関数型記号処理言語の分散システムに適した評価法†

沼尾正行†† 志村正道††

分散処理システムで，グラフリダクション方式により関数型言語を評価するためには，システムを構成する各プロセッサにグラフを分散して配置する必要がある．グラフが各プロセッサに分散していると，リダクションを行うために他のプロセッサ内の記憶装置を参照したり，書き換えたりすることが必要となる．また，複数のプロセッサで一つのグラフを書き換えるため，グラフのアクセスに対して危険領域を設定しなければならない．本論文では，これらの煩雑な問題を解消するため，ノード単位でリダクションを行う方法を述べる．この方法では，プロセスがグラフの各ノードに割り当てられており，アークを通して互いに通信し合う．これらのプロセスにより，リダクションがノード単位で行われるので，グラフを分割して各プロセッサに割り当てることが容易である．リダクションの基本操作はノードの消去とノードのコピーであり，これらの二つの操作を組み合わせることにより，リダクションが行われる．ノード単位でリダクションを行うことにより，グラフを各プロセッサの共用データとする必要がなくなるため，共用データにアドレスを割りふったり，アクセスを行うプログラムに危険領域を設けたりする必要がなくなる．このため，大規模で拡張性の高い分散リダクションシステムを構築することが可能となる．

1. まえがき

単一プロセッサ上で関数型言語を処理する方法としては，スタックを用いた逐次実行方式が一般的である．しかし，関数型言語はむしろ多数のプロセッサによる分散処理に適している．これは，プログラムの各部分の評価を他の部分の評価と独立に並行して行えるためである．

このような研究として，たとえば，プロセッサをツリー状に接続し，末端のプロセッサに関数の各部分の評価を，それ以外のプロセッサに末端のプロセッサ間の通信を行わせるシステム[1)~3)]や，パケットプールに溜めこまれたパケットを多数のプロセッサによって書き換えるシステム[4)]，データフローマシンにより実行するシステム[5)]などが提案されている．

一方，関数型言語を処理する方法として，関数型言語の特徴をうまく利用したグラフリダクション方式が提案されている[6)]．この方式はプログラムを同値なプログラムによって繰り返し書き換えることで結果を得るものであり，関数型のプログラムの各部分は同時に書き換えてゆくことができることから，並列実行にも適している．

分散処理システムで，並列にグラフリダクションを行う方法が提案されている[2),3)]．このような方法では，グラフリダクションにおいてプログラムを表現しているグラフを，各プロセッサに分散して配置しなければならず，したがってグラフのリダクションのためには，他のプロセッサ内の記憶装置を参照したり，書き換えたりする機構が必要となる．このような機構を実現するためには，システム全体を統一するアドレス空間を設けて仮想アドレス方式により互いの記憶装置間で相互に参照を行うことが必要になる．しかし，このためには各プロセッサにアドレスを割りふらねばならず，分散処理システムの構成を稼動前に固定する必要が生じ，システムを拡張したり他の分散処理システムと結合したりする場合に問題が生じる．

また，複数のプロセッサで一つのグラフを書き換えるため，グラフのアクセスに対して危険領域 (critical region) を設定しなければならない．

これらの問題は共用 (shared) データを使用したシステムに固有の問題であり，共用データをなくすことで避けることができる．共用データを用いずに分散処理システムを構成する方法としては，通信し合うプロセスを用いる方法が有効である[7),8)]．この方法では，システムを構成するプロセス間に共用データが存在せず，プロセス間の情報交換が通信によるメッセージ交換だけに制限される．このため，危険領域を設定する問題や共用データにアドレスを割りふる問題がなくなるので，システムを構成するのが容易になる．

本論文では，通信し合うプロセスによってグラフリダクションを行う方法を述べる．まず，グラフのノードの消去およびノードのコピーの二つの操作を組み合わせることにより，グラフを書き換える操作が実現されることを示す．さらに，この二つの操作を通信し合

† Evaluation of the Functional Symbol Manipulation Language in Distributed Systems by MASAYUKI NUMAO and MASAMICHI SHIMURA (Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology).

†† 東京工業大学工学部情報工学科

うプロセスによって行う方法を述べる.

2. グラフの構成とリダクション規則

2.1 リダクションの例

グラフリダクション方式においては，プログラムはグラフによって表現され，リダクションはグラフの一部を書き換えることによって実現される．たとえば，次のようなフィボナッチ数列を求めるプログラムにおいて，Fib を評価した場合を考える.

```
Fib≜(1, 1|Mapcar(Plus, Fib1(Fib)))
Fib1≜lambda((X),
        ((car(X), cadr(X))|Fib1(cdr(X))))
Mapcar≜lambda((F, L),
        (F(car(L))|Mapcar(F, cdr(L)))).
```

ただし，$(a|b)$ は cons(a, b) を，$(a, b, c, \cdots)$ は list $(a, b, c, \cdots)$ を表すものとする.

リダクションによる評価は，名前の処理を繰り返して，次のように行われる.

```
Fib
⇒(1, 1|Mapcar(Plus, Fib1(Fib)))
⇒(1, 1|
    (lambda((F, L),
        (F(car(L))|Mapcar(F, cdr(L))))
      (Plus, Fib1(Fib))).
⇒(1, 1, Plus(car(Fib1(Fib)))|
      Mapcar(Plus, cdr(Fib1(Fib)))).
```

本論文で示す方法においては，名前および評価環境が NAME ノードと LAMBDA ノードからなるグラフで表現され，ノードの消去によって名前の処理が行われる．仮引数 L を実引数 Fib1(Fib) で書き換える操作を**図1**に示す．ノード NAME : L および LAMBDA : (F, L) にはプロセスが割り当てられており，図の操作は次の過程を表している.

