計算機アーキテクチャ59−13
オペレーティング・システム29−13
(1985・10・25)

リスト処理指向パイプライン処理システムに関する一検討

Pipelined machine for List processing

坂井 賢一　三科 雄介　中村 維男　重井 芳治
Ken-ichi SAKAI　Yuusuke MISHINA　Tadao NAKAMURA　Yoshiharu SHIGEI
東北大学工学部
Tohoku Univ.

1．はじめに

計算機のハードウエアは、近年の集積回路技術の進歩により、真空管からVLSIへときわめて大きな発展を遂げてきた。しかし、その間ノイマン型計算機におけるアーキテクチャにも、プログラミング言語にも、革新的な変遷は起こらなかった。その結果として、ハードウェアの能力がますます強力になるのに対し、それを十分に利用できるソフトウェアが存在しない、というのが現状である。またソフトウエアに対する要求は高まる一方であり、その費用は上昇し続けている。

このような背景のもとに、計算機の性能向上を目的とした様々な並列処理システムの研究が現在各所で盛んに行われている。並列処理システムの主なものとしては、パイプライン処理方式をとり入れたベクトル計算機、内部にプロセッサアレイをもつアレイ計算機、複数の汎用プロセッサを結合し、それらを並列に動かすマルチプロセッサシステムがあげられる。特にパイプライン処理方式は、制御の容易さや価格性能比の良さなどから、現在論理回路レベル、命令実行レベル等、様々な形で用いられている。しかしそのほとんどはパイプラインの各セグメントの処理機能が固定されており、専用演算器として用いられている。このような動きの中で、パイプラインの各セグメントの処理機能を処理されるデータに付随するプログラムで供給することによりパイプラインの汎用化を目的としたシステムの研究が行われてきた(1)(2)(3)。

一方、ソフトウエア生産性の面から関数型言語によるプログラミング方式が注目を集めるようになってきた。従来のノイマン型プログラミング言語ではメモリからメモリへの写像というハードウェアレベルでプログラムを記述するのに対し、関数型言語ではオブジェクトからオブジェクトへの写像という抽象レベルでプログラムを記述する、という特徴を持つ。さらに、高度な並列処理が可能なことから関数型言語を直接実行できるような計算モデル、処理系の研究が進められている。その代表的なものとしてデータフロー計算機、リダクション計算機等があげられる。従来のノイマン型のデータ処理方式がコントロールフローに従っていたのに対し、これらの計算機の特徴は基本的に異なる。まず、データフロー計算機ではデータそのものの流れがデータ処理の制御を行っていく方式である。また、リダクション計算機のデータ処理は、命令が実行されるのが、その演算結果が、ある他の命令により要求されるときだけであるという要求駆動機構に基づく。我々は、このような二つのアプローチとは別に、関数型言語の実現モデルとして、前述の汎用パイプラインを用いることを提案した(4)(5)。

本稿では、まず関数型言語を用いたリスト処理を行うパイプライン処理の概要について述べる。次にシステム構成及び実行制御について検討し、基本関数の実行の様子を例を用いて示す。最後にプロセッサ内処理アルゴリズムの確立を目的として、動作の定式化を行う。

２．関数型言語とパイプライン

本稿での議論は、J.Backus によって提案されたFPを念頭においている。FPシステムは、次の三つの集合から構成される(6)。

(1) オブジェクトの集合
(2) オブジェクトをオブジェクトに写像する関数の集合
(3) 新しい関数を形成するために、既存の関数やオブジェクトを結合する関数形式の集合

関数型言語でのプログラミングは、(3)の関数形式を用いて単純な関数から複雑な関数を組立てることができるという特徴を持つ。　一般に、関数型言語は並列処理の自然な記述を可能とし、かつ大量の暗黙の並列性が含まれているため、並列処理システムでのソフトウェアとして期待されている。しかし、実行効率上の問題があるため関数型の並列処理能力が打ち消される問題もある。

