの経験からもこれが第五世代プロジェクトの推進に不可欠なツールとなる兆候が示されている。

それらは次のようなものである。

① DECSystem 20などよりはるかに大きいメモリ空間。
② ウィンドウ・システムを用いたユーザ・インタフェース。
③ パーソナル・マシンとしての実行効率。
④ 日本語処理機能。
⑤ グラフィックス機能。

①と③はPSI というハードウェア自体の成果にも関連している。②、④、⑤は広い意味ではユーザに対するインタフェース、通信の種類が一躍広がったことを示している。現在の限られた機能でも、PSI 上で対話的にプログラムを書き、デバッグする方が全体としてのプログラム開発時間が著しく少なくなる。

これからシステム・ソフトウェア、ユーティリティなどが充実すればSIMPOS／PSI はさらに使いやすくなると期待される。

5．将来の拡張

SIMPOS／PSI 上で実際に開発する用途ごとに様々な拡張が将来さらに考えられると思うが、現時点でわかっているものとしては次のようなものがある。

① ヘルプ機能の高度化／充実。
② 図形・文書処理機能の高度化／充実。
③ 静的・動的なプログラム管理。

①のヘルプ機能は高度なワークステーションとしてPSI を使いこなすために必要な機能である。PSI の端末の前にいれば、他の文書を参照したり、他人にたずねたりする必要なしに何でも実行できるようにするということが狙いである。

②の文書処理は図形などの処理も含む。入出力での生データの処理だけではなく、知識処理を含めて考えてゆきたい。端的には日本語処理やグラフィックス処理を高度化／充実させることである。新しいフォントの登録、加工も含むし、ヘルプ機能と関連して文書の内容自体の加工も考えたい。

③のプログラム管理は、現在でもライブラリにより部分的に行なわれているのだが、これを高度化／充実するとともに、プログラムのデータベース化とか、プログラムの解析、さらにはプログラミングのプロジェクト管理のようなことを含めて考えたい。

研究分野としてのプログラミング・システムの高度化は、同じく第五世代プロジェクトで進められている知的プログラミングの研究と相互に関与するものであるから、「知的な」プログラミング技術をシステムとしてどう支援してゆくかが研究課題となる。

6．おわりに

SIMPOSプログラミング・システムの各構成要素については今回個別に報告されているのでそれを参照していただきたい。

全体については昨年11月に開かれたFGCS'84 の予稿集[3]を見ていただくのが適当であろう。

参考文献

1) Chikayama, T. "Unique feature of ESP", Proc. FGCS'84, (1984)
2) Cannon, H.I., "Flavors—A non hierarchical approach to object-oriented programming", MIT AI Memo, (1982)
3) Yokoi, T., Uchida, S., "Sequential Inference Machine : SIM —Its Programming and Operating System", Proc. FGCS'84, (1984)

SIMPOSのプログラミング・システム
―カナ漢字変換―

藤牧 茂 (学習研究社(株))　栗山 健 (学習研究社(株))　上田尚純 (三菱電機(株))　辻 順一郎 ((財)ICOT)

1．はじめに

Prolog専用マシンである逐次型推論マシンφ（PSI）のプログラミング／オペレーティング・システムであるSIMPOSにおける漢字入力機能について報告する。

SIMPOSは知識情報処理プログラムの開発支援システムであり、日本語を用いた優れたマン・マシン・インタフェースを実現する事、及び日本語を含む自然言語処理を効率よく行う事はφおよびSIMPOSの主要な設計目標の一つであった。このためφは16ビット・コードを内部コードとして採用しており、出力装置としてもビット・マップ・ディスプレイ（BMD）、漢字プリンタを用意している。更にはレーザ・プリンタも間もなく追加される予定である。漢字入力については、ワープロで主流となってきているカナ漢字変換方式による入力方法を採用している。カナ漢字変換機能をシステム中核部に持たせており、柔軟性に富む方式となっている。

