# データフロー・アーキテクチャに於ける先行評価・遅延評価機構とその並列推論制御への応用

Eager and Lazy Evaluation Mechanism in Data Flow Architecture and Its Application to Parallel Inference Machine

雨宮 真人　長谷川 隆三　清木 康

Makoto Amamiya　Ryuzou Hasegawa　Yasushi Kiyoki

（日本電信電話公社 武蔵野電気通信研究所）

Musashino Electrical Communication Laboratory, NTT

## 1　まえがき

近年、並列処理計算機アーキテクチャに対する関心が高まっている。中でも高度の並列処理を実現する方式としてデータフロー方式が積極的に研究されている。他方、関数型言語、論理型言語との親和性がより良いリダクション型の計算モデルを実現する計算機方式について関心が高まりつつある。特にリダクションモデルについては、遅延評価、高階関数（部分計算）等、計算能力の点でデータフローモデルより優れている。しかしその効果的なハードウェア実現方式という面からはまだ不明確な点が多い。（並列リダクション機構の実現方式としてノイマン型のコントロールフローとパケットコミュニケーションを組み合わせる方式（ALICE〔1〕の方式はこのタイプと考えられる）とデータ駆動型をベースとする方式が考えられる。）

本稿ではデータフローアーキテクチャが高並列のリダクション機構を実現する有効な方式であるとの立場に立ち、その実現の基礎となる先行評価（Eager Evaluation）と遅延評価（Lazy Evaluation）の機構について論ずる。また、本機構を用いた論理型プログラムの並列処理方式を示し、本機構が今後知識処理等で重要になる並列推論処理に有効であることを示す。（なお、高階関数サポートの問題については文献〔2〕に譲る。）

データフローモデルに於いて、データフローグラフ上を流れるトークンに何を対応させるかによりモデルの能力に差が生じる。トークンに直接値を対応させてしまえば原始的なデータ駆動の能力しか持ち得ない（これを by value メカニズムと云う）。一方、トークンに値を保持するセル（へのポインタ）を対応させれば先行評価、遅延評価の実現や高階関数の部分的サポートが可能となる。（これを by reference メカニズムと云う）。

先行評価では、データ生成プロセス（Producer）と消費プロセス（Consumer）がコンカレントに処理され、高度のパイプライン処理効果が得られる。（lenient cons〔3〕はその一例である。）また遅延評価では、Producer にデータの生成を途中で打ち切らせConsumerからの要求が伝えられたとき実行を再開させるという要求駆動型の処理が実現される。（lazy cons〔4〕はその一例である。）

論理型プログラムの実行では OR 並列と AND 並列により高度の並列処理性が期待できるが、変数管理の問題を考慮すると現実には AND 処理を逐次化せざるを得ない。しかし先行評価機構を用いると AND 処理に於いてパイプライン効果を引き出すことが出来る。また、現実の有限資源下では並列処理対象の爆発を如何に抑止するかが重要な課題であるが、この問題を遅延評価によって解決することが出来る。

本稿では、先ず、2章で先行評価と遅延評価の機構とその実現法を述べる。次に、3章で両評価機構の応用としてストリーム処理の例を述べ、Concurrent Prolog〔5〕や PARLOG〔6〕と同様のコンカレント処理やオブジェクト概念が実現されることを示す。最後に4章で先行評価、遅延評価機構を用いた論理型プログラムの実行方式について述べる。

## 2　先行評価（Eager evaluation）と遅延評価（Lazy evaluation）

並列処理において，先行評価及び遅延評価の機能が果たす役割は重要である。本章では，データフローとリダクション型の並列計算モデルをデータメカニズム（引数データの使われ方）に着目して整理し，データメカニズムと先行・遅延評価との関係を明らかにする。

### 2．1　並列計算モデルの整理

データフローは言語のセマンティクスよりむしろ操作に視点を置いた計算モデルであり，データ駆動（data driven）による演算実行が基本原理である。一方，リダクションはラムダ計算のセマンティクスを反映した計算モデルであり，計算を書換え規則の適用とみなす。

表1は，データフロー及びリダクション型の計算モデルをデータメカニズムの差異によって分類したものである。引数として値自身が渡されるものを by value と呼び，値を保持するセルが（概念的に）渡されるものを by reference と呼ぶ。ここでは特に， by reference メカニズムを持つデータフローモデルにおいて，セルが引渡されると直ちにその内容を読み出す方式を data driven と呼び，セルの内容が必要になった時に読み出す方式を demand driven と呼ぶ。

先行評価・遅延評価は引数データに対するアクセス・評価方式と密接に関連しており，計算モデルの差異には依存しない。そこで以下ではデータフローモデルを前提にして，データメカニズムの違いをみることにする。

表1　並列計算モデルの分類

| | by value | | by reference |
|---|---|---|---|
| data flow | data driven | | demand driven |
| reduction | string reduction | graph reduction | |