関数型言語実現のモデルとしてパイプラインアーキテクチャを採用した理由は次のようなことにある。

(1) 関数列のパイプライン性

FPの関数の基本的構成は、合成の関数形式を用いた関数の列として表現される。関数の起動は要求駆動により行われるが、実際のシステムではデータ駆動に基づいて内側の関数から次々に実行される。これらの個々の関数は空間的にマッピングすることにより合成された関数がパイプライン的に実行可能と見做せる面がある。

(2)リスト処理のパイプライン性

構造データを扱うにおいて、アレイではなくリンクドリストを使用することにより、目的とする要素をアクセスするのに線形時間を必要としてしまう。しかし、リストをたどるという非本質的な作業に存在する時間的なずれを、パイプライン制御における時間空間の二次元空間にうまくマッピングして、空間的なずれに置き換えることにより、処理速度の減少を防ぐことができる。

以上の理由により、関数型言語に存在する並列処理構造から様々なレベルでパイプライン性を抽出し、パイプラインで効率よく実行しようとする狙いが理解できる。

３．システム構成

3.1 システムの概要

従来の汎用パイプライン処理システムでは、処理されるデータとプログラムがすべてパイプライン内を流れていたが、リスト構造データすべてをパイプライン内に流すのは現実的ではない。そこで、リスト構造データは、構造体メモリ内に格納し、パイプライン処理部と分離し、パイプライン処理部では、プログラムとリストへのアドレスをセグメント間に渡り流すことにより処理を進める。しかし、このような形体ではメモリープロセッサ間のトラフィック増大(メモリアクセスボトルネック)により、複数のプロセッサが同時的に処理を行うことによる並列性が失われてしまう可能性がある。それゆえ、本システムでは、メモリアクセスが十分高速である、という仮定のもとに、関数型言語に内在するパイプライン的並列性を抽出し、価格性能比の良いアーキテクチャでパイプライン処理することを目的としたものである。

3.2 システム構成

関数型言語をパイプラインアーキテクチャ上においてインプリメントするための一ステップとして、　本稿では特に２章の(2)の点に着目する。そして、対象とする関数をFPの基本関数における構造変換関数に限定したときのシステムの実行制御について考察する。以下に本システムの基本構成について述べる。

本システムの構成を図１に示す。システムは、パイプライン処理部、構造体メモリ(SM)、ネットワーク(I_NET,R_NET)、そしてコントロールユニット(CU)から構成される。

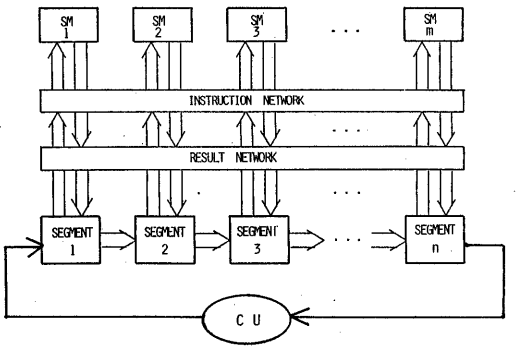


図１．システム構成

コントロールユニット(CU)はパイプライン処理部で実行させる関数の供給を行う。まずFPで記述されたテキスト表現をスキャンし、データ構造をSMに生成し、パイプライン処理部に対し、同時に実行可能な関数形式のパケット(以下これをジョブと定義する)と作用させるデータ構造のルートアドレスを供給する。

SMはLISPにおけるデータ構造のようにセルによるリスト構造の形でデータを保持するメモリで、パイプライン処理部から転送される命令に従って構造データ操作を行う。SMは、アトムデータ用領域と、データ構造用領域に分割されている。各セルは固定長で、CARフィールド、CDRフィールドの他、各フィールドの示すセルがアトムか非アトムかのtagフィールドを持っている。論理的にはSMは一個であるが、複数のプロセッサからのメモリアクセス競合を軽減するため多バンク構成とし、ネットワーク(バス)でパイプライン処理部のプロセッサと結合されている。SMの各バンクは、パイプライン処理部から転送される命令を解釈する機能を持っているとする。