以下で、SIMPOSのカナ漢字変換をはたす機能モジュールの概要と特徴について述べる。

2．実現方式

SIMPOSにおけるカナ漢字変換モジュールの位置を図1に示す。キーボードからの入力データは、英数字入力モード（通常モード）時は、1文字の入力毎にその単位で、キーボード・ハンドラからウィンドウ・システムに渡されるが、漢字入力モード時はキーボード・ハンドラからのデータは一旦カナ漢字変換モジュールに渡されそこにバッファリングされる。ユーザが（ファンクション・キーを使用して）漢字への変換を指示した時点でバッファリングされていた文字列は漢字に変換されてウィンドウ・システムに渡される。この変換過程はユーザ・プロセス側からは一切意識されず、ユーザ・プロセスには仮想の漢字入力機器から漢字が直接入力されたかのように見える。

φのキーボードはカナ・キーが無いためカナ入力はローマ字で行う。カナ漢字変換モジュールではローマ字をまずカナに変換した後、カナ漢字変換を行っている。

漢字を含む文書やプログラムのファイル作成はエディタ(EDIPS)を使用して行う。EDIPSは汎用のスクリーン・エディタであり、マウスを用いた強力な編集機能を提供している。

辞書は処理高速化と実現の容易さを考慮して主記憶常駐とした。

3．ユーザ・インタフェース

φのキーボードには10数種のファンクション・キーが用意されており、このキーを使用して漢字入力モードのオ

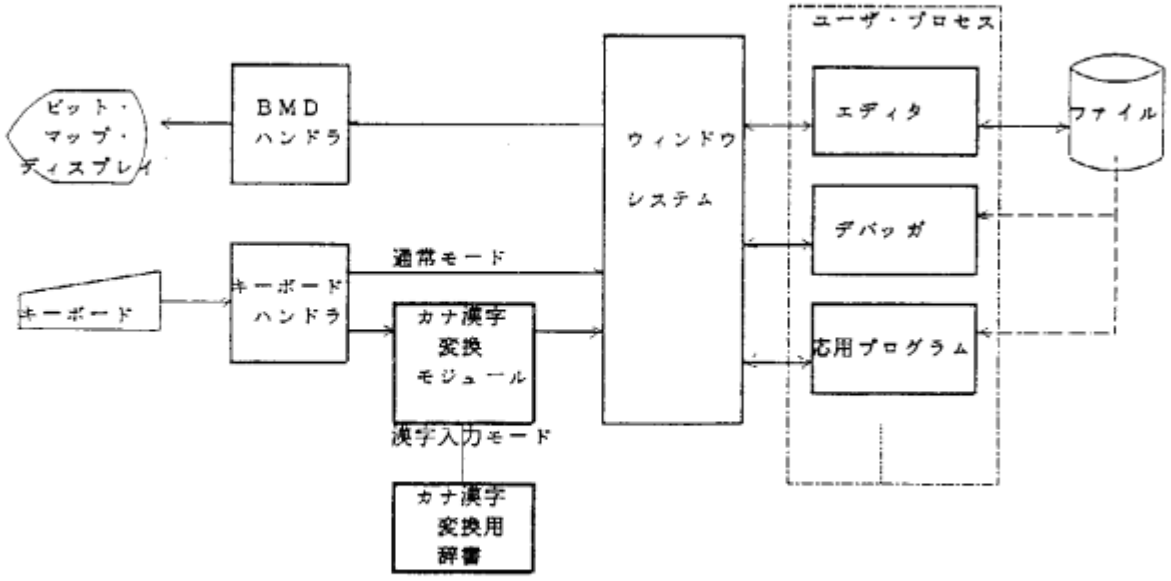


図1　SIMPOSのカナ漢字変換実現方式

ン／オフ、変換種別の指定等の各種の制御は１タッチで行えるようにしている。

カナ漢字変換に際して、入力文字のエコー・バック、同音異義語の表示、ユーザによるその選択等の処理が必要であるが、そのためのウィンドウとして、ＢＭＤ画面の下端にシステム専用として確保している３行分の領域の先頭２行を使用している。（図２）　１行目にキー入力データのエコー・バックおよび同音異義語の表示を行う。
２行目はガイダンス情報表示用であり、ファンクション・キーの用途等の情報を表示する。ユーザが漢字選択を指示すると、選択された漢字を入力対象のウィンドウに送ると共に１行目をクリアして次のキー入力を待つ。

４．機能

今回組込んだカナ漢字変換モジュールには、

１．　文節変換　（１文節）
２．　単語変換
３．　漢字変換
４．　ＪＩＳ区点コード入力による漢字変換
５．　ＪＩＳ漢字コード入力による漢字変換
６．　平仮名変換
７．　片仮名変換