### 2．2　データフローモデルにおけるデータメカニズム

by value と by reference メカニズムの特徴を以下に示す。

・ by value 型計算（図2．1参照）

値を保持する変数セルの概念は存在しない。変数は単に値を denote（指示）するために使用される。（e.g.,　x = f（..）において，x は f（..）の結果値を示す。）スカラデータも構造データも一つの value として扱われ，トークンとして value 自身が流される。

by value では，引数値が全て得られた後に関数が適用されるため（最内側評価），不要な引数評価がなされたり，場合によっては永久に計算が停止しなくなるなどの問題が生じる。これを避けるため，data driven 方式では control 概念を一部取り入れ，条件式の評価を最左外側から行うような制御ノード（switch）を用意している。

・ by reference 型計算（図2．2参照）

概念上は，全ての変数に対して値を保持するためのセルがとられ，変数セル（又はセルを指すポインタ）がトークンとして流される。セルへのアクセスは，data driven 方式ではセルが到着した時に，demand driven 方式では要求がトークンとして伝えられた時に行われる。

by reference では，値を一度計算し，変数セルに格納する。その後その値が必要になった場合は，セルを参照するだけでよい。セルに対する並列アクセス制御は，セルに同期制

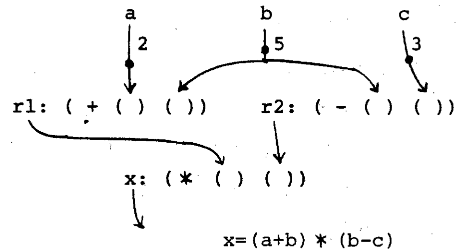


図2.1　by value メカニズム

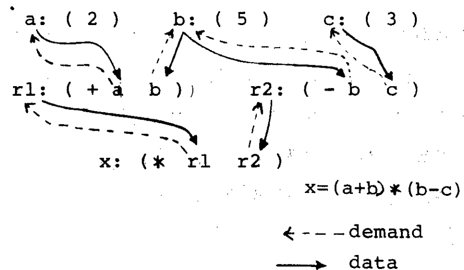


図2.2　by reference メカニズム

御用のタグを付すことにより容易に実現され，異なるプロセス間の通信もこのメカニズムにより統一的に扱えるようになる。しかし，プリミティブ演算の度に毎回メモリ参照が行われるという欠点がある。

## 2．3　先行・遅延評価とデータメカニズム

先行評価・遅延評価は，value（データ構造）を生成する producer とそれを使用する consumer の問題に帰着する。

by value では基本的に先行評価（eager evaluation）が行われるが，最大限の先行評価は consumer が producer に先行して処理を進めることであり，これはデータ構造に leniency の概念〔3〕を導入することによって可能となる。

by reference では，予め値を保持するセルを生成し，これを consumer に引渡せるため，この種の先行評価（パイプライン処理）が可能である。したがって，by reference メカニズムが eagerness を与えていると言える。

一方，value の生成を中断させる遅延評価は by reference メカニズムのみでは行えず，必要に応じて producer を起動する demand driven 方式によって実現可能となる。〔4〕但し，demand driven 方式でも単に引数評価のための要求を伝播するだけでは先行評価と変わらない（実質的な計算が行われるのは最内側であり，要求が内側に向かって送られるため）。遅延評価が達成されるのは，例えば以下のような書換え規則に基づき，最外側評価が行われる場合である。

```
if T then A else B   => A
if F then A else B   => B
car (cons (x,y))     => x
cdr (cons (x,y))     => y
```

## 2．4　by value と by reference の融合

by reference メカニズムを導入すれば，先行評価と遅延評価が共に可能となるが，by reference を徹底するとプリミティブ演算の実行の度に要求の送出，変数セルの参照が繰り返され，効率が低下する。現実的には，by value の利点と by reference の利点を組合せた融合方式が有望である。

ここでは，原則的には（default は）by value で実行し，必要に応じて by reference で実行する方式を考える。以下に本方式の特徴を述べる。

・変数セルはプリミティブ演算に対して取らず，式または関数アプリケーションに対して取る。

・変数セルには以下のように先行・遅延評価制御のためのタグを付す。

| r | c | value ／ recipe |
|---|---|---|

```
r :  ready tag
     0 : not ready
     1 : ready
c :  recipe tag
     0 : value
     1 : closure
```

・式または関数アプリケーションは図2．3に示すように，Bind，Create,Eval ノードを含むデータフローグラフに変換する。これらのノードの機能を以下に示す。

Bind (x,v)

セルxに値vを書き込む。

Create (s)

シグナルsにより起動され，新セルを確保し，そのアドレスを値として送出する。

Eval (x,s)

シグナルsにより起動され，セルxの内容を送出する。このとき，セルxのタグrが not ready を示していれば wait し，タグcが閉包（closure）を示していれば，demand トークンを送出し，wait する。