パイプライン処理部では、複数のセグメントが多段に縦続接続されており、各セグメントはCUから供給されるジョブの中の関数(タスク)の起動および構造データ操作命令の起動を行う。各セグメント間では、ジョブと構造データ操作命令のオペランドとして用いられるアドレスが転送される。セグメントの構成を図2に示す。セグメントはインタプリティングユニット(IU)、メモリアクセスユニット(MAU)、リンケージユニット(LU)、入力コントローラ(IC)、出力コントローラ(OC)から構成される。セグメントの基本的な処理の流れを図3に示す。MAUはSMに構造データ操作命令を出力し、結果アドレスを受け取る。LUはパイプラインで処理されるタスク内のパイプライン通過における一セグメント当の処理単位であるプロセスを結合するための情報、すなわち並行に実行されるプロセスの結果をリンクするのに必要なアドレスを保持する。ここで、LU内に保持されるアドレスのことを履歴と呼ぶ。ICは前段から転送されるアドレスと関数を受信し、OCは後段にアドレスと関数を転送する。IUは各セグメントで現在起動されている関数とIC、LU内のアドレスの示すデータの状態から、プロセスに対する処理を決定し、プロセッサ内を制御する。またプロセス単位の制御とは独立に後続のタスク列(関数列)の転送制御を行う。

I_NETはセグメントから出力される構造データ操作命令をSM内に存在するコントローラに転送し、R_NETはSMから出力される構造データ操作命令の結果としてのアドレスをセグメントに転送する。

本システムにおいて構造変換関数に必要な基本命令を図4に示す。通常のリスト処理における基本演算には、メモリ参照としてCAR,CDR命

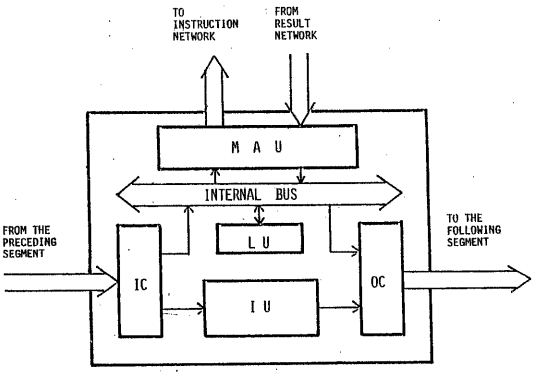


図2．セグメント構成

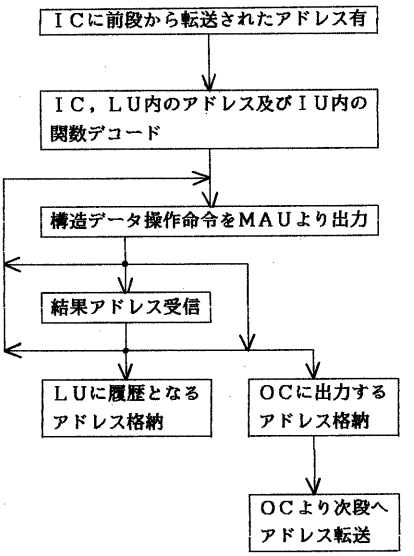


図3．セグメントの基本処理フロー

MAU　構造データ操作命令

Read op1	op1で指定されたセルのＣＡＲ，ＣＤＲの値を返す
Write_CAR op1	セルを１個生成し、そのＣＡＲフィールドにop1を書きこみ、生成したセルのアドレスを返す
Write_CDR op1,op2	op1で指定されたセルのＣＤＲフィールドにop2を書きこむ

LU　履歴命令

Link op1	op1を、次のセグメントタイムで使用するアドレスとして保持する
Link_long op1	op1を、nilが入力となるセグメントタイムまで保持し続ける

OC　出力命令

Send op1	op1を、次のセグメントへ転送する
Send_long op1	op1を、nilがlinkされているセグメントまで転送し続ける

図４．構造変換関数を実行するセグメントの基本命令

令、新しいセルを生成してそのCARフィールド、CDRフィールドに書き込むCONS命令がある。本システムでは先行制御を取り入れるためリストをたどる操作とリストを生成する操作がオーバラップして行われる。メモリ参照はRead命令一つで行い、新しいセルの生成と書込はWrite_CAR命令とWrite_CDR命令に分けて実現される。

