記号処理 13－3
（1980. 10. 17）

リスト処理向きデータフローマシンの検討

雨宮真人　長谷川隆三　三上博英
（日本電信電話公社　武蔵野電気通信研究所）

1．まえがき

現在、データフローマシンアーキテクチャは次の点で期待が持たれている。

(1) 問題に内在する並列処理性を素直に反映させた並列実行が可能であり、高性能を得ることができる。

(2) 分散制御が可能であり、現在進展が著しいVLSI技術を効果的に利用することができる。

(3) 副作用を排除した関数型言語を効率的に実行することができ、ソフトウェア生産性の高いプログラミングシステムを実現することができる。

(4) 並列実行により、非決定性処理を効果的に行い得る可能性を持っており、今後の情報処理システムにとって不可欠な知能処理アーキテクチャのベースを与えることができる。

しかし、これらの期待を現実的なものとするには、多くの解決すべき問題が残されている。特に(3),(4)の観点からは汎用性，非数値処理問題への適用性が重要であり、そのためには構造体データの処理方式を明確にしていかねばならない。

非数値処理の典型はリスト処理であり、それはLispに代表される。本稿での議論はLispを念頭に置いているが、それは言語構造，データ構造が単純明解であること、そしてその単純さの中に構造体データ処理の基本的問題が集約されているためである。

実用的Lispではprog-featureなど純粋性を損う副作用を持込んでいるが、これはノイマン型マシン上での効率的処理を得るためには必要なことであろう。即ち逐次的実行制御下での実行速度の向上、中央集中制御によるメモリ管理下での効率的なメモリ使用を得るためには、ポインタの書替えは必然であろう。

しかし、データフロー制御により高度の並列実行が達成されるなら、またVLSI技術の適用によりLogic in memory等の高機能化メモリ素子の実現が可能となるならば、副作用を排除した純関数的性質に基づくリスト処理方式も十分実用的な意味を持つことになると思われる。

本稿ではこのような観点から、先ずリスト処理における基本的演算について考察し、メモリ中に演算機能を埋込んだ構造体メモリの構成を示す。続いて、この構造体メモリを用いたリスト処理向きデータフローマシンアーキテクチャを示し、最後にデータフロー制御によるLispインタプリタの構成例を通し、Lispの並列処理性を論ずる。

2．リスト処理と構造体メモリ

本章では、データフロー制御による、構造体、特にリストデータの並列処理性の問題について考察する。トークンとして、構造体データ自体を流すのは現実的ではないので、構造体へのポインタを流すことにする。(1)

リストの表現は種々考えられるが、リスト処理は基本的には二進木の操作としてとらえてよい。そこでここでは、副作用を排除したpure Lispにおける基本演算を考慮の対象とする。以下の節ではpure Lispの基本演算の特徴を明確化し、これを反映した構造体メモリの構成法を探る。

2.1 リスト処理における基本演算とメモリ機能

pure Lispの基本関数はcar, cdr, cons, atom, eqの5つである。このうち、cons以外はすべてメモリ参照オペレーションであり、consのみが新データセルをつくり出し、そこにcar, cdrポインタを書込む。書込みは一度限りで書替えはない。しかも、新データセルがつくられる場所は任意である。従って、これら5つの基本関数から構成されるプログラムは、副作用がなく関数的性質を保持する。

リスト処理は基本関数に帰着するので、本質的にメモリ参照を主体とするメモリオペレーションとして捉えることができる。従ってメモリオペレーションをいかに効率良く行うかが、リスト処理の鍵を握ることになる。

リスト処理の並列性を追求する上での問題点は、主としてメモリ競合, 副作用の2点である。pure Lispのように関数性を徹底することで、メモリオペレーションどうしの独立性は得られる。残る重要な問題はメモリ競合である。データフロー制御により、実行制御とメモリオペレーションがパイプライン化される。パイプの容量が十分あれば、メモリアクセス・オーバヘッドが実行制御に影響しなくなり、メモリセルへの不断のアクセスが可能となる。従って概念的には、メモリセル対応にオペレーションユニットを設け、演算の並列性を上げることにより、メモリ競合は回避できる。この考えは、物理的には図1に示すLogic in memoryにつながる。これをどの程度までとり入れるかは、並列処理性とコストとのトレードオフである。現実的な構造体メモリ構成法については2.3節で論じることにする。