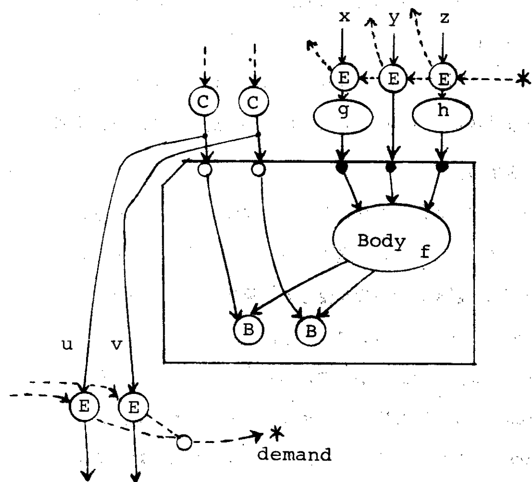



図2.3 by reference による関数起動メカニズム

ここでは Eval の起動を demand トークンによって制御すること (gating) によって遅延評価を実現している。このような制御を行わず，セルが到着すると直ちに Eval を起動すれば，最大限の先行評価が行われることになる。

### 2．5　遅延評価の制御

遅延評価は，式の実行または関数アプリケーションにおいて引数データの本体への引渡しを止めることにより行われる (図2．3において，demand トークンが Eval ノードに到着する迄，引数トークンは本体へ送られない)。遅延評価の指定は delay,force を用いて行う。
例えば， (u,v) = delay f (g (x),y,h (z)) は，図2．3のように変換される。demand の送出は force u, force v と指定することによりなされる。lazy cons 〔4〕はここで述べたメカニズムの特殊形であり，この場合 car, cdr 命令により implicit に force がかけられる。

## 3．マルチプロセスの制御

本章では，2章で述べた先行・遅延評価機構の応用としてオブジェクト及びコルーチンの実現例を示す。

### 3．1　オブジェクトの実現

Concurrent Prolog 〔5〕 のように、内部状態を引数として持ち自分自身を再帰的に呼ぶプロセスとして記述されるオブジェクトは、関数型でも同様に記述することができる。Program 1 にカウンタ・プロセスの例を示す。(注1)

メインの関数 manycounters (sig) は、シグナル sigにより起動され、 input, umc, ct, output をそれぞれ並列に評価する。

input は端末からコマンドを一つずつ受取り、それをメッセージストリームとして送出する。ここではメッセージストリームは、遅延評価機構を使って無限リスト〔read (sig) | input (sig)〕の形で生成されている。

umc は, input から送られてくるメッセージのうち 'create' を拾いあげ，プロセス名を付してカウンタプロセス (usc) を生成する。 usc は自プロセス名に対するメッセージを抽出し，それをカウンタオブジェクト ct に伝える。カウンタオブジェクト ct は、'clear', 'up', 'down' などのメッセージを受け取ると、それに対応して自己の内部状態を変更するプロセスである。〔show. X〕メッセージを受け取った場合は、関数 bind によって変数Xと、現在の内部状態を cons し、outputに通知する。(関数型の実行にお

---

(注1)　この例は関数型言語 Valid-E (Extention) による記述例である。

関数の引数の授受、値定義等において、パタン照合によるデータ選択を許している。同様に関数の invocation もパタン照合により行われる。関数定義では、同じ関数名を持つ定義が複数個存在するが、パタン照合で起動される関数定義は唯一、定まらなければならない。

```
Program 1
-- counter process --

manycounters(sig)
= output(s)
  where{c=input(sig),
        s=umc(c)}

umc([['create.nam].nmsg])
= [z|umc(nmsg)]
   where{s=usc(nam,nmsg),
         z=ct(s,0)}.
umc([cmd.nmsg])=umc(nmsg).

usc(nam,[[nam.op].nmsg])
= if nam=naml then [[nam.op]|usc(nmsg)]
  else usc(nmsg).
usc(nam,[['delete.nam].nmsg])= [].

ct(['clear.nmsg],state)= ct(nmsg,0).
ct(['up.nmsg],state)= ct(nmsg,state+1).
ct(['down.nmsg],state)= ct(nmsg,state-1).
ct([['show.v].nmsg],state)
= [bind(v,state)|ct(nmsg,state)].

bind(x,y)= [x.y].

input(sig)= [read(sig)|input(sig)].
output([cs.ns])= {print(cs); output(ns)}.


-- Note --
 where{...} construct defines values
which are used in the expression
directly in front of it.
 Pattern is written in a special
list notation,
   [x.y] == cons(x,y)
   []    == nil
   [x,y.z] == [x.[y.z]]
   [x,y,z] == [x.[y.[z.[]]]] .
 Pattern matching rule is
   [x.y]=[a,b,c] == x=a, y=[b,c] ,
   [[a,x.y].z]=[[a',b,c].b]
      ==> x=b, y=[c], z=b  if a=a',
          otherwise failure.
 [x|y] is the abbreviation of [x.delay y].
  i.e. [x|y]=exp ==> x=car(exp),
                     y=force cdr(exp) .
```