の、各変換機能を持つ。無変換は機能のｏｎ／ｏｆｆで対処している。学習処理、次候補、先頭戻り、ガイド表示、等で使い勝手についても考慮している。

文節変換は単文節による変換で、接頭語処理、接尾語処理、数詞検定、文法検定等も行っている。

使用した辞書は約３．５万単語（此の中には地名約１５００件、姓名約２０００件含まれている）あり、約４２０Ｋbyte、第１水準、第２水準込みの漢字辞書は、約７０Ｋbyteである。

これらは主記憶に読み込んでいる。今回のカナ漢字変換モジュールは文節変換には最長一致法を使い、辞書の配列もそれに合わせてある。学習処理はラスト・イン・ファースト・アウトである。文節変換で単語の検定時において、次候補による自動的な絞め処理をやると、無駄なヒットが多くなりかえって使いにくい結果となる場合が多い。このため最初に文法検定等でＯＫとなった単語が最適であり、次候補はその同音の中で見つけるということにし、次候補による自動的な絞め処理はやっていない。

又、文法検定についても極力最低限度におさえた。

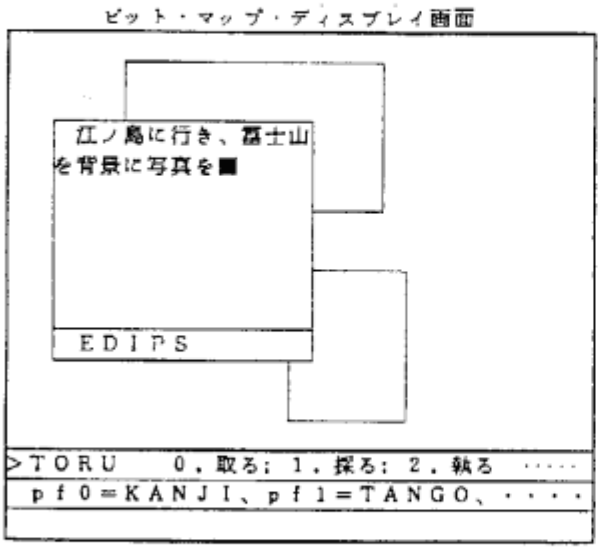


図２　カナ漢字変換用ウィンドウの位置

５．ＥＳＰでの記述による利点

カナ漢字変換モジュールはSIMPOSの他のシステムと同じく全てＥＳＰで開発した。従来の手続き型言語でカナ漢字変換モジュールを記述するのに較べて作成が全体的に容易であった。そのうち次のような点で、特に効果的であった。

例えば、使用した漢字、又は単語を記憶し、取出すという学習機能において、リスト操作と組込述語を使うと、アッセンブラでは１０００step位かかってしまうとおもわれることがわずか１０step位で出来てしまうことである。

又、文法検定時、特に接頭語、接尾語、前置助数詞、後置助数詞のサーチ時バック・トラックの利用等で、テーブルの該当データが一時に取りだす事ができる。つまり、該当データを、ユニフィケーションを利用して見つけそれをリストに追加し、そしてそこで"fail"することにより、バックトラックさせ、次のデータを見つけに行く、これを繰り返すことにより全てのデータが、リストとして作成されるということになる。この処理はテーブルを別にして約１５step位で出来てしまう。

６．おわりに

今後の課題として、ユーザー・インタフェースの改善、カナ漢字変換機能の一層のインテリジェント化、文書編集機能の強化等がある。

プロダクションシステムを用いた日本語文生成

高橋　雅則　．　鈴木　浩之　．　清野　正樹

松下電器産業（株）　システム研究開発センター

1　はじめに

日本語の深層格構造（以下D構造と呼ぶ）から表層格構造（以下S構造と呼ぶ）への変形をプロダクションルールを用いて行うシステムを試作し、よい実験結果が得られたので、システム構成及び変形規則並びに実際の変形例について報告する。