(1) ノード NAME : L がノード LAMBDA : (F, L) に対してメッセージ 'NAME(L)' を送る.

(2) メッセージの要求に従って，ノード LAMBDA : (F, L) がノード NAME : L と実引数とを接続する.

(3) 最後に両ノードが消去されて，名前の処理が完了する.

名前の処理が，グラフの生成時ではなくグラフのリダクション時に行われることによって，次のような利点が生じる.

a) ソースプログラムをグラフに変換する際の名前の処理が不要となるので，グラフの生成アルゴリズムが単純で高速なものとなる.

b) リダクションの進行に応じて名前を処理するために，動的スコープによる変数評価が可能となる.

また，リダクションの効果で最初の評価時に NAME ノードが消去されるため，名前の処理が評価のオーバヘッドとなることはない.

Fib1(Fib) をリダクションすると，次のようになる.

```
Fib1(Fib)
⇒lambda((X),
    ((car(X), cadr(X))|Fib1(cdr(X))))(Fib)
⇒((car(1, 1|Mapcar⋯),
    cadr(1, 1|Mapcar⋯))|
    Fib1(cdr(1, 1|Mapcar⋯)))
⇒((1, 1)|
    Fib1(1|Mapcar(Plus, Fib1(Fib)))).
```

上で示されたリスト操作のうち，リストの car を取り出す操作をノードの消去によって行うと**図2**に示されるようになる．ノード CAR および CONSにはプロセスが割り当てられており，図の操作は次の過程を表している.

(1) CAR ノードが CONS ノードに対してメッセージ 'SEL(CAR)' を送る.

(2) メッセージの要求に従って，CONS ノード

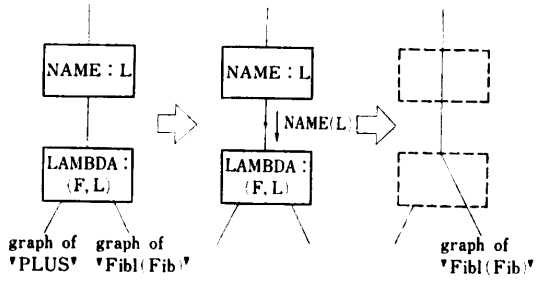


図 1 名前の処理
Fig. 1 The reduction of identifiers.

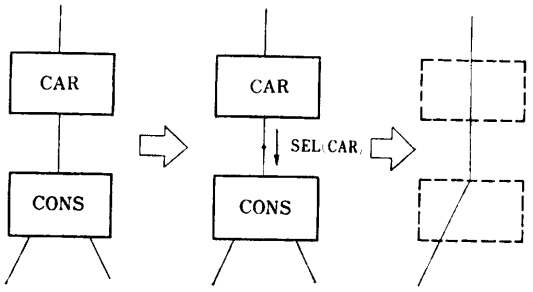


図 2 リスト操作
Fig. 2 The manipulation of lists.

が CAR ノードと CAR 側の子ノードを接続する.

(3) 最後に両ノードが消去される.

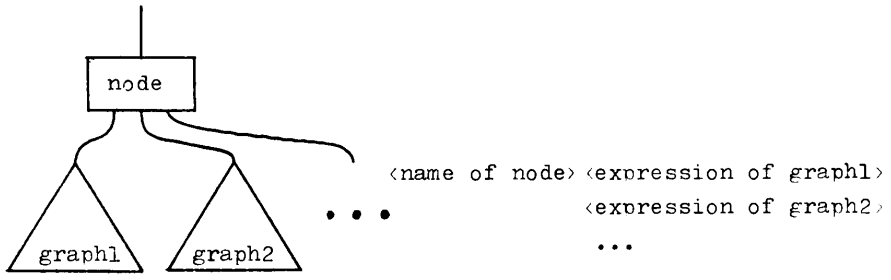
以上述べたように, 名前の処理やリスト操作が隣接したノード間の通信によって行われるようにグラフが構成されている.

2.2 グラフの構成

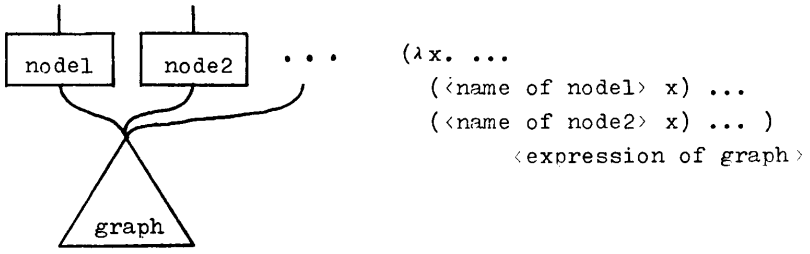
図3に示すような構文をもつ関数型言語を例にとって, グラフの構成およびダクションをさらに詳しく述べる. 以下, **図4**に示すように, リダクションに使用するグラフを λ 論理に基づく記法で表現する. さらにグラフの記述の簡略化のため, (CONS *a b*) を [*a*|*b*] で, (CONS *a*(CONS *b*(CONS *c* ⋯))) を [*a*, *b*, *c*, ⋯]で表して, グラフの構成およびリダクション

```
<expression> == [ <function> ] <list> ¦
                <variable> ¦ <constant> ¦ <lambda expression>

<list>       == ( <expression> {,}... [ '¦'<expression> ] )

<function>   == <expression>

<lambda expression> == lambda( <parameter list>, <expression> )

<parameter list> == ( <parameter list> {,}... ['¦'<variable>] ) ¦
                    <variable>

<variable>   == <capital letter> <letter>...

<constant>   == <atom>

<atom>       == <small letter> <letter>... ¦ <number>

<global definition> ==  <variable> = <expression>
```