いても、Lenient consのような by reference メカニズムがサポートされている場合は、このような back communication は実現可能である。）

〔特徴〕

・履歴，即ち状態は無限リスト（tail recursive data structure）の形で保持される。

・プロセス間通信は by reference メカニズムを利用することによって行われる。すなわち、メッセージストリームを形成する共有変数セルは、コンカレント・プロセス間の通信チャネルとみなすことができる。

・状態の共有、参照も by reference メカニズムによる。ただし変数セルを破壊的に使用することはなく、あくまで applicative に用いられる。

・状態の伝達は、引数渡しにより明示的に行われる。なお，任意個のプロセスがカウンタと通信しようとする場合には、任意個のメッセージ・ストリームを一本のストリームにまとめる必要があるが，これは non-strict merge 機能によって実現される。

## 3．2 コルーチンの実現

ほかのプロセスの実行を再開したり、自分のプロセスを停止させたりするコルーチン的機能は、先行・遅延評価機構によって効果的に実現される。

Program 2 にコルーチン制御の例を示す。(図3．1参照)

fibo,pasc,prim は、それぞれ Fibonacci 数列，パスカルの三角形，素数列を生成するプロセスで，起動されると概念的には無限の数列を生成する。

inp は、カウンタの例と同様に、端末からのコマンドをメッセージ・ストリームとして送出する。コマンドは、〔seqname, count〕の形で与えられる。（seqname で指定された数列に対し、count で指定された個数だけ出力せよという指示）

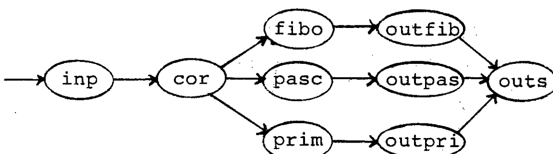



図3.1 コルーチンの例

cor は、inp からメッセージを受け取ると、メッセージ中に書かれた数列名に従って、メッセージをそれぞれの数列の出力ルーチンに振り分けるだけの働きをする。

例えば、cor が〔 'fib, count 〕メッセージを受け取った場合、そのメッセージは outfibに渡され、ここで count で指定された個数分だけ数列が出力される。

〔特徴〕

・遅延評価機構を利用し、あらかじめストリームの形で無限数列を生成しておく。このような無限系列を仮定すると、プログラミングは、容易になる。

・ストリームの送出の中断・再開は、delay force の機能による。

・遅延された関数の force は、recipe タグの付されたセルがアクセスされた時に自動的に行われ、プログラマは、陽

```
Program 2
-- coroutine --

coroutine(s)
= cor(m,u,v,w)
  where{m=inp(s),
        u=fibo(s),
        v=pasc(s),
        w=prim(s)}.

inp(sig)=[read(sig)|inps(sig)],

cor([['fib,count].nmsg],u,v,w)
= cor(outfib(u,count),v,w).
cor([['pas,count].nmsg],u,v,w)
= cor(u,outpas(v,count),w).
cor([['pri,count].nmsg],u,v,w)
= cor(u,v,outpri(w,count)).

outfib(u,count)= outs(u,count,'fib).
outpas(v,count)= outs(v,count,'pas).
outpri(w,count)= outs(w,count,'pri).

outs([e|x],count,name)
= if count=0 then [e|x]
  else outs(x,count-1,name)
       where{s=print(e,name)}.

fibo(s)= [1|[1|addlist(fibo(s)]].
addlist([e,x1|x])
= [e+x1|addlist([x1|x])].

pasc(s)= pastri([1|zeros(s)]).
pastri([v1|v])
= [[v1|v]|pastri([1|vsum([v1|v],v)])].
zeros(s)= [0|zeros(s)].
vsum([u|x],[v|y])= [u+v|vsum(x,y)].

prim(n)= sieve(n,ints(2)).
ints(m)= [m|ints(m+1)].
sieve(n,[e|m])= [e|sieve(n,del(e,m))].
del(m,[e|n])
= if remainder(e,m)=0 then del(m,n)
  else [e|del(m,n)].
```

に force をする必要はない。
・count で指定された個数だけ eager に数列を生成し、そこで停止させている。
このように eager evaluation の回数を指定する方法は、資源管理の面で有用である。

## 4 論理型プログラムの実行制御

本章では2章で述べた先行評価，遅延評価の方法を利用して論理型プログラムの並列実行〔7〕）を制御する方法について論ずる。 以下の議論では Horn Clause の実行を前提に議論する。 すなわち、Clauseは