なお、本システムは、変形生成文法に基づいて、意味表現から日本語文を生成するシステムの一部分である。

2　システムの構成と概要

本システムはPrologで記述されており、図1のような構成になっている。

2.1　変形部

変形部は生成用の文法である変形規則をプロダクションルールとみなして、これをトランスレータを用いてPrologのプログラムに変換したものからなる。各ルールは一つのPrologの述語に変換される。　従って、実際の変形は、Prologの処理系により直接実行される。変形部はパターンマッチングにより、エンジンから送られたSub_treeに適用可能な規則を一つ探して変形を行い、エンジンに返す。

2.2　エンジン

エンジンは入力されるD構造を分割し変形部に送り、変形後のデータからS構造を合成する働きをする。

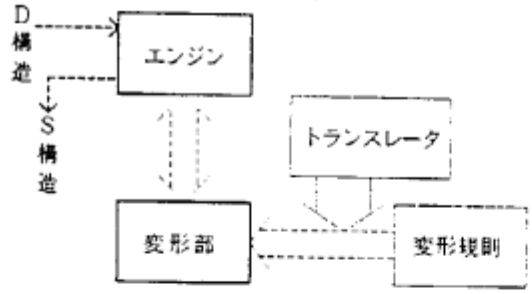


図1　システム構成図

```
joshi:[CX1,X11,
        [pred,X21,[doushi@[move],D],X22],
        X12,
        [case,X23,[*place,X31],X24],
        X13] -->
      [CX1,X11,
        [pred,X21,[doushi,D],X22],
        X12,
        [case,X23,[place,X31,[joshi,ni]],X24],
        X13].
```

図2　変形規則の例（助詞付加規則）

D構造はTree構造を持っている。エンジンは深さ優先の探索によりSub_treeを捜し出し、その都度変形部に送る。次に、変形部で変形を受けたSub_treeを受け取り、元のSub_treeと置き換えて探索を続ける。

変形部では送られてきたSub_treeに対して一度にただ一つの規則しか適用しないが、一般にSub_treeは複数の規則の適用を受ける。このためエンジンでは、Sub_treeが規則の適用を受けずに帰って来るまで繰り返し変形部に送って変形を行う。さらにすべての変形を終えたSub_treeは、再帰的にエンジンに送られ再び深さ優先の探索を受け、変形が試みられる。この操作もまたSub_treeが全く変形を受けずに帰って来るまで繰り返される。

3　変形規則

変形規則は変形生成文法に従って記述されており、現在のところ約１００の規則が用意されている。[1][2]

これらの規則は、削除・付加・移動の三種類の基本的な変形操作の何れかまたはこれらの組み合わせから構成されている。それぞれの基本操作を用いた代表的な規則としては、削除変形には補文構造に対する同一名詞句削除規則が、付加変形には助詞付加規則が、移動変形には語順規則が挙げられる。また、受動態・願望・使役を表す文のように補文をとる文には、補文構造を展開するために削除・付加・移動の各変形操作を組み合わせた規則も用意されている。

このなかでも特に語順規則は、日本語の基本的語順がＳＯＶであるといった日本語文法の土台をなす規則、各述語毎に決定されている格構造[3]に従って名詞句を並べ替える規則、その他自然な日本語を生成するための様々な規則が用意され、変形規則の中では最も数も多くまた重要な規

```
rule(A,B,C,A,B,D,joshi,E,[F|[joshi([],[],ni)]]) :-
        divide(C,pred(G,H),I,J,K),
        divide(J,doushi(L,M),N,O,P),
        intersect([move],M),
        divide(K,case(Q,R),S,T,U),
        divide(T,place(V,W),X,Y,Z),
        not_member(joshi,V),
        !,
        divide(D,pred(G,H),I,A1,B1),
        divide(A1,doushi(L,M),N,O,P),
        divide(B1,case(Q,R),S,C1,D),
        divide(C1,place([joshi|V],W),X,G1,Z),
        divide(D1,joshi([],[]),Y,F,[]).
```

図3　変換後の変形規則

則となっている。 これを充実させることで、より自然な日本語の生成を行うことができる。

図２に変形規則の例を示す。 この規則は、「述語の動詞が動的動詞の場合に場所格を表す名詞句に助詞"に"を加える」という変形を表している。 図中、大文字で始まる記号は変数を表し、ここにはどんな構造も入ることができる。 また、規則はリスト形式で記述されており、入力データと同じ形式であることから、規則の記述が容易であるという特徴を持つ。 図２の変形規則をトランスレータを用いてPrologのプログラムに変換したものが図３である。

tree中の各ノードは図４に示すような付加情報を持つ。即ち、第一引数としてそのノードをルート（根）とするSub_treeにこれまでに適用された規則名。第二引数としてそのSub_treeが持つ性質（品詞名等）及びテンス・アスペクト・ムードの情報を何れもリスト形式で持つ。