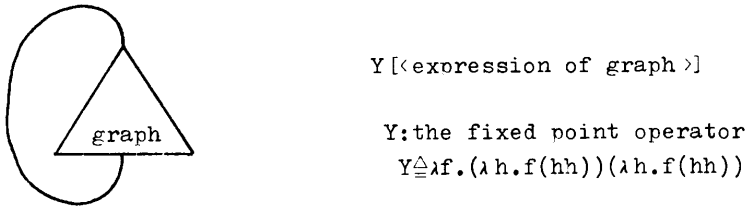
図 3 関数型言語の構文
Fig. 3 The syntax of the functional language.



(a) nodes



(b) sharing



(c) loop

図 4 グラフの表現
Fig. 4 The expression of graphs.

を規則 (rule) の形で記述する.

グラフ全体の構成は次のようになる.

(λenv. PRINT 〈expression〉*)

Y[λenv. LAMBDA : 〈グローバル関数名のリスト〉

[〈グローバル関数定義のリスト〉*]].

* のついた項は, それぞれ**図5**に示されたグラフの生成規則にしたがって, プログラムをグラフに変換したものである. PRINT ノードは3.3節で述べるように 〈expression〉* の評価結果を出力装置上に表示するためのものである. たとえば, 前述したフィボナッチ数列を求めるプログラムは, **図6**に示されたグラフに変換される. フィボナッチ数列を求めるプログラムでは評価すべきプログラムは Fib であるので, 〈expression〉* は図5の規則 (T 4) を用いて NAME : Fib env となる. グローバル環境として, LAMBDA ノードによって Fib, Fib1, Mapcar および Plus の定義が結合され, ローカルな名前と同様なメカニズムで参照されるようになっている.

```
(T1) lambda(<parameter>,<body>)
       => (GRAPH λarg.( λenv.<body>' )
            (LAMBDA:<parameter>[arg | env])).

(T2) <function>が組込関数のとき,
   <function><list> => <function> <list>' .

(T3) <function>が組込関数でないとき,
   <function><list>
      => (FUN  <function>'  <list>' ).

(T4) <variable> => NAME:<variable> env .

(T5) <constant> => CONST:<constant> .

(T6) (<expression>,... | <expression>)
   =>  [ <expression>' ,... | <expression>' ].

(T7) (<expression>) =>  <expression>' .
```

図5 グラフの生成規則
Fig. 5 Rules for generating graphs.

```
(λenv.PRINT(NAME:Fib env))
 Y[λenv.
   LAMBDA:(Fib,Fib1,Mapcar,Plus)
    [[
      [CONST:1,CONST:1 ¦
       (FUN (NAME:Mapcar env)
            [(NAME:Plus env),
             (FUN (NAME:Fib1 env) (NAME:Fib env))])],

      (GRAPH λarg.
       (λenv.[[CAR(NAME:X env),CAR(CDR(NAME:X env))]
              ¦ (FUN (NAME:Fib1 env) (CDR(NAME:X env)) )])
       (LAMBDA:X [arg ¦ env])),

      (GRAPH λarg.
       (λenv.[ (FUN (NAME:F env) (CAR (NAME:L env)) )
              ¦ (FUN (NAME:Mapcar env)
                     [(NAME:F env) (CDR (NAME:L env))])])
       (LAMBDA:(F,L) [arg ¦ env])),

      (GRAPH λarg.(plus arg)) ]

    ] ].
```

図6 グラフの例
Fig. 6 An example of the graph.

2.3 グラフのリダクション規則

以上述べたような構成のグラフのリダクションを規則の形で述べる. 名前の処理, すなわち変数名や関数名をその値に書き換えるための規則は, NAME : v で v という名前をもつ NAME ノードを表し, LAMBDA : (pl, ···, pn) で (pl, ···, pn) という引数リストをもつ LAMBDA ノードを表すものとすれば, 次のようになる.

(R 1) NAME : pi
(LAMBDA : (pl, ···, pi, ···, pn)
[[*bl*, ···, *bi*, ···, *bn*]|env])
⇒*bi*.

(R 2) v が pl, ···, pn のどれとも一致しないとき,
NAME : v(LAMBDA : (pl, ···, pn)
[*b*|env])
⇒NAME : v env.

これらの規則は, 名前が LAMBDA ノードの変数リストにあれば対応する実引数を取り出し, なければ外側のスコープを参照する.

リストのリダクション規則は, 次のように表現できる.

(R 3) CAR(CONS *a b*)⇒*a*.

(R 4) CDR(CONS *a b*)⇒*b*.