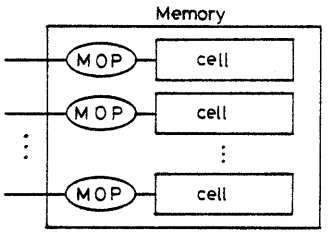


図1　ロジックインメモリ

2.2 ガーベジコレクション

構造体データをデータフロー制御で処理する場合、従来の計算機方式に比べると、より頻繁にデータのコピーが生じる。大量のデータを取扱う構造体メモリに於いて、メモリの有効利用は極めて重要であり、並列処理での効率的ガーベジコレクション法の探究が必要である。

従来、ガーベジコレクションにはマーク・スキャン法が用いられてきたが、ここでは次の理由により参照カウント法を用いる。

1. データフロー制御でリスト処理を行うので、リストへのポインタがトークンとして演算ユニット, インストラクションメモリ, ネットワークに散在する。このため実行を止めずにアクティブなリストをつかまえることは、非常に困難である。

2. 副作用のない演算を基本としており、リストデータの値の変更を行う場合はconsにより新たにデータセルを生成するので、循環リストをつくる危険性はない。

本方式では各データセル毎に参照フィールドが設けられ、参照カウントはメモリオペレーションの実行の度に更新される。　次に、基本関数の参照カウント設定アルゴリズムをAlgol風言語で示す。　ここでは、

セルxの参照カウントをr(x), 演算の結果をz, zを待っている演算の数をdで表わす。

```
procedure Car(x,d);
 begin
      z ← car(x);
      Red(x);
      r(z) ← r(z)+d
 end

procedure Cdr(x,d);
 begin
      same as Car(x,d)
 end

procedure Atom(x);
 begin
      z ← atom(x);
      Red(x)
 end

procedure Eq(x,y);
 begin
      z ← eq(x,y);
      Red(x);
      Red(y)
 end

procedure Cons(x,y,d);
 begin
      z ← cons(x,y);
      r(z) ← d
 end

procedure Red(x);
 begin
      r(x) ← r(x)-1;
      if r(x)=0 ∧not(atom(x))
         then begin
                 Red(car(x));
                 Red(cdr(x))
              end
 end
```

参照カウントによるガーベジコレクションの完全性で問題となるのは、条件式中の演算が実行されない場合の処置である。 例えば、

if P then f(x) else g(y)

において、Pが真のとき g(y)は実行されない。この場合セルyの参照カウント r(y)は1減じられないまま残る。そこで図2のように、手続きRedのみを実行する消滅(Erase)アクターを特別に用意する。

2.3 構造体メモリ

本節では、構造体データを処理し格納する構造体メモリの構成法について述べる。

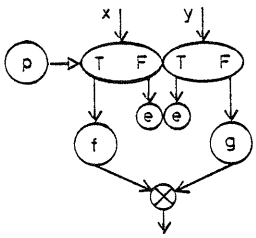


図2 条件式と消滅アクター

構造体メモリを共有メモリとして実現した場合、メモリ競合、アクセスネックの問題は回避されない。構造体データの参照に関数対応の局所性があれば、関数対応のローカルメモリとして構造体メモリを構成することにより、関数レベルの同時性は向上するであろう。しかし一般には、関数対応の局所性は少いと思われる。この場合コピーをして局所性を持たせるか、又は参照を分散化させるかがトレードオフ点となる。ここでは徹底して参照を分散化することで同時性を追求する。さらに低レベルの同時性を追求するには、データセルの各フィールドを、それぞれ独立にアクセスできるようにすれば良い。

ガーベジコレクションのための操作はメモリオペレーションの度に必要であるので、参照カウントおよびガーベジタグの操作がネックとなる。このため、加減算等の機能を持つ Logic in memory に近い機能が要求される。論理的にはセル対応に演算ユニットが存在するものとみなしてもよいが、実際問題としては、複数個のセルからなるブロック対応に演算ユニットがつくと考えるのが妥当である。