P − A1，A2，．．．Am. m＞0 （1）

の形式に表現される。ここに、P，Ａｉは atom である。Ａ1，．．．Ａmは and 結合されている。特にＰを head 、Ａｉを body とよぶことにする。

atom はR（ｔ1，．．．ｔｎ）の形式で表現される。Rは relation name 、ti は term である。一般に term は constant、variable、function のいずれでもよい。式（1）はＰが真となるのは各Ａｉが全て真となるときであることを意味する。ｍ＝0のときはＰが無条件に真であることを意味する。Horn Clause の集合をデータベースとよぶことにする。

### 4・1 論理型プログラムの並列実行

論理型プログラムの実行では Goal （ and結合された atoms ）中の atom ひとつひとつをデータベース中の同じ relation name を持つ head と Unify して行く。このとき論理型プログラムの実行には次の点で並列処理が期待できる。

（1）ＯＲ並列 — Goal atom との Unify に成功した複数の Clause の body を同時並列に実行する。

（2）ＡＮＤ並列 — ひとつの Clause の body 実行に於いて各 atom の実行を並列に行う。

（3）並列Unification − Goal atom と同一の relation name を head に持つ Clause の探索及び各 head との Unification を並列に行う。以下の2面で並列性がある。

（3・1） head atom の探索を並列に行う（並列探索）

（3・2） atom中の各termの変数置換を並列に行う（並列置換）

並列処理の対象をprocess と呼び、ＯＲ並列、ＡＮＤ並列、並列探索、並列置換をそれぞれ OR Process ，AND Process， Search Process，Substitution Processと呼ぶ。

atom 中の変数管理の問題を抜きにすれば、OR Process，AND Process ともに完全に並列実行が可能であり実行過程は図4・1 のように証明木で模式化できる。しかし現実には atom中の変数（値）の管理が重要な問題であり、この点を考慮すると変数共有のある atom 間の並列処理には大きなオーバヘッドを伴うことになる。このため処理効率の点からは AND Process の処理を逐次化したほうが良い。

Substitrution Process についても、Process 間に依存関係があるので同様である。一方、OR Process については Clause 間の変数共有がないので完全な並列実行が可能である。 Search Process についても同様、変数共有の問題はない。

このときの証明木は OR 木として図4・2のように表される。

AND Process の処理において、変数を共有する AND Process 間の処理を逐次化せざるを得ないが、非決定性処理を伴う論理型プログラムの実行では変数の値が複数個存在するので個々の変数値を Sream 化すれば AND Process ではパイプライン処理が可能となりこの点での並列処理の効果は大きい。

以下変数管理の問題に着目してAND Process 処理、OR Process 処理について述べる。処理のイメージを図4・3に示す。

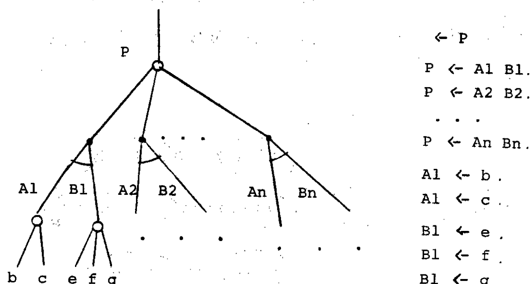



図4.1 AND-OR 証明木

変数の管理には2章に述べた by reference 方式用いる。変数・値の対をしめすのに cell 〔varnam・value 〕を用いる。 valueフィールドに値が到着していない cell を未定義cell、値が入っている cell を定義cell と呼ぶ。未定義cell の場合その value フィールドに未定義を示すタグが付され、定義cell の場合にはその値が非変数term か変数cellポインタかを示すタグが付される。

body中の atom の実行順序は body の実行に先立ってスケジュールされているものとし Clause body の処理は左から右に進むものとする。 body 中の変数については最左位置にあるものがその値を生成し、他はそれを参照することになる。また head 中の変数については Goal atom との Unification 時に値が定義された変数(以下 def-var という)と未定義の変数(undef-var という)が決定され、それぞれに対して定義cell、未定義cell が生成される。(Unification 機構については文献〔7〕を参照)。

body中の atom の処理では、その処理環境として変数cell のリスト(木構造をなす)が与えられ、atom 処理の結果、新しい環境が生成される。一個の atom 処理の結果、複数の変数値が生成されるときはこれらの変数値がリスト化されて次の atom 処理に渡される。環境(変数リスト)はその時点で成功に至った処理結果を示している。 body 処理の結果求められた変数値は上位(call側)atom 処理の環境へ引渡される。

例えば Clause

P (x,z) :- Q (x,y) R (Y,Z) S (y,z)

では、Goal が P (a,xx) のとき、head P (x,z) が x の値 a を生成し、以下 Q (x,y) が y の値、S (y,z) が z の値を生成する。また Q (x,y)、R (y,z)、S (y,z) の処理環境 e0, e1, e2 は、例えば、

e0 = [x:a],[y:undef]

e1 = [x:a],[y:b1];[x:a],[y:b2];
　　. . . [x:a],[y:bn]

e2 = [y:b1'],[z:undef];[y:b2'],[z:undef];
　　. . . [y:bm'],[z:undef]