ラムダ式を実引数に適用するには, 次のリダクション規則を使用する.

(R 5) (FUN(GRAPH *f*) *b*)⇒*f b*.

env で外側のスコープの LAMBDA ノードを表せば, ラムダ式は次のように表現される.

(GRAPH λarg.
〈本体のグラフ〉
(LAMBDA : 〈仮引数のリスト〉
[arg|env])).

したがって, (R 5) により, FUN ノードを作用させ, 実引数のリストを結合すれば, 次のようなグラフを得る.

〈本体のグラフ〉(LAMBDA :

〈仮引数のリスト〉
[〈実引数のリスト〉|env]).

このグラフに，規則(R 1)および(R 2)を適用して，仮引数を実引数で書き換えて本体の処理を行う．

以上のようにリダクションが規則(R 1)～(R 5)により記述されるので，以後これらの規則にしたがってリダクションのメカニズムを述べることにする．

3. リダクションのメカニズム

3.1 リダクションの基本操作

関数の定義を含め，すべてのデータはグラフで表現され，分散システムを構成する各プロセッサに分割され，割り当てられる．格子状に結合されたセルラ計算機，バス結合のマルチプロセッサまたはそれらを組み合わせた不規則な結合構造をもつシステムなど，分散処理システムは種々の構造をとる可能性がある．本論文で述べる方法では，リダクションがノード上のプロセスによるそのノード自身の消去あるいはコピーという形で行われる．したがって，分散処理システムの構造がどのようなものであっても，グラフを分割して各プロセッサに割り当てることが可能となる．

また，通信は，アークで接続された相手に限られており，直接相手のノードを指定する代りに，アークを指定して行われる．このように相手ノードを指定するので，システム全体を参照するための仮想アドレスを設ける必要がなく，各プロセッサ内のアドレスおよびネットワーク上のパケット通信のための識別子などおのおののシステム構成要素内でのみアドレスを設ければよい．このことにより，システムの拡張を行ったり，システム同士を結合したりする際のアドレス割当ての煩雑さがなくなる．

ノード上のプロセスによるリダクションの基本操作は，**図7**に示すようなノードの消去とコピーである．リダクションは，ノードのコピーによるグラフの共有部分のコピーとノードの消去によって行われる．ノード上に割り当てられたプロセスによって，そのノードの消去とコピーが指示され，プロセスを管理する各プロセッサのスーパバイザによって，これらの指示が実行される．ノード上のプロセスはノードの消去後不要となるので，ノードの消去と同時にプロセスを停止させる．また，新しいノードを生成するとそのノードにプロセスが必要となるので，ノードのコピーと同時にプロセスのコピーすなわちフォーク (fork)[9] を行う．

ノード間のすべての情報交換は，グラフのアークを

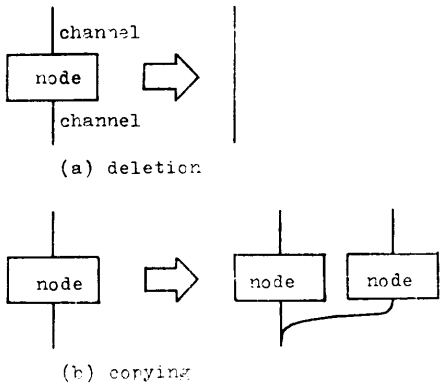


図7 リダクションの基本操作
Fig. 7 Primitive operations of reduction.

$$\begin{aligned} & a\ b \\ =>& a'\{m\rightarrow\}b \\ =>& a'b' \end{aligned}$$

i) ノード a から b にメッセージ m が送られて，それぞれ状態を変え，a' と b' になる．

$$\begin{aligned} & a_0 b_0 \\ =>& a_n\{m_n, m_{n-1}, \cdots, m_2, m_1\rightarrow\}b_0 \\ =>& a_n b_n \end{aligned}$$

ii) ノード a_0 から b_0 にメッセージ $m_1, m_2, \cdots, m_n$ が送られて，それぞれ状態を変え，a_n と b_n になる．

図8 メッセージ交換の表現
Fig. 8 The expression of message passing.

通信路としたプロセス間通信によって行われる．通信の形態は，Hoare の Communicating Sequential Processes[7),8)] と同様である．すなわち，入力命令の標的変数 (target variable) の型が出力命令の式の型と合致 (match) したときに，入力命令と出力命令が同時に実行され，一つのデータが送受信される．

以上により，リダクションの過程は，図4に示したグラフの表現と**図8**に示したノード間のメッセージ交換の表現を用いて，記述されうる．たとえば，リスト操作の過程は次のように記述される．

CAR(CONS *a b*)
⇒CAR′{SEL(CAR)→}(CONS *a b*)
⇒*a*

ところで，グラフの生成規則(T 1)と(T 4)によれば，LAMBDA ノードが一般には複数の NAME ノードによって次のように共有される．

(λenv. …(NAME : s1 env)
…(NAME : s2 env)…)
(LAMBDA : 〈parameter〉…).