1) 構造体メモリ(SM)の構成

構造体メモリの全体構成を図3に示す。SMは独立動作可能な複数個のメモリバンクに分割され、各メモリバン

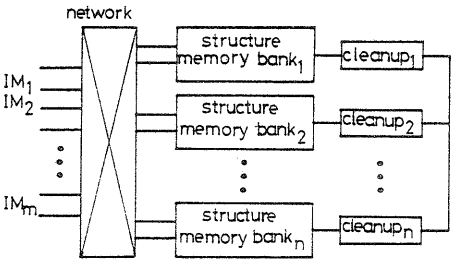


図3　SMの構成

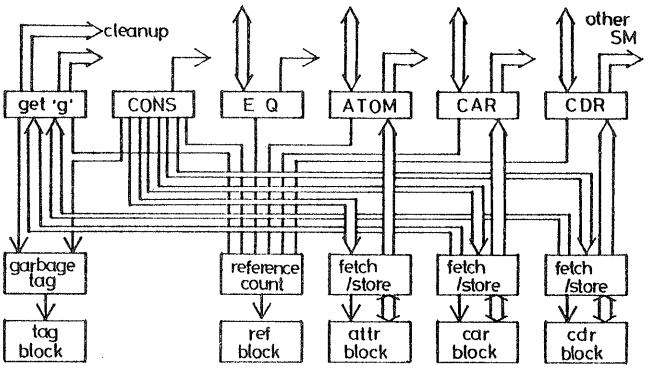


図4　メモリバンク機能構成

ク対応にガーベジコレクション用のcleanupモジュールが用意される。

実行制御部は命令パケットを、そのオペランドアドレスでデコードし、対応するメモリバンクへ送出する。

図4にメモリバンクの機能構成を示す。メモリバンクは独立動作可能なtag, ref, attr, car, cdrのブロックから構成され、各ブロックにはそれぞれ専用の操作ユニットが設けられる。Carオペレーションを例にとって説明しよう。CAR操作ユニットは命令Car(x,d)を受取ると、セルxの参照カウントを1減じるため、count down(x)という指令をreference count controllerへ送ると共に、fetch(x)指令をfetch controllerに送出する。fetch controllerはxの場所からデータz(=car(x))を読出し、CAR操作ユニットへ返す。CAR操作ユニットでは結果を実行制御部へ送ると共に、セルzの参照カウントをd増加するため、count up(z,d)指令をreference count controllerへ出す。この時、結果zが自メモリバンクの範囲外であれば、当該メモリバンクに対しcount up(z,d)要求を送る。

2) ガーベジタグ操作

cleanupモジュールの処理概要を以下に示す。

reference count controllerがcount down(x)指令を実行した結果、セルxの参照カウントが0になると、そのセルのtagブロックにはガーベジコレクションの必要ありという表示gがセットされる。

get 'g' controllerは任意の時点でg表示のあるセルを読出し、そのcar, cdr部を得る。get 'g' controllerは、これら2つのポインタが指すセルのcount down指令をつくり、reference count controllerへ送る。これで当該セルは用済みとなるので、g表示はリセットされ、代わりに空きセル表示fがセットされる。

この後、次のg(if any)に対し同様の処理を繰返す。

このようにして、スタックを用いずにg, fが次々に伝搬されガーベジコレクションが行われる。しかも、タグ操作が遅れても、生きているセルが回収されるという論理的矛盾は生じない。

3. リスト処理向きデータフローマシンの構成

2章で述べた構造体メモリを用いるデータフローマシンの全体構成, インストラクションメモリの構成, ネットワーク構成を中心に述べる。

3.1 全体構成

全体構成を図5に示す。本マシンはインストラクションメモリに連想メモリを用いたアーキテクチャ[2]を基本としているが、特に次の点が特徴となっている。

(1) インストラクションメモリ(IM)は、これをプログラムメモリ(PM)とデータメモリ(DM)とに分離し、メモリの有効利用を図っている。

PMは読出し専用のメモリであり、プログラム(関数本体)が格納される。DMは到着オペランドを保持するバッファとして用いられる。

(2) 各IMには関数プログラムが割付けられており、関数間のCall/Returnパラメタがイム間通信ネットワークを介して転送される。IM間通信ネットワークは3.3で述べるような構成により、論理的に動的な木構造を実現する。

(3) 従来の演算ユニット(functional unit : f)は構造体メモリ(SM)の中に埋込まれている。SMは多数個のバンクに分割され、各バンクに演算ユニットが埋込まれている。