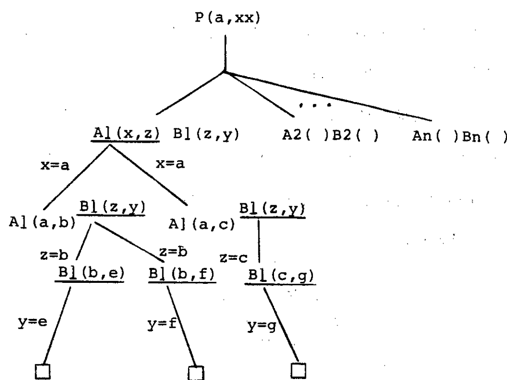
となる。そして S (y,z) の処理結果作られる環境、

ef = [y:b1"],[z:c1];[y:b2"],[z:c2];
　　. . . [y:bk"],[z:ck]

は、このbody 処理の結果作られる新しい環境となり、P (a,xx) の処理結果として、環境

ep = [xx:c1];[xx:c2]; . . . [xx:ck]

が得られる。




```
← P(a,xx).

P(x,y) ← A1(x,z) B1(z,y).
P(x,y) ← A2(    ) B2(    ).
 . . .
P(x,y) ← An(    ) Bn(    ).
A1(a,b).
A1(a,c).
B1(b,e).
B1(b,f).
B1(c,g).
```

図4.2 OR 証明木

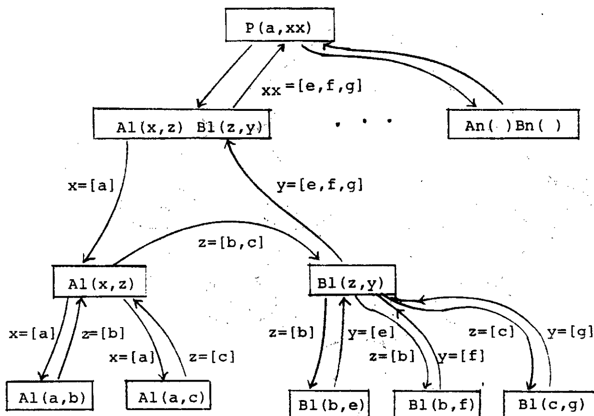



図4.3 AND, OR Process の処理イメージ

### 4・2 OR Process 処理

OR Process 処理部には環境と処理すべき atom が与えられる。処理手順は以下の通り。（図4・4）

（1）データベースを探索し、同一 relation name を head に持つ clause を OR1 Process として Fork する（Search Process）。この際、Fork した OR1 Process の実行結果を新環境として保持するための環境 cell を作りリスト化しておく。（lenient cons 機構により、OR1 Process の結果は自動的に環境cellに書き込まれる。OR1 Process が失敗した時は failure 情報が cell に書き込まれる。）

（2）木構造で与えられた環境を個々の処理環境に分解し、個々の処理環境に対して AND Process を Fork する。このときも（1）と同様にして環境cell のリストを生成する。（値の書込みは AND Process が行う。）

（3）Fork した OR1 Process、AND Process からの実行結果（変数・値のリスト）を Join して新環境を生成する。（実際には lenient cons 機構により OR1 Process、AND Process の中で環境値の書込みが行われる。）

### 4・3 AND Process 処理

AND Process 処理部には（OR Process 処理部によってばらされた）個々の環境と clause head とが渡される。

処理手順は以下の通り。（図4・5）

（1）Substitution Process を起動して変数・値の引渡を行い，実行環境を初期設定する。

この際 def-var，undef-var が決定される。

（2）def-var，undef-var のセットから body 中の atom の実行順序をスケジュールする。（スケジュールの方策については例えば def-var を多く持つ atom を先に実行する。）

（3）body 中の atom をスケジュールされた順に処理する（AND1 Process）。

現環境と先頭 atom をパラメータとして OR Process 処理部を起動する。次に、OR Process 処理部より返された新環境を用い次の先頭 atom を処理する。これを body 中の全ての atom について行う。

（4）最終的に作られた環境を基に変数・値の引渡を行う（Return Process）。

トップレベルの Goal atom の処理では Goal atomの処理の結果求められた環境からその木をたどって個々の変数値を集め出力する（Gather Process）。

以上先行評価機構を用いることにより、OR Process、AND Process、Gather Process の間で高度のパイプライン処理と並列処理が達成されることになる。より具体的なアルゴリズムを Program 3 に示す。