このため，名前の処理を行う前にノードをコピーしな

ければならない．各ノードは，COLOR(〈色〉) の形式のカラーと呼ばれるメッセージを受け取ると，自分自身をコピーし，受け取ったカラーを子ノードに送る．生成直後のグラフでは，NAME ノードのみがノードを共有しているので，カラーは最初に NAME ノードで生成される．このメカニズムにより，グラフの共有部分が逐次コピーされる．

⇒(λ env. …(NAME' : s 1{COLOR(NIL)→} env)
…(NAME' : s 2{COLOR(NIL)→} env)…)
(LAMBDA : 〈parameter〉…).
⇒λx. …
(NAME' : s1(LAMBDA : 〈parameter〉
{COLOR(NIL)→} x)…
(NAME' : s2(LAMBDA : 〈parameter〉
{COLOR(NIL)→} x)…
)…

LAMBDA ノードのコピーにより，各 NAME ノードにはそれぞれの LAMBDA ノードが生成され接続されるので，名前の処理，すなわちリダクション規則(R 1)および(R 2)の適用が可能となる．規則(R 1)の操作は次のようになる．

(R 1′) NAME' : pi(LAMBDA : (pl, …, pi, …, pn)
[[bl, …bi, …, bn]|env])
⇒{NAME(pi)→}(LAMBDA : (pl, …, pi, …, pn)
[[bl, …, bi, …, bn]|env])
⇒{SEL(CAR), SEL(CDR)$^{i-1}$, SEL(CAR)→}
[[bl, …, bi, …, bn]|env]
⇒{SEL(CAR), SEL(CDR)$^{i-1}$→}
[bl, …, bi, …, bn]
⇒{SEL(CAR)→}[bi, …, bn]
⇒bi.

ただし，SEL(CDR)$^{i-1}$ は $i-1$ 個の連続した SEL (CDR) を表す．

また，規則(R 2)の操作も同様であるので省略する．

3.2 関数の適用

前述のように，関数の適用は次の規則にしたがって行われる．

(R 5) (FUN(GRAPH f) b)⇒f b.

この規則の左辺のグラフは，後述する引数選択ノードを追加すれば**図 9** (a)のようになり，このグラフを規則にしたがって変形すると図 9 (b)のようになる．FUN ノードと GRAPH ノードのコピー操作と消去操作を組み合わせることにより，この変形操作を行うことができる．

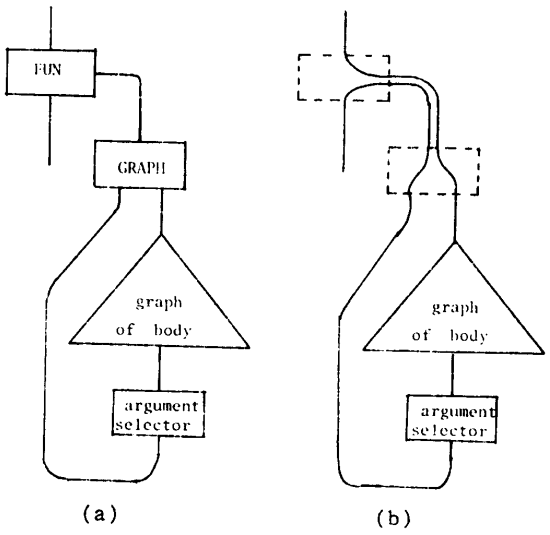


図 9 関数の適用
Fig. 9 Applications of the function.

一般に関数の定義体はプログラム全体によって共有されているので，リダクション方式で関数を呼び出すためには，関数の本体をコピーする必要がある．しかし，コピーを関数の適用時に一度に行うと，そのための空間的，時間的オーバヘッドが大きいうえ，コピーと関数本体のリダクションを並行して行うことができない．これらの問題を解決するため，関数の適用時には関数本体のグラフのコピーを行わず，図 9 に示したようなリンクだけを行い，前述したカラーを用いてグラフをコピーすることにする．この際，関数本体を共有するためのメカニズムとして，引数選択ノードが使用される．

この方法による関数の適用においては，最初に NAME ノードと LAMBDA ノードの作用で，FUN ノードと GRAPH ノードが結合される．その後に，NAME ノードで生成されたカラーが GRAPH ノードに入ることになる．

(λx. …(FUN x b)…(FUN x c)…)
{COLOR(NIL)→}(GRAPH λy. f(argsel y))

ここで，argsel は引数選択ノード (argument selector) である．上の操作の結果，GRAPH ノードがコピーされ，各 GRAPH ノードによりそれぞれの識別子がカラーに乗って送り出される．

⇒(λx. …
(FUN(GRAPH({COLOR(1)→} x))b)…
(FUN(GRAPH({COLOR(2)→} x))c)…)
(λy. f(argsel y)).

図9の操作で実引数を引数選択ノードに結合する.

⇒(λx.
　…({COLOR(1)→}x)
　…({COLOR(2)→}x)…)
　(f(argsel b c)).

関数の本体fのうち，カラーを受け取った部分が逐次コピーされてゆく．引数選択ノードには，この関数へ適用操作を行った FUN ノードの実引数がすべて結合されており，カラーが到着したときにカラーの値によって対応する実引数が選択される.

⇒(λx.…
　(f({COLOR(1)→}x))…
　(f({COLOR(2)→}x))…)
　(argsel b c)
⇒…(f b)…(f c)…

すなわち，argsel のリダクション規則は次のようになる.

(R 6)　{COLOR(k)→}(argsel a1 a2 … ak …)
　⇒ak.

