結合子を用いた並列リダクション

堀，有
静岡大学工学部情報工学科

廣川，佐千男
静岡大学工学部情報工学科

関本，彰次
静岡大学工学部情報工学科

出版情報：電子情報通信学会論文誌. D, 情報・システム. J70-D (8), pp.1498-1507, 1987-08
バージョン：
権利関係：

論　文

結合子を用いた並列リダクション

非会員　堀　　有[†]　　非会員　広川佐千男[†]　　正　員　関本　彰次[†]

Parallel Combinator Reduction

Yutaka HORI[†], Sachio HIROKAWA[†], *Nonmembers and* Syouji SEKIMOTO[†], *Member*

あらまし　本論文では，結合子にグラフの書換えだけでなく，プロセッサの起動を制御する機能ももたせた並列制御結合子を提案する．これにより，遅延性が保証され，かつ効率の良い並列結合子リダクションの戦略が可能となることを示す．並列にリダクションを行う方法は，Knuth-Grossの方法がよく知られている．この方法は，リダクションステップ数の短縮には確かに効果があるが，「必須」でないリデックスを実行することによるさまざまな問題が生じる．我々は，必須リデックスの形，および，それがいつ生成されるかを明らかにし，結合子が必須リデックスにリダクションを開始させるようにすることによって，これらの問題も同時に解決する．また与えられたプログラムにストリクト性解析を行い，その情報を利用することによってできるだけ効率のよい並列リダクションを行うようにする．

1. まえがき

関数型言語の処理系の実現に結合子を用いる方法がTurner[6]によって示された．そこにおいても並列処理の可能性は既に指摘されていた．しかし，現れるリデックスをすべてリダクションする単純な方法では，必須でない部分の計算などの無駄が多く，並列化の効果が現れない[5]．また，言語自身にリダクションの制御（いつ並列に処理して良いかという制御）の記述をもたせる方法[1]も可能ではある．

本論文では，言語，ひいてはプログラマーにそのような負荷を負わせずとも無駄もなく効率も良い並列リダクションが可能であることを述べる．

本論文で提示する方式は，まずストリクト性解析[2]の手法を用いてリダクション制御のための情報を抽出し，その情報を結合子にもたせることにより，単なるグラフの1ステップの書換えだけでなく，部分グラフについてのリダクションの開始も結合子に行わせる．これにより，リダクションの戦略（どのリデックスをリダクションするかという戦略）は，もはや大域的には不用となり，すべて結合子による局所的なリダクションのみで計算が行える．

† 静岡大学工学部情報工学科，浜松市
Faculty of Computer Science, Shizuoka University, Hamamatsu-shi, 432 Japan

そこで，我々は「必須リデックス」とは何かを明らかにし，また，実行時に「必須リデックス」となるか否かを翻訳時に解析して，そこで得られた情報を結合子にもたせることによって，これらの問題を解決した．こうすることによって，必要なプロセッサの起動は結合子が行うことになり，プロセッサを強制的に停止させる必要はなくなった．

これにより，遅延性を保証しかつ効率の良い並列リダクションが可能であることを示した．また，VAX 11-780上にFranzlispで作成したシミュレータによる実験結果としても確認した．

1.1 結合子

本論文で用いる結合子は，基本的にはTurner[6]の用いた次の結合子の書換え規則に従う．

$$Ix \longrightarrow x$$
$$Kxy \longrightarrow x$$
$$Sxyz \longrightarrow xz\,(yz)$$
$$Bxyz \longrightarrow x\,(yz)$$
$$Cxyz \longrightarrow (xz)y$$
$$S'kxyz \longrightarrow k\,(xz)\,(yz)$$
$$B'kxyz \longrightarrow kx\,(yz)$$
$$C'kxyz \longrightarrow k\,(xz)y$$

関数型のプログラムを，変数の除去されたこれらの結合子から成る式に抽象化することを翻訳と呼び，翻訳した結果得られる式を結合子式と呼ぶ．

```
(S'(S'S)
   (B'(B'C)
      (B if (B (= 0) I))
      (C'B(B + I) I))
   (C'(C'B)
      (C'B (B f (C (B + I) 1)) (C (B + I) 1))
      (C (B - I) 1)))
```

図1　例1を翻訳して得られた結合子式
Fig. 1　Combinator code of example 1.

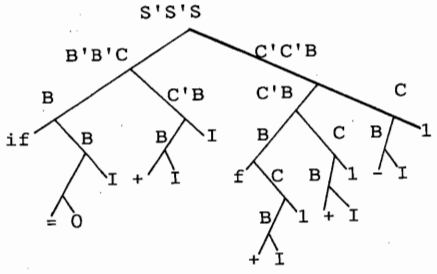

図2　例1の結合子列表現
Fig. 2　Extended code for example 1.

[例1]

$fxyz =$
　if $(=0\ z)$ then $(+x\ y)$
　　else $(f\ (+x\ 1)(+y\ 1)(-z\ 1))$　□

例1のプログラムを翻訳した結果得られる結合子式を図1に示す.

1.2　結合子列表現

関数型のプログラムを結合子式に翻訳すると図1のような式が得られる. 本論文では, 結合子が引数を分配する役割をすることから, これを各節に結合子の列を付けた二分木(結合子列表現[4])を用いて表す. 結合子列表現およびその翻訳アルゴリズムは文献(4)を参照(但し, extensionality はここでは適用しない).