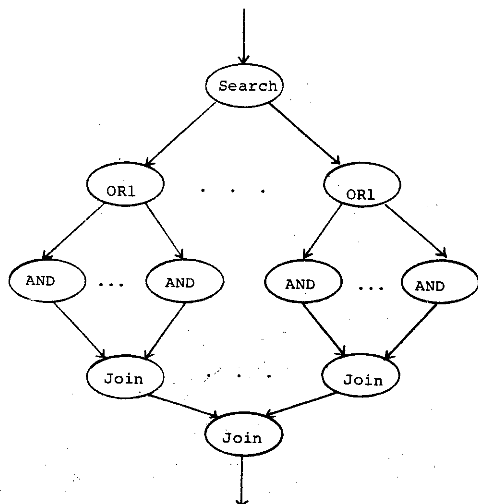



図4.4 OR Process

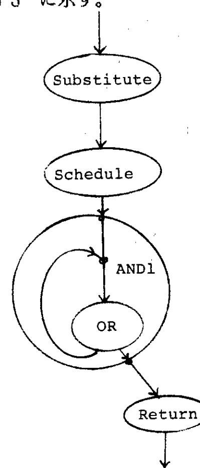



図4.5 AND Process

Program 3

```
--- Gather Process ---
Gather([x.y])
= if x is leaf then [x.Gather(y)]
  else append(Gather(x),Gather(y)).
Gather(['failure.x])=Gather(x).
Gather([])=[].

-- OR Process --
ORprocess(database,atom,env)
= Activateclause(clauses,atom,env)
   where{clauses
          = SearchDB(database,relname(atom))}.

Activateclause([clause.restclauses],atom,env)
= [env1.env2]
  where{
    env1=OR1process(head,atom,env),
    env2=Activateclause(restclause,atom,env)}.
Activateclause([],atom,env)=[].

OR1process(head,atom,[env1.env2])
= [x.y]
   where{
      x= if env1 is leaf then
              ANDprocess(head,atom,env1)
          else OR1process(head,atom,env1),
      y= if env2 is leaf then
              ANDprocess(head,atom,env2)
          else OR1process(head,atom,env2)}.
OR1process(head,atom,['failure.env])
= OR1process(head,atom,env).
OR1process(head,atom,[env.'failure])
= OR1process(head,atom,env).

-- AND Process --
ANDprocess([clausehead.clausebody],atom,env)
= Returnprocess(retvars,env1)
   where{(retvars,env0)
           = Substitute(clausehead,atom,env),
         body= Schedule(clausebody,retvars,env0),
         env1= AND1process(body,env0)}.

AND1process([atom.rest],env)
= AND1process(rest,newenv)
   where{newenv=ORprocess(DataBase,atom,env)}.
AND1process([],env)= env.
```

4・4 OR Process の爆発抑止

先行評価機構を用いることにより、かなりの並列処理効果が得られることになるが、このままでは逆に Process 生成の爆発をきたしてしまう。具体的に言うと、 Program 3 での Activateclause が全ての OR Process を起動してしまうことが爆発の原因である。この問題は、遅延評価機構と counter をもちいることにより解決することが出来る。

Program 4 の Activateclause のように counter を用いて遅延評価 ( delay ) を制御し OR Process の生成を n 個に留めておく。 未生成の OR Process は後に demand が伝えられたときに自動的に生成、起動される。この場合、demand はトップレベルでの変数値収集 Process ( Gather Process ) より出される。(変数値収集の際、必要個数の値が求まってないときに demand が発せられることになる。)

この実行方式は遅延評価により Bounded Parallel 実行を実現しており、 Depth First 処理と Breadth First処理を融合したものである。(実際、 count の方式を変えることにより Depth First 実行と Breadth First実行を適宜切り換えることも可能である。)

Process の生成を n 個に留めた場合の実行過程例 (OR 木) を図4・6に示す。またn=3のときの実行過程例を図4・7(この木でn=1としたときは Depth First 実行になることに注意。)に示す。さらに Gather Process がこの OR 木より値収集した結果の OR 木を図4・8に示す。図では値の必要個数が5以上の場合 rest-environment に(自動的に) demand が伝えられることを示している。

Program 4

```
-- Top Process --
Countvalues(append(Gather(finalenv))).
-- here, finalenv is
-- the goal atoms processing result.

Countvalues([v.x],n)
= if n=0 then x
  else Countvalues(x,n-1).

-- Gather Process --
--Gather(env,count)yield(values,restenv)
Gather([x.y],count)
= if count=0 then delay Gather([x.y],n)
  elsif x is leaf then
     ([x.u],v) where{(u,v)=Gather(y,count-1)]}
  else (append(u1,u2),en)
    where{en= case{null(v1)->v2,
                   null(v2)->v1,
                   other->[v1.v2]},
          (u1,v1)= Gather(x,count),
          (u2,v2)= Gather(y,count-1))}.
Gather(['failure.x],count)= Gather(x,count-1).
Gather([],count)= ([],[]).

-------------------

Activateclause([clause.restclauses],atom,env,count)
= [env1.env2]
   where{
    env1= OR1process(head,atom,env,count),
    env2
     = if count=0 then
         delay Activateclause(
                 restclause,atom,env,n)
       else Activateclause(
                 restclause,atom,env,count-1)}
Activateclause([],atom,env,count)= [].
```