従来のリダクションシステムでは，コンビネータを使用したもの[6),10)]を除き，呼び出された場所に関数の定義がコピーされてから，リダクションが行われていた．これに対し，上述のメカニズムによれば，関数の定義のリダクションは，コピー前，コピー中およびコピー後を通じて中断せずに連続して行われる．また，リダクションに必要な部分のグラフのみに必要な時点でカラーを送るので，グラフのコピー量を最小限に抑えることができる.

リダクションに必要な部分のグラフのみにカラーを流すには，CONS ノードでカラーをただちに引数に送らないようにする．すなわち，カラーを CONS ノードのなかに保持しておき，メッセージ SEL(CAR) または SEL(CDR) が送られてきたときにこれらのメッセージによって選択されたほうの引数にのみカラーを送り出す.

{COLOR(c)→}(CONS a b)
⇒(CONS : COLOR(c) a b)
{SEL(CAR)→}(CONS : COLOR(c) a b)
⇒{COLOR(c)→}a.
{SEL(CDR)→}(CONS : COLOR(c) a b)
⇒{COLOR(c)→}b.

本評価法では，関数の適用が起こりうるところではただちに関数の適用が行われる．しかし，関数の適用時には関数本体のコピーが行われず，適用後にカラーの流れた部分だけがノード単位でコピーされる．上に示したような CONS ノードでのカラーの流し方により，リスト構造のコピーが遅延されることとなり，リスト構造を表す表現が $X \triangleq (1|X)$ のように無限構造を表していても，グラフが際限なくコピーされることはない.

3.3 リダクション結果の参照

単一プロセッサでグラフリダクションを行った場合には，処理結果のグラフを参照することは容易であるが，分散処理システムにおいてはグラフが各プロセッサに分散して存在するため，グラフを参照するためのメカニズムが必要である．このようなメカニズムとしてまず考えられるのは，他のプロセッサのローカルメモリを読み出すことができるような仮想アドレッシング機構である．しかし，このような機構はかなり複雑なものとなるし，プロセッサの結合形態に依存した機構となるために，分散処理システム全体の拡張性を阻害することにもなる．以下，これらの問題を解決するための方法について述べる．すなわち，グラフを構成する各ノードに組み込まれた参照機構によりグラフの参照を行う方法である．リダクションの終了したグラフは CONST ノードと CONS ノードによって構成されるので，グラフの参照機構をこれらのノードに組み込むことにする.

グラフの参照機構は，メッセージ 'EOS( )' を受け取ったときに動作する．CONST ノードの場合には，メッセージ 'EOS( )' を受け取るとアトムが送り返される.

{EOS( )→}CONST : atom
⇒{←atom, EOS( )}CONST : atom
e≠EOS( ) のとき，
{e→}CONST : atom⇒CONST : atom

CONS ノードがメッセージ 'EOS( )' を受け取ると，まず CAR 側のサブグラフの評価結果がそのまま親ノードに送られる.

{EOS( )→}(CONS a b)
⇒{←'('}(OUT−CAR{EOS( )→}a b).
(OUT−CAR{←a 1, …an, EOS( )}a b)
⇒{←a1, …, an}(OUT−CDR{EOS( )→}b).

次に CDR 側のサブグラフを親ノードと接続して評価結果を送る.

(OUT−CDR{←'('}b)⇒b.
(OUT−CDR{←NIL}b)⇒{←')'}b.

CONS ノードがカラーを保持していた場合には，

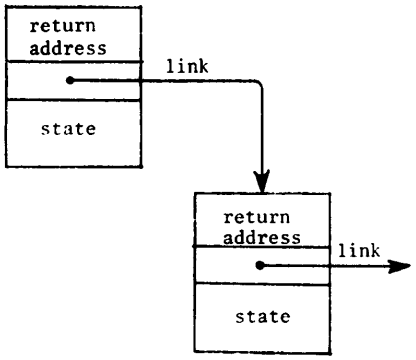


図 10 プロセッサ内でのグラフの表現
Fig. 10 Implementation of graphs in a processor.

まず CAR 側のサブグラフにのみカラーが送り出され，CAR 側のサブグラフの評価結果が親ノードに送り出されてから CDR 側のサブグラフにカラーが送り出される．すなわち，次のようになる．

{EOS( )→}(CONS : COLOR(c) a b)
⇒{←'('}(OUT−CAR : COLOR(c)
{EOS( ), COLOR(c)→}a b).
(OUT−CAR : COLOR(c){←$a1, \cdots, an$,
EOS( )}a b)
⇒{←$a1, \cdots, an$}
(OUT−CDR{EOS( ), COLOR(c)→}b).

このようにして，カラーを流すことにより，グラフが参照されるのに合わせてリストが遅延評価される．

このようにして得られたリダクション結果は PRINT ノードによって出力装置に出力される．すなわち，次に示すように，PR1 が element を受け取るごとにそれが出力装置に表示される．

PRINT⇒PR1{EOS( )→}.
PR1{←element} $a \overset{\text{write(element)}}{\Rightarrow}$ PR1 a.