3.3 実行例

システムの基本動作を説明するために、一つの構造変換関数が実行される様子を示す。ここでは例として(1)式に示すように転置をとる関数transpose(タスク)を、図５(a)に示すオブジェクト
<<1,2,3>,<4,5,6>>に作用させた場合を考える。

transpose:<<1,2,3>,<4,5,6>>　　(1)

(1)式の結果として、図５(b)に示すオブジェクト<<1,4>,<2,5>,<3,6>>が生成される。ここでtransposeはさらにプリミティブな関数を使って次のように再帰的に表現される。

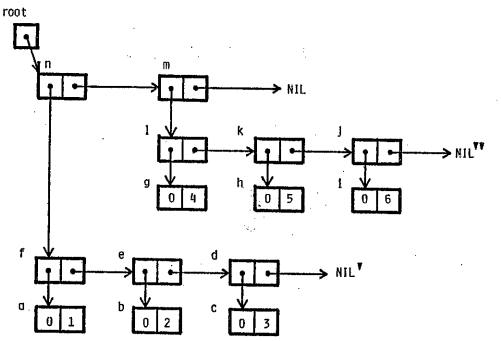


図５(a)．オブジェクト<<1,2,3>,<4,5,6>>の２進木表現

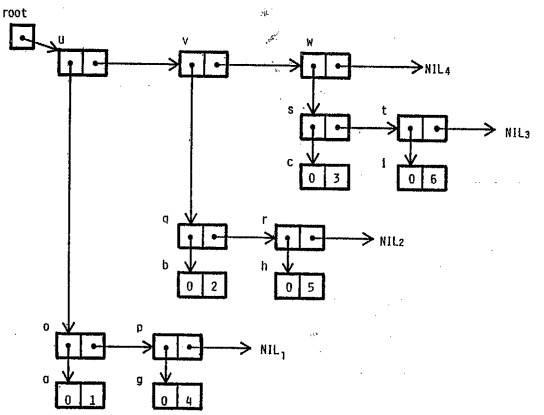


図５(b)．オブジェクト<<1,4>,<2,5>,<3,6>>の２進木表現

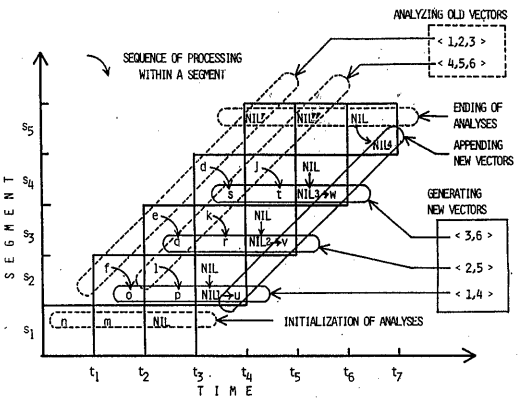


図６．transpose実行の時間空間図

```
transpose = null∘1 → [];
    apndl∘[α1,transpose∘αtail]  (2)
```

(2)式の再帰的表現にもとづいて、transposeがパイプライン上で実行される様子を図６に示す。ここで図中のアルファベットは図５(a),図５(b)のアドレスに対応する。図６において、プロセスは古いオブジェクト<1,2,3>と<4,5,6>を解析する処理となっている。また、各セグメントでは新しいオブジェクト<1,4>,<2,5>,<3,6>を生成する処理が行われている。すなわち、transposeという基本関数（タスク）の実行プロセスがパイプライン処理部に供給されることにより、複数のプロセスに分割されて各セグメントで処理されながらパイプライン上を流れていく。そして各セグメントでは、複数のプロセスの間で必要なアドレス(履歴)がLUを通して受け渡され、プロセスの結合がセグメント内で行われる。

3.4 実行制御に関する考察

前節の例は、transposeをリストデータに作用させる時に行わなければいけない処理を、時間空間の二次元空間にマッピングしたものである。この例では、(2)式における再帰が空間軸方向に進むようにマッピングした場合であるが、時間軸方向に再帰が進むようにマッピングすることも可能である。どちらが効率的かは関数の内部性質や作用させるデータ構造に依存するため明確に決まらない。