(4) IM-SM間のパケット転送には調整ネットワーク(A-net),分配ネットワーク(D-net)が用いられる。

A-netは命令パケット中のオペランドアドレスからSMバンクを選び、そこへパケットを転送する。D-netは結果パケットを送るべきIM(パケット中にその演算を要求したIMの番号が保持されている。)を選び、そこへパケットを転送する。A-net, D-netにはrouting network[3]が用いられる。

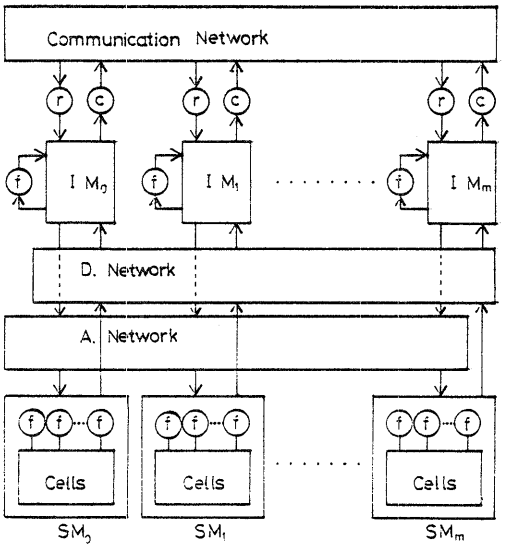


図5 マシンの全体構成

3.2 IMの構成

IMの構成を図6に、PMおよびDMのフィールド構成を図7に示す。

PMは関数番号(func#), アクター名(an), 第1・第2オペランド名(opr n1・opr n2), オペランド数(n), 行先指定数(des), 命令コード(opcode)の各フィールドからなる。一方、DMは起動番号(in), アクター名, オペランド値(val), 第1/第2オペランド指定(r), ガーベジタグ(t)の各フィールドからなっている。プログラムの共有を行う

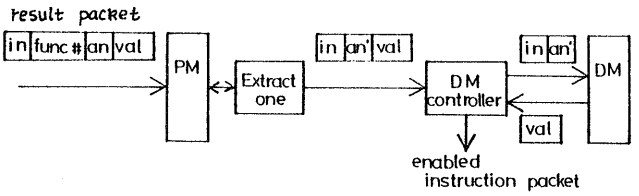


図6 IMの構成

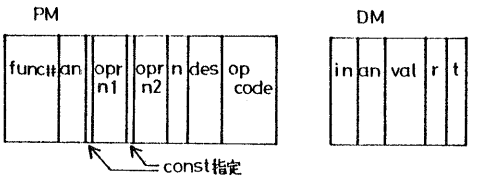


図7 PM, DMのフィールド構成

ので、データには起動番号が付される。

結果パケットがPMに送られると、パケット中の<func#, an>をキィーとして関数番号および第1・第2オペランド名フィールドが検索される。一致した命令語のオペランド数が1であれば、直ちに結果の値を基に1オペランド型の命令パケットをつくりA-netへ送出する。　オペランド数が2の場合は、一致した命令語のアクター名とパケット中の<in>をキィーとしてDMが検索される。一致すれば当該データを読出し、先のパケット中の<val>と合わせて2オペランド型の命令パケットをつくり、A-netへ送出する。
不一致の場合は、ガーベジタグもを検索して空きセルをみつけ、そこにパケット中の<val>と第1/第2オペランドの区別を示すrを書込む。

3.3　IM間通信ネットワーク

各IMにはいくつかの関数ボディが割付けられる。IM間の通信はプログラム実行時に生起する関数間のCall, Returnでのパラメタ授受として行われる。従ってIM間の論理的結合は木構造である。更に実行中に於けるIMの負荷を分散させるには、関数ボディの割付け並びにIM間の論理的結合構造を動的に変化させることが必要である。この要求を実現する動的可変構造ネットワークを、スイッチモジュール(これをIntelligent Switch：ISと呼ぶ)を用いて図8のように実現する。図は4×4のIS16個，IM48個の場合の構成例である。