以上のメカニズムによれば，全体のリダクションの完了を待たずに各部分のリダクション結果をストリームの形で得ることができる．

3.4 分散処理システムでのグラフの実現法

プロセッサが多数結合された分散処理システム上で各プロセッサにまたがったグラフを実現する手法を述べる．グラフはノード上のプロセスがアークの通信路によって結合された構造となっており，任意の部分に分割して各プロセッサに割り当てることができる．このとき，分割されたグラフ間はネットワークを使用した通信によって結合され，ノード上のプロセスの働きで並行してグラフ全体のリダクションが行われる．

各プロセッサ内では，**図 10** に示されるように各ノードはリンク (link) によって結合されている．各ノードにはプロセスの状態 (state) が保持され，リンクと逆向きにリンクをたどるためのリターンアドレス (return address) を用いて，各プロセスが交互に実行される．

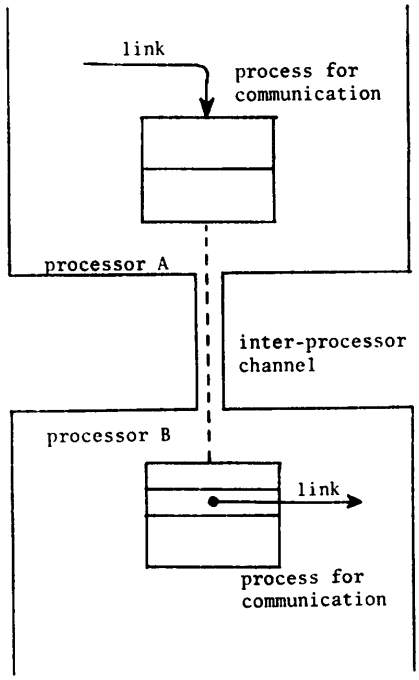


図 11 プロセッサ間にかかるアークの実現
Fig. 11 Implementation of arcs between processors.

プロセッサ間では，**図 11** に示されるように，ネットワークの機能を使用したチャネルによってアークの通信路が実現され，各チャネルごとに通信用のプロセスによって管理される．したがって，グラフがコピーされるときは，新しいチャネルを生成するため通信用のプロセスもコピーされる．

なお，本システムでは，小さなプロセスを多数用いてリダクションを行うため，プロセスのスイッチングのオーバヘッドが問題となる．スケジュリーングに伴うオーバヘッドについては，各プロセスをコルーチンとみなして実行すれば問題ないが，プロセスの状態の退避や復帰に関しては，ハードウェアの支援が必要となる．

4. グラフの自己最適化と部分評価

リダクション方式を用いた評価法では，プログラム自身がより単純なプログラムに変形されてゆくことで評価が行われてゆく．このため，プログラムの変形の

途中結果を保存し，いくつかの処理において共有することにより，評価の効率を上げることができる．Turner のコンビネータによる評価法[6),10)]は，変形の途中結果の共有が自己最適化性（self-optimizing property）により自動的に行われることで，よく知られている．

本論文での評価法では，コンビネータを用いずソーステキストの構造を反映したグラフを使用しているが，グラフのコピーをノード単位で制御するため，自己最適化性をもたせることができる．この際，共有された関数本体の引数の処理が引数選択ノードによって行われる．

グラフの書換え可能な部分をリデックスと呼ぶが，自己最適化の効果をもたせるためには，リデックスを含んだグラフのコピーを避けることが必要である．そのためには，リデックスとなりうるノードがカラーを受け取らないようにすればよい．

カラーを受け取ったノードがコピーされることによって，グラフの共有部分がそれぞれのグラフの中に埋め込まれてゆく．グラフの中に共有部分が埋め込まれると，埋め込んだグラフとの間で再びリダクションが行えるので，さらに最適化を進めることができる．関数の定義もグラフによって表され，各引用先で実引数に適用された後も引数選択ノードの働きにより共有されている．リデックスとなりうるノードがカラーを受け取らないようにすると，共有部分の変形中にはグラフのコピーが行われない．そのため，変形結果を各引用先で共有することができる．

変数名や関数名の処理をプログラムの実行時に行うシステムは，解釈実行系と呼ばれ，一般には効率が悪いとされている．しかし，本論文の評価法では，自己最適化性により変数名や関数名の処理を一度だけで済ませることができるので，実行時に処理を行うことによるオーバヘッドがない．コンビネータを用いたシステムでは，ソーステキストからコンビネータ表現への変換を行うことにより，変数の消去を行っている．これに対し，本評価法では自己最適化性を利用して実行中に変数を消去しているので，変換が不要である．

また，関数に引数の一部だけを与えた場合にも，自己最適化性により，与えられた引数だけで評価が進められる．すなわち，

$F(a, b, c, d)$,

に $a=1$ と $c=3$ だけを与えて，

$F_c(b, d) \triangleq F(1, b, 3, d)$,

のように定義すると，定義した直後から並列リダクションのメカニズムが働いて $a=1$ と $c=3$ だけを用いて評価が進められる．データの一部だけを与えて行われるこのような評価は部分評価（partial evaluation）[11)]と呼ばれ，コンパクトで高速なプログラムを生成するのに利用される．

5. 動的スコープをもつ名前の導入法

本評価法では，名前の評価環境が NAME ノードに接続された LAMBDA ノードのグラフにより決定される．このため，グラフの構成を変更することにより，任意の評価環境を得ることができる．この節では，新たな評価環境を導入する例として，動的スコープをもつ名前の導入法を述べる．

図5に示された規則にしたがって生成されたグラフでは，名前のスコープがプログラムのテキスト上での静的な入れ子構造で決定され，PASCAL や ALGOL などのコンパイラ言語と同様に静的スコープとなる．これに対し動的スコープでは，Lisp のようなインタプリタで実行される言語に見られるように，関数の呼び出された順序にしたがった動的な入れ子構造で名前のスコープが決定される．

前述の生成規則によれば，ラムダ式は次のようなグラフで表される．

GRAPH(λarg.(λenv.⟨body⟩*)
(LAMBDA : ⟨parameter⟩[arg|env])).