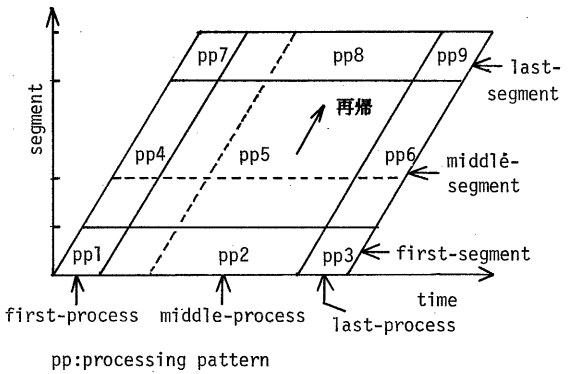
前節の例において、各セグメントタイムでの処理内容に着目した時間空間図を図７に示す。この図よりtransposeという構造変換関数の内部性質から、その処理内容は９種類の処理パターン(pp)に分けられることがわかる。また図７のpp5での最初の状態と実行される命令を図８に示す。

ここでプログラムの制御方法について考えると、大別して次の二つの方法がある。

(1) transposeのように、パイプラインに処理をマッピング可能な関数自体を一つの命令とし、そのような命令の列をプログラムとしてパイプライン処理部に流す。ただし各プロセッサは、関数を実行するマイクロプログラムを全部持っているとする。このとき各セグメントのIUは、入力されたアドレスの示すデータの状態と、履歴として残っているアドレスの示すデータの状態から図７のどの処理パターンを実行するかを決定する。プログラムの転送は、図７におけるfirst-processのみでtransposeの転送を行い、後続の関数はtransposeが最初のセグメントで終了するまでに次のセグメントに転送すればよい。

(2) transposeを構成するさらにプリミティブな関数を中間コードに変換し、そのような中間コードの列をプログラムとしてパイプライン処理部に供給する。この場合IUは、入力と履歴の状態から、プロセッサに入力されるプログラムの中からどの部分を取り入れ、取り入れたプログラムのどの部分を実行するかを決定する。

(1)の場合はプログラムの転送量が少なくなるが、制御記憶の容量が増大する。(2)の場合は、関数とパイプラインの処理内容との対応関係が確立されていないので、中間コード作成が困難である。



pp:processing pattern

図７．処理パターン(pp)に着目したtranspose実行の時間空間図

```
<最初の状態>
関数    transpose
入力    not nil     (解析中のセルアドレス)
履歴    not nil     (生成中のセルアドレス)

<処理内容>
構造データ操作
        Read 入力       ( car , cdr )
        Write_CAR car   ( getcell )
        Write_CDR 履歴 、getcell
履歴    link getcell
出力    Send cdr
```

図８．transposeの実行におけるpp5での処理

４．動作の定式化

３章で述べた基本関数の処理の時間空間二次元空間へのマッピングではセグメントでの処理内容が複雑であり、処理内容をその基本関数の意味から導きだす明確なアルゴリズムは今のところ存在しない。よって関数が大きく複雑化してくると、マッピングが不可能になり、CUによるマッピング可能な関数列の分解と合成等の制御に頼ることによって、複雑な関数の実行が行われることになる。また3.4で述べたように、マッピングの単位となる大きさによってマッピング方法が違ってくる。

そこでパイプラインにマッピング可能な基本関数の動作の定式化を行う。そしてこのような定式化される基本関数を拡張していくことにより、ひとつの体系を作っていき実行制御アルゴリズムの確立を目指す。この基本概念を図９に示す。