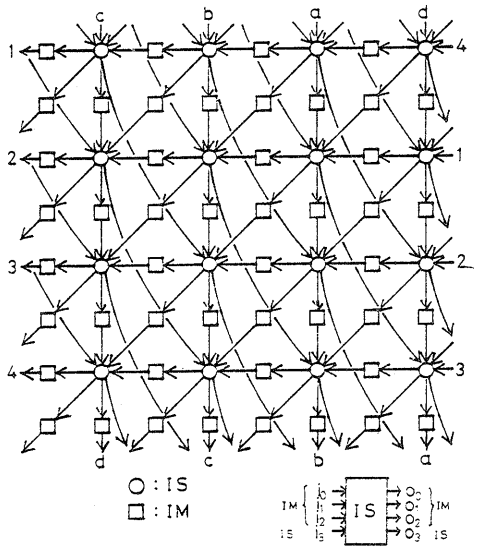


図8　IM間通信ネットワーク

ISの入力ポートi_0, i_1, i_2にはIMがつながり、i_3にはISがつながっている。出力ポートO_0, O_1, O_2からはIMがつながり、O_3からはISがつながっている。　ISは2つのモードで動作する。パケットに転送先の指定がないとき(即ち関数callが起こったとき)は、出力ポートO_0, O_1, O_2につながる軽負荷のIMを探す。転送先IMが選択されたとき、転送元－転送先間のIM相対アドレスが定まり、以後の転送ではこの相対アドレスが転送先指定情報として用いられる。

O_0, O_1, O_2がすべて過負荷のときはO_3につながるISにより、更に他のIMを探索する。パケットに転送先が指定されているときは、ISは指定情報に従ってパスを設定する。　相対アドレス方式を用いるのは、call, return両時点でのデータ転送に同一指定情報が利用できるからである。

4.　Lisp インタプリタ

本章ではLisp 1.5インタプリタ[4]の核であるApply, Eval, Evcon, Evlis, およびPairlis, Assocのデータフローによる実現を示す。

我々のリスト処理プロセッサが対象とするソース言語には、単一代入と再帰を基本とする高級言語を想定してい

る。(これを我々はValid: Value identification language と呼んでいる。)

Validによるソースプログラムの実行には2つの方法が考えられる。第1の方法ではコンパイラによってマシンコード(データフローグラフ)に変換し、直接実行する。第2の方法ではプリコンパイラにより5式に変換し、Lispインタプリタで解釈実行する。これは既存のLispシステムにおけるコンパイラとインタプリタの関係と同じである。

ここでは特にLispインタプリタの実現例を示すことによって、データフロー制御で効率的なLispインタプリタが実現可能なこと,並びにLispでの高度な並列処理が可能であることを示す。

4.1 データフローによる実現

Lispインタプリタの各関数は再帰関数として定義されているが、ノイマン型マシンの場合と同様に、実現する立場からは効率的な実行制御が可能なように再構成する必要がある。このような観点から再構成した各関数のデータフロー表現を図9に示す。
(またValidによるLisp 1.5インタプリタ記述を付録に示す。)

図中□で示した部分はマクロ名であり、▭で示した部分はサブルーチン(関数)名である。関数はその入力アークに最初のトークンが到着したとき活性化され、部分的な実行が開始される。