これらの付加情報は、変形規則に以下のような特徴を与えている。

(1) 第一引数は規則の繰り返し適用を防ぐために用いている。 一般には変形されるSub_treeのトップノードをその規則の要のノードだと思い、その第一引数中にその規則名が入っていない事を確認し（図３のnot_member）、変形後のそれにその規則名を付け加える。 但し、トップノードがその変形規則の要のノードでない時もある（再帰代名詞化規則など）ので、要のノードを"＊"を用いて指定することができる。

(2) 規則の適用条件の細かい所までTreeのノード名を変えて対処すると煩雑になるので、ノード名に性質を付加し、それをチェックできるようにした。 それらは、適用条件のTree中に、"@"や"&"を用いて記述できる。 これらは規則を適用する際に指定されたノードの第二引数を調べる述語に変換される。（図３のintersect）

4 実行例

図５に実際に変形を行った例を示す。 左側のD構造に変形操作を加えて右側のS構造を生成している。 S構造は既に表層文の語順を反映した形になっているので、ここから語彙項目を取り出し、形態素処理を施して表層文を生成する。

5 おわりに

以上、変形生成文法の文法規則である変形規則をプロダクションルールとみなし、それらをprologのプログラムにプレコンパイルする方式による文生成について述べた。この方式を採用したことにより、次の二点が達成できた。

(1) 文法記述の容易性

(2) 処理の高速化

今後、より柔軟な日本語を生成するために、単語の分類、言語現象の研究を進めるとともに、上記の変形規則の記述方式をベースに、よりよい記述方式を求めていく考えである。 また今回作成した枠組みが適当なものであるかどうかの評価を含めて、日本語以外の言語の生成を行うことも検討中である。

本研究は(財)新世代コンピュータ技術開発機構(ICOT)よりの委託(発伝339)-02)の一環として行ったものである。

参考文献

[1] 井上「変形文法と日本語」、大修館書店（1976）
[2] 柴谷「日本語の分析」、大修館書店（1978）
[3] 水谷他「文法と意味Ⅰ」、朝倉書店（1983）

```
ノード名([これまでに適用された規則名],
        [構造の持つ性質，テンス・アスペクト情報])
```

図４ ノードの付加情報

```
bun([kako,hitei])
 |-pred
 |   |-jodoushi([shieki])--seru
 |-case
 |   |-agt([focus])
 |   |   |-np
 |   |       |-meishi([human,male])--tarou
 |   |-dat
 |   |   |-np
 |   |       |-meishi([human,female])--keiko
 |   |-bun
 |       |-pred
 |       |   |-doushi--yomu
 |       |-case
 |           |-agt
 |           |   |-np
 |           |       |-meishi([human,female])--keiko
 |           |-obj
 |               |-np
 |                   |-meishi([not_human])--hon
```

```
bun([kako,hitei,ga_ni,shieki,seq1],[kako,hitei])
 |-case([seq2],[])
 |   |-agt([focus,seq2,joshi],[focus])
 |   |   |-np([],[])
 |   |   |   |-meishi([],[human,male])--tarou
 |   |   |-joshi([],[])--ga
 |   |   |-joshi([],[])--wa
 |   |-dat([joshi,seq2],[])
 |   |   |-np([],[])
 |   |   |   |-meishi([],[human,female])--keiko
 |   |   |-joshi([],[])--ni
 |   |-agt([seq2,joshi],[])
 |   |   |-np([],[])
 |   |   |   |-meishi([mcut],[human,female])--*
 |   |   |-joshi([],[])--ni
 |   |-obj([joshi,seq2],[])
 |   |   |-np([],[])
 |   |   |   |-meishi([],[not_human])--hon
 |   |   |-joshi([],[])--wo
 |-pred([],[])
 |   |-doushi([],[])--yomu
 |   |-jodoushi([],[shieki])--seru
 |   |-jodoushi([],[])--nai
 |   |-jodoushi([],[])--ta
```

図５ 実際の変形例