まず最初にシステムの構成要素、状態、動作から、プロセッサの動作に関する定義を図10に示す。

プロセッサの動作は９項組S=(P, I, H, O, δ_1, δ_2, δ_3, δ_4, M)で表される。Pはプログラム体系を表し、本稿では構造変換関数の基本関数の集合に限定している。Iは入力の状態の集合を表す。Hは履歴の集合で、何を履歴とするか、および履歴の状態を表す。Oは出力の集合で、何を出力とするか、および出力の状態を表す。δ_1はメモリ操作の決定関数、δ_2は履歴の状態及び動作の決定関数、δ_3は出力の状態及び動作の決定関数である。δ_4は補助関数で、履歴及び出力の動作(何を履歴及び出力としたか)から状態を決定する。Mはメモリ体系で、７項組$M=(M_s, M_i, M_o, M_r, m_1, m_2, m_3, m_4)$から表される。$M_s$はメモリセルの状態の集合、$M_i$はメモリ操作のオペコードの集合、$M_o$はメモリ操作のオペランドの集合、$M_r$はメモリ操作によって返されるアドレスの種類の集合

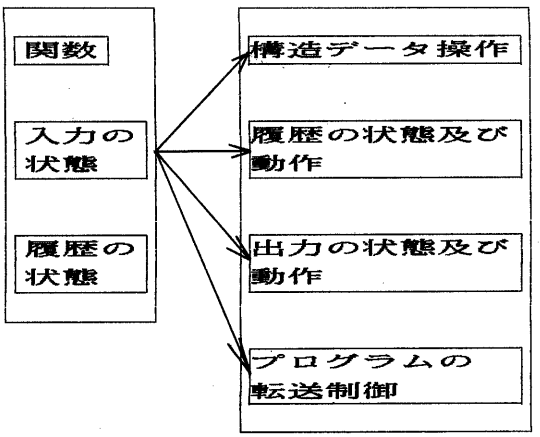


図９.定式化の概念図

$S=(P, I, H, O, \delta_1, \delta_2, \delta_3, \delta_4, M)$

P：プログラム体系
I = { empty, nil, root, normal }
H = { empty, nil, root, normal, M_car, M_cdr, M_cell }
O = { empty, nil, root, normal, M_car, M_cdr, M_cell }

<基本関数>
$\delta_1 : P\times I\times I\times H\times H \rightarrow (M_i\times(M_o)^{m3(Mi)})^3$
$\delta_2 : P\times I\times I\times H\times H \rightarrow H\times H$
$\delta_3 : P\times I\times I\times H\times H \rightarrow O\times O$

<メモリ体系>
$M=(M_s, M_i, M_o, M_r, m_1, m_2, m_3)$
M_s = { nil, normal }
M_i = { Read, Write_car, Write_cdr }
M_o = { In1, In2, M_car, M_cdr, M_cell, His1, His2 }
M_r = { Car, Cdr, Cell, ϕ }

$m_1 : M_i \rightarrow M_r\times M_r$
  m_1 (Read) = (Car, Cdr)
  m_1 (Write_car) = (Cell, ϕ)
  m_1 (Write_cdr) = (ϕ , ϕ)
$m_2 : M_r \rightarrow M_s$
  m_2 (Car) = { normal, nil }
  m_2 (Cdr) = { normal, nil }
  m_2 (Cell) = { normal }
$m_3 : M_i \rightarrow N$　　(N：自然数でオペランドの個数)
  m_3 (Read) = 1
  m_3 (Write_car) = 1
  m_3 (Write_cdr) = 2

<補助関数>
$\delta_4 : H \rightarrow H$　($O \rightarrow O$)
  δ_4 (M_car) = { normal, nil }
  δ_4 (M_cdr) = { normal, mil }
  δ_4 (M_cell) = { normal }

図10. セグメントの動作に関する定義

<pp1>
δ_1 (trans,normal,root,empty,empty) = ((Read,In1),ϕ,ϕ)
δ_2 (trans,normal,root,empty,empty) = (M_cdr,root)
δ_3 (trans,normal,root,empty,empty) = (M_car,empty)

<pp2>
δ_1 (trans,empty,empty,normal,root) = ((Read,His1),ϕ,ϕ)
δ_2 (trans,empty,empty,normal,root) = (M_cdr,root)
δ_3 (trans,empty,empty,normal,root) = (M_car,empty)