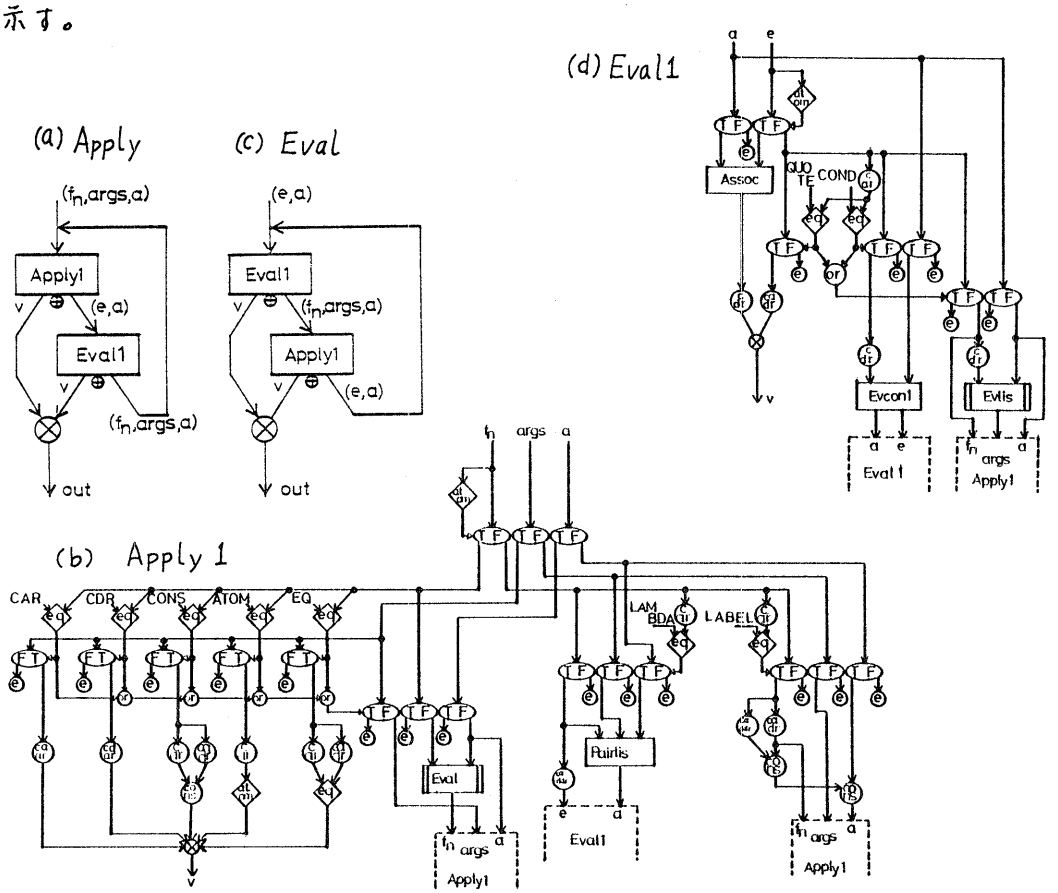
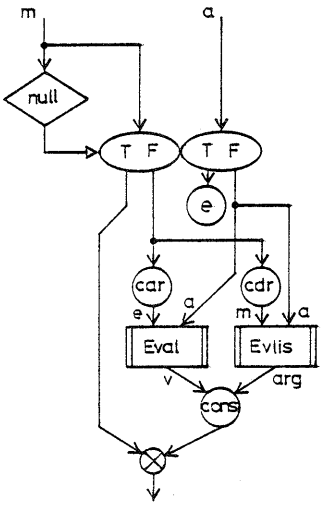
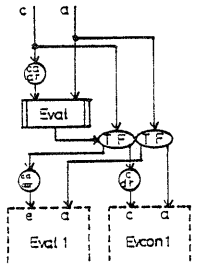


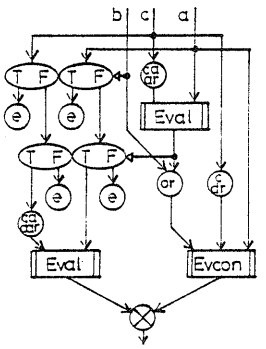
図9 データフロプログラムによるLispインタプリタ(1)



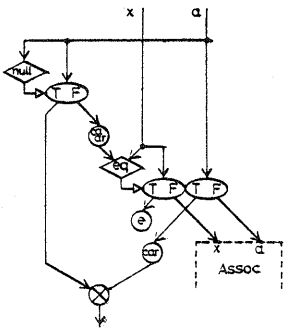
(e) Evlis



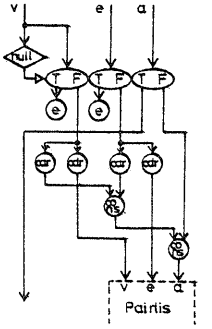
(f) Evcon1



(g) Evcon の parallel version



(h) Assoc



(i) Pairlis

図9　データフロープログラムによるLispインタプリタ(2)

ここに示したインタプリタ実現は、①実行上並列性のないtail recursiveな構造を持つ部分はループ制御により、実行の高速化を図る。②non-tail recursiveな部分はそこに並列実行性を求めるという方策によっている。

例えば図に示すように、Apply, Evalは Apply1, Eval1 を交互に用い、ループ制御になっている。一方、Evlisはnon-tail recursiveな構造を持ち、この部分からはEvalが並列的に活性化される。