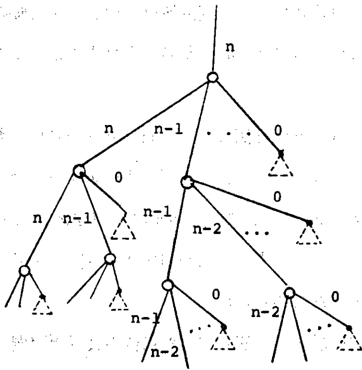


図4.6 Bounded Parallel 実行例

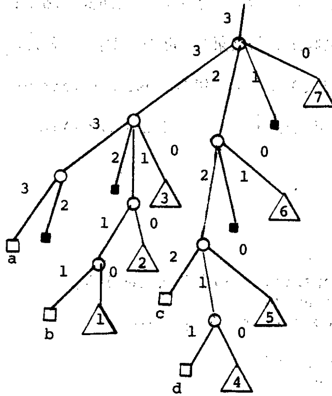


図4.7 n =3 のときの実行例

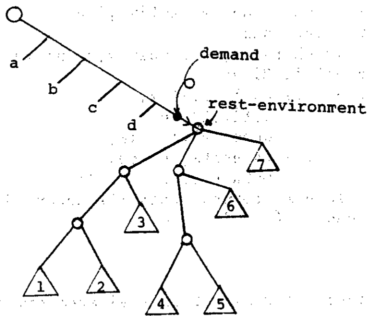


図4.8 値収集の結果得られた値と未評価 OR 木

## 5 あとがき

以上 by reference 概念を導入してデータフローモデルに於ける先行評価、遅延評価の機構について述べ、両機構を用いた stream 処理の例を示した。また先行評価、遅延評価を有効利用し、OR Porcess 処理、AND Process 処理に於ける並列処理性の最大限抽出と必要以上の OR Process の生成抑止を考慮した論理型プログラムの並列実行制御方式を示した。

Concurrent Prolog や PARLOG のように、don't care のセマンティクスに制限してまで論理型言語でコンカレント処理やオブジェクト概念を実現することについては議論の余地がありそうである。3章に示したように先行評価、遅延評価を用いることにより関数型言語の枠組みで同様のことが可能だからである。論理型言語によることのメリットは変数・値の結合機構が Unification の中に組み込まれているため変数管理が自動化され、通信変数〔6〕等の概念がスマートに実現出来る点にある。しかし、これも Concurrent Prolog〔5〕に於ける Queue の例のように直観的に理解し難いという問題がある（先行評価概念により同一のことがより直観的に記述出来る）。

論理型言語のメリットを真に生かすためには don't know セマンティクスをサポートすることが必要である。従ってこれを並列実行する場合には変数・値の管理を含めて OR 並列と AND 並列の効果的処理法の確立が重要な問題となる。この観点から本稿では、先行評価機構を利用して OR Process 、AND Process 処理の並列性を最大限引き出すと同時に、遅延評価機構を利用して並列処理 Process の爆発を抑止する実行制御方式の提案を行った。

先行評価、遅延評価機構については現在設計・試作中のデータフローマシン・プロトタイプ DFM で実現することになっている。実際の効果については DFM による実験およびシミュレーションにより評価データを収集する予定である。

最後に，御討論頂いた第8研究室の諸氏に感謝致します。

〔参考文献〕

〔1〕Darlington J. and Reeve M.J., ALICE : a multi-processor reduction machine, In Proc. Conf. on Functional Programming Language and Computer Architecture, ACM, New York, pp.65-75, 1981.

〔2〕Ono S, Takahashi N. and Amamiya M., Partial Computation with a Data Flow Machine, R.I.M.S. Symposium on Mathematical Methods in Software Science and Engineering, 5th Conference, Kyoto Univ. 1983.

〔3〕Amamiya M., Hasegawa R. and Mikami H., List Processing with a Data Flow Machine, Lecture Notes in Computer Science, R.I.M.S. Symposia on Software Science and Engineering, Springer-Verlag, 1982.

〔4〕長谷川、雨宮、 データフローマシン上での Lazy Evaluation の実現について，第24回情処全大5D-8.

〔5〕Shapiro E.Y., A Subset of Concurrent Prolog and Its Interpreter, The Weizmann Institute of Science Rehovot 76100, ISRAEL, 1983.

〔6〕Clark K.L. and Gregory S., PARLOG : A Parallel Logic Programming Language, Rsearch Report DOC 83 ／5 Imperial College of Science and Technology, 1983

〔7〕雨宮、長谷川、 データフロー制御による論理型プログラム実行機構、Proc. of THE LOGIC PROGRAMMING CONFERENCE, Sponsored by ICOT, 1983.