<pp3>
δ_1 (trans,empty,empty,nil,root) = (ϕ,ϕ,ϕ)
δ_2 (trans,empty,empty,nil,root) = (ϕ,ϕ)
δ_3 (trans,empty,empty,nil,root) = (root,nil)

<pp4>
δ_1 (trans,normal,empty,empty,empty) = ((Read,In1),
(Write_car,M_car),ϕ)
δ_2 (trans,normal,empty,empty,empty) = (M_cell,M_cell)
δ_3 (trans,normal,empty,empty,empty) = (M_cdr,empty)

<pp5>
δ_1 (trans,normal,empty,normal,normal) = ((Read,In1),
(Write_car,M_car),
(Write_cdr,His1,M_cell))
δ_2 (trans,normal,empty,normal,normal) = (M_cell,normal)
δ_3 (trans,normal,empty,normal,normal) = (M_cdr,empty)

<pp6>
δ_1 (trans,normal,nil,normal,normal) = ((Write_cdr,His1,In2),
(Write_car,His2),
(Write_cdr,In1,M_cell))
δ_2 (trans,normal,nil,normal,normal) = (empty,empty)
δ_3 (trans,normal,nil,normal,normal) = (M_cell,In2)

<pp7>
δ_1 (trans,nil,empty,empty,empty) = (ϕ,ϕ,ϕ)
δ_2 (trans,nil,empty,empty,empty) = (nil,nil)
δ_3 (trans,nil,empty,empty,empty) = (empty,empty)

<pp8>
δ_1 (trans,nil,empty,nil,nil) = (ϕ,ϕ,ϕ)
δ_2 (trans,nil,empty,nil,nil) = (nil,nil)
δ_3 (trans,nil,empty,nil,nil) = (empty,empty)

<pp9>
δ_1 (trans,nil,normal,nil,nil) = ((Write_cdr,In1,His1),
ϕ,ϕ)
δ_2 (trans,nil,normal,nil,nil) = (empty,empty)
δ_3 (trans,nil,normal,nil,nil) = (empty,empty)

図11. transposeのセグメント内処理の定式化記述

を表す。m_1,m_2,m_3はメモリ体系内の関数でメモリ操作に関する意味を与える。

このような定義にもとづいて基本関数transposeを実行するための、状態及び動作決定関数δの記述を図11に示す。このように定式化を行うことによりパイプライン上に分散させるtransposeの処理の中で何を行うかが、プログラムと履歴と入力で一意に決定できることがわかる。

5．まとめ

本稿では関数型言語を用いてリスト処理を行うことを目的としたパイプライン処理システムについて、まず関数型言語とリスト処理にあるパイプライン性について述べ、プログラムを基本関数としての構造変換関数に限定したときの構成、制御について述べた。そして、本システムにおいて構造変換関数を実行したときの動作について示した。また、プロセッサの処理アルゴリズム確立の一ステップとして動作の定式化を行った。

今後の課題としては、プロセッサの処理アルゴリズムを確立するために、5章で述べた体系を拡張し、基本関数として算術関数、論理関数を想定したときの制御方法、CUを含めたシステム全体でのプログラム(関数)の供給方法の検討が挙げられる。また、本システムの有効性について検討を行う必要がある。

［参考文献］
(1) 遠藤,中村,重井:"汎用パイプライン処理における機能割付の最適化",信学技報,EC82-38 (1982-10).
(2) 遠藤,中村,重井:"階層化汎用パイプラインシステム",昭60-7,信学論(D).
(3) 小林,遠藤,中村,重井:"汎用パイプライン処理システムの性能評価", 昭60-10, 信学論(D).
(4) T.Nakamura, K.Sakai and Y.Mishina: "Function Level Computing on the Brain Structured Computer," Proceedings of The IEEE Computer Society's Ninth International Computer Software & Application Conference (Compsac85) 1985.
(5) 坂井,三科,中村,重井:"リスト構造を並列処理するパイプライン処理システムの構成", 昭60 東北支部連大 1C12 .
(6)J.Backus:"Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,"CACM, vol.21.No.8,pp.613-614,Aug.1978.