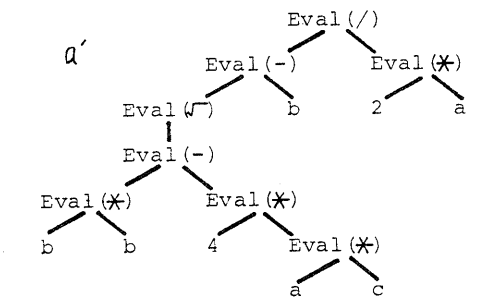
4.2　並列処理性

non-tail recursiveな構造はEvlisの再帰実行，Evcon実行中でのEvalへの再帰において現われる。Lispプログラムの並列実行はこの部分の制御を並列化することにより得られる。

Evlisは関数の引数を1つずつ順に評価していく。このEvlisを図9(e)のような構成により、Evalを並列的に活性化し、それらの結果をまとめてconsすれば各引数の評価を並列化することができる。

(a)

```
/[-[√[-[*[b;b];*[4;*[a;c]]]];b];*[2;a]]
```



(b)

```
fact[m;n] = [equal[m;n] → n;
             t → times[fact[m;quotient
                              [plus[m;n];2];
                       fact[addl[quotient
                              [plus[m;n];2]];
                            n]]
```

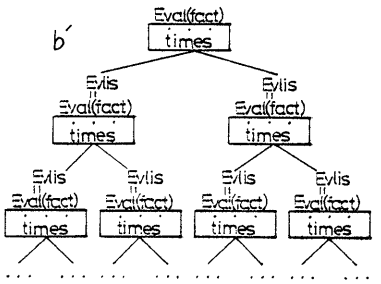


図10　Evalの並列化の例

引数の並列評価の効果は大である。（この場合、メモリ競合を避けるにはボディの評価を行う前に、そのS式データをコピーする必要がある。）
例を図10に示す。(a) は trivial な例である。(b) は divide and conquer 方策による階乗計算の例である。

Evcon の実行も並列化の可能性を持つ。図9(f)の実現では、cond式のP部を逐次評価するLispのセマンティクスに従って、ループ制御を用いている。しかし、Evconを再帰により図9(g)のように並列化させることもできる。（Evconは入力アークにトークンが到着していなくても活性化されることに注意。）この場合、P部のEvalが並列的に活性化され、結果が真であるものについてそのe部のEvalが活性化される。（このような実行制御で、P部評価の並列化効果を引出すには、従来のプログラム技法に若干の変更が必要となる。例えばequal関数は

```
equal[x;y] = [and[atom[x];atom[y]] → eq[x;y];
              not[or[atom[x];atom[y]]]
                  → and[equal[car[x];car[y]];
                        equal[cdr[x];cdr[y]]];
              t → f]
```

と定義することにより高い並列性が得られる。

Evcon の並列化は非決定性制御[5]への可能性を秘めているが、その具体的な制御法は今後に残された問題である。

5. あとがき

本稿ではデータフローマシンでの構造体データ処理の問題を議論した。並列処理性の観点からは構造体メモリへのアクセス競合が問題となるが、関数性を遵守した処理方式では基本的演算がデータセルの参照と新データセルの生成であり、且つこれらの演算が個々独立であるという点に着目し、演算機能を備えた構造体メモリを多数個用いるデータフローマシンを提案した。そしてデータフローによるLispインタプリタの例により、リスト処理の効果的実現および並列処理性について論じた。

本稿の方式によるリスト処理では、並列性の観点からデータの局所性がないことが好ましいが、一般には局所性については未だ不明である。局所性の問題はconsの際の新データセルの取り方に依存する。またデータのコピーを行って局所性を持たせることの良否も大きな問題である。

本方式の場合、データの局所性を前提にすれば各SMバンクをIMに固定することができ（A-net, D-netが不要となり）、単純な構成となる。その他関数割付けの問題などマシン構成上のいくつかのトレードオフ点があるが、これらは並列実行下におけるデータの統計的性質に依存する問題であり、今後シミュレーションにより明らかにしていきたい。

最後に日頃我々の研究に対し叱咤激励して下さる山下紘一第一研究室長，並びにアーキテクチャ研究グループの諸氏に感謝する。

参考文献

(1) Ackerman, W.B.: A Structure Memory for Data Flow Computers, MIT/LCS/TR-186, 1977.
(2) 長谷川，三上，雨宮：連想記憶を用いたデータフローマシンの一構成法，信学会計算機アーキテクチャ研究会資料 EC79-55,1980
(3) Tripathi,E.and G. Lipovsky: Packet Switching in Banyan Networks, Proc. 6th Annual Symposium on Computer Architecture, pp.160-167, 1979.
(4) McCarthy, J. et.al.: Lisp 1.5 Programmer's Manual, The MIT Press, p.106, 1966.
(5) Dijkstra, E.W.: Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, Comm.ACM,18,8, pp.453-457, 1975.