⟨body⟩* に現れた名前は，LAMBDA ノードの仮引数リストに存在するか否かが調べられる．存在しなければ，自由変数となるので，上式で env と書かれたグラフを参照して，外側のスコープで束縛されているか否かが調べられる．上式においては，外側のスコープがプログラムテキスト上での静的な入れ子構造で決定される．これが動的スコープで決定されるようにするには，生成規則(T 1), (T 3)および(T 4)を次のように変更して生成されたグラフを用いればよい．

(T 1′) lambda(⟨parameter⟩.⟨body⟩)
⇒(GRAPH λargenv.
(λenv, denv ⟨body⟩*)
(LAMBDA : ⟨parameter⟩
[(CAR argenv)|env])
(LAMBDA : ⟨parameter⟩ argenv)).

(T 3′) ⟨function⟩ ⟨list⟩
⇒(FUN ⟨function⟩*[⟨list⟩*|denv]).

(T 4) ⟨variable⟩ が静的スコープで評価した

い変数のとき,

〈variable〉⇒NAME : 〈variable〉 env.

(T 4′) 〈variable〉が動的スコープで評価したい変数のとき,

〈variable〉⇒NAME : 〈variable〉 denv.

(T 4)によって変換された変数は,従来どおりプログラムテキスト上での静的な入れ子構造にしたがって評価される.(T 4′)で変換された変数は,denv すなわち (T 1′)の (LAMBDA : 〈parameter〉 argenv) を参照して評価される.この場合,自由変数によって参照される LAMBDA ノードは,動的な関数呼出しの入れ子構造にしたがって argenv の cdr 部に接続される.このため,動的スコープによる評価が行われることになる.

ところで,静的スコープをもつ名前は,リダクションにより書き換えられ,消去される.一方,動的スコープをもつ名前の場合には,カラーの到着した後で,引数選択ノードにより評価環境が決定される.この結果,名前が消去される前にグラフがコピーされることになり,刻々と値が変化するような動的スコープの名前については,最適化が自動的に避けられている.

6. むすび

分散処理システムで,グラフリダクション方式により関数型言語を並列評価する方法について述べた.本方法では,グラフのノード上のプロセスを用いてノード単位でリダクションを行う.このため,プロセスに危険領域を設ける必要がなく,システム全体を統一するアドレス空間も不要となるので,大規模で拡張性の高いシステムを構築することができる.

また,本方法によりノード単位でグラフのコピーを行えば,コピー量を少なくすることができ,関数の本体の変形結果が全引用個所によって共有される.この共有化により,関数名や変数名をそのまま含んだグラフを評価しても,名前の処理のオーバヘッドが無視できるようになる.したがって,中間言語を設けるよりソーステキストに近い形のプログラムを直接評価したほうが,デバックのしやすさやスコープのとり方の柔軟性の面から有利になると考えられる.

なお,本研究は文部省科学研究費特定研究(1)課題番号 59118003 によるものである.

参考文献

1) Mago, G. A.: A Network of Microprocessors to Execute Reduction Languages, Two Parts, *Int. J. Comput. Inf. Sci.*, Vol. 8, No. 5, pp. 349-385; Vol. 8, No. 6, pp. 435-471 (1979).
2) Keller, R. M., Lindstrom, G. and Patil, S.: A Loosely-Coupled Applicative Multi-Processing System, AFIPS Conf. Proc., Vol. 48, pp. 613-622 (1979).
3) Keller, R. M., Lindstrom, G. and Patil, S.: An Architecture for a Loosely-Coupled Parallel Processor, UUCS-78-105, Dept. of Computer Science, Univ. of Utah (1978).
4) Darlington, J. and Reeve, M.: ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, Proc. Functional Programming Languages and Computer Architecture, pp. 65-75 (1981).
5) Amamiya, M., Hasegawa, R., Nakamura, O. and Mikami, H.: A List-Processing-Oriented Data Flow Machine Architecture, AFIPS NCC, pp. 143-151 (1982).
6) Turner, D. A.: A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.*, Vol. 9, No. 1, pp. 31-49 (1979).
7) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666-677 (1978).
8) Bernstein, A. J.: Output Guards and Nondeterminism in "Communicating Sequential Processes," *ACM Trans. Prog. Lang. Syst.*, Vol. 2, No. 2, pp. 234-238 (1980).
9) Ritchie, D. M. and Thompson, K.: The UNIX Time-Sharing System, *Comm. ACM*, Vol. 17, No. 7, pp. 365-375 (1974).
10) Hughes, R. J. M.: Super-Combinators—A New Implementation Method for Applicative Languages, Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh, pp. 1-10 (1982).
11) Futamura, Y.: Partial Computation of Programs, Lecture Notes in Computer Science, No. 147, RIMS Symposia on Software Science and Engineering, Kyoto, Springer-Verlag, Berlin, Heidelberg (1982).

(昭和59年3月23日受付)
(昭和59年9月20日採録)