付録　Validによる Lisp インタプリタの記述

Valid は以下の特徴を持つデータフローマシン用高級言語である。

(1) 変数の概念は持たず、プログラムは Value 定義と function, macro 定義とからのみなる。特に value 定義では $[x_1, x_2, \cdots, x_n] \Leftarrow$ Expression により複数 value を同時に定義できる。

(2) 式の集合である Block の概念を持ち、Block の中では scope rule に従う local value の定義が許される。また Block 中の式の評価順序は陽に示されない。

(3) iteration の記述はすべて再帰概念に基づく。従来の loop 表現は、

for <iteration value name>.
:<initial value>

do <block including recur>

の構造を持つ無名の再帰関数として表現される。

(4) その他データの構造化機能, fork - join による並列実行式の記述機能等を持つ。

ここでは、データは list type のみを扱うプログラムとして記述されている。

```
function Apply(fn,args,alist)
= for (x,y,z):(fn,args,alist)
   do begin
      (t,xx,yy) <= Apply1(x,y,z);
      xxx <= if t then xx
             else begin
                  (tl,xl,yl,zl) <=Eval1(xx,yy);
                   x2 <=  if tl then xl
                                else recur(xl,yl,zl);
                  return(x2)
                  end;
      return(xxx)
      end;


macro Apply1(fn,args,alist)
=  for (x,y,z):(fn,args,alist)
    do case
         atom(x) => case
                      x='car' => return('t',caar(y),nil);
                      x='cdr' => return('t',cdar(y),nil);
                      x='cons' => return('t',cons(car(y),cadr(y)),nil);
                      x='eq' => return('t',eq(car(y),cadr(y)),nil);
                      others => recur(Eval(x,y),y,z)
                    end;
         car(x)='lambda' => begin
                              x' <= caddr(x);
                              y' <= Pairlis(x,y,z);
                              return(nil,x',y')
                            end;
         car(x)='label' => begin
                             x'<= caddr(x);
                             z'<= cons(cons(cadr(x),caddr(y)),z);
                             recur(x',y,z')
                           end
       end;

function Eval(e,a)
= for (x,y):(e,a)
   do begin
       (t,z1,z2,z3) <= Eval1(x,y);
       xx <= if t then z1
                  else begin
                         (t1,y1,y2,y3) <= Apply1(z1,z2,z3);
                         if t1 then y1 else recur(y1,y2,y3)
                         end;
       return(xx)
      end;

macro Eval1(e,a)
= for (x,y):(e,a)
   do
     case
       atom(x) => return('t',cdr(Assoc(x,y)),nil,nil);
       car(x)='quote' => return('t',cadr(x),nil,nil);
       car(x)='cond' => begin
                          (x',y')<=Evcon(cdy(x),y);
                          recur(x',y')
                        end;
       others => begin
                   y1 <= car(x);
                   y2 <= Evlis(cdr(x),y);
                   y3 <= y;
                   return(nil,y1,y2,y3)
                 end
     end;

macro Evcon1(c,a)
= for (x,y):(c,a)
   do if Eval(caar(x),y) then return(cadar(x),y)
                         else recur(cdr(x),y) ;

function Evlis(m,a)
= if null(m) then nil
             else begin
                    x <= Eval(car(m),a);
                    y <= Evlis(cdr(m),a);
                    return(cons(x,y))
                  end;

macro Assoc(x,a)
= for (y1,y2):(x,a)
   do case
        null(y2) => return(nil);
        equal(y1,caar(y2)) => return(car(y2));
        others => recur(y1,cdr(y2))
      end;


macro Pairlis(v,e,a)
= for (x,y,z):(v,e,a)
   do if null(x) then return(z)
                 else begin
                        x'<= cdr(x);
                        y'<= cdr(y);
                        z'<= cons(cons(car(x),car(y)),z);
                        recur(x',y',z')
                      end;
```