例1の結合子列表現を図2に示す.

1.3　リデックス

リデックスとはリダクションできる状態にある式のことをいい, 次の二つの場合がある.

(1)　結合子のリデックス　　結合子の引数の数がそろった状態にある式

(2)　組込み関数のリデックス　　組込み関数の引数の数がそろい, しかもすべての引数が値になっている状態にある式

2.　リダクションの原則

本論文では, 「あるリデックスのリダクションを実行する」ということを「そのリデックスとなっている部分式をあるプロセッサに渡し, その後にはそのプロセッサは他のプロセッサとは独立にリダクションを行う」ことによって実現する方式を仮定する.

この仮定に基づいた並列リダクションにおいて遅延性を保証することには, ただ単に無駄な計算を行わないだけでなく, プロセッサを強制的に停止させる必要がなくなることも意味する. これによってプロセッサ資源の有効利用と, プロセッサの制御の簡単化が可能となる.

更に, 並列リダクションのリダクションステップ数短縮(並列度向上)のためには, 実際に計算に必要なリデックスは早い時点でリダクションを開始する必要がある.

そこで, 効率の良い並列リダクションは次の原則を満たさなければならない.

[原則1]　遅延性を保証する.

[原則2]　[原則1]の下でできる限り並列に実行する.

3.　単純なリダクション

3.1　最左リダクション

遅延性を保証する最も一般的で単純な戦略は, 最左リデックスを逐次にリダクションをする, 最左リダクションである.

最左リダクションではその計算に必要なリデックスしか実行されないので, 無駄な計算は一切行われない. 次の例2に示すプログラム F を, 最左リダクションにより実行した様子を図3に示す.

[例2]　$Fx = (*(+x\ 2)(+x\ 5))$　□

これは[原則1]は満たされているが, 逐次に実行しているので[原則2]は満たされない.

しかし, 実行時に現れるリデックス(図3では#で示されている)が, 一般には同時に複数個現れ得る. このようにリデックスが複数存在しているとき, それらを並列にリダクションすれば, リダクションのステップ数は減少する.

なお, 議論を簡単にするために, 本論文では各結合子および組込み関数の実行に要する時間はすべて等しいものと仮定する.

3.2　K. G. リダクション

リダクションを並列に行う方法として最も単純な方法は, 現在注目しているグラフに存在するすべてのリデックスについてリダクションを並列に行う Knuth-Gross リダクションである.

例1についての最左リダクションと Knuth-Gross リ

ダクションでの比較結果を表1に示す．なお，表中のtarai，pfac，fibは以下のように定義される関数である．

```
tarai xyz=
      if (>x y) then (tarai (tarai (−x 1)y z)
                            (tarai (−y 1)z x)
                            (tarai (−z 1)x y))
                 else y
pfac  n=fac 1n
fac   lh=
      if (=lh) then l
      else if (=(+l 1)h)   then (* lh)
           else (*(fac 1(/ (+lh)2))
                  (fac (+1(/ (+lh)2))h))
fib n=
  if (<=n 2) then 1
    else (+(fib (−n 1))(fib (−n 2)))
```

この結果からは，Knuth-Grossの方法はリダクションステップ数の減少にかなりの効果が期待できるように見える．しかし，リダクションの総数に注目してみるといずれも最左リデックスのみを実行した場合よりも増加している．最左リダクションは必要な計算しか行っていないのであるから，これは無駄な計算を行っていること，すなわち[原則1]の遅延性が保証されていないことを示している．

Knuth-Grossの方法においてこのようなことが起こる典型的な場合は，関数ifの引数となる部分式のリダクションが，ifの実行が開始される前に開始されるときである(図4参照，これ以後の図では，節に付けられた*はリダクションの開始を示す)．

このような無駄な計算には，無視できない場合がある．図4のggが次のように定義されていたとする．

```
gg xy=if (=0 (−y 1)) then (* x x)
        else (gg (+1 x)(−y 1))
```

図4の状態で(gg 1 0)の実行が行われた場合，ggは自分では停止しないので，ggを実行しているプロセッサは永久に開放されない．

しかも，その中で更に，部分式の実行のために新たにプロセッサを起動し，いくらでもプロセッサの数が増大する(但し，本論文の実験結果ではこれらのプロセッサを理想的に停止できた場合を仮定した)．しかし，例3の実行結果の表2でもわかるように，並列度のわりには実行ステップ数は短くなっていない．同様な並列性の浪費は次のように定義されるプログラムではますますはなはだしくなる．

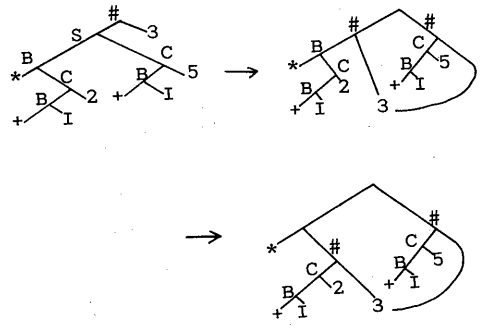

図3　最左リダクションによる例2の実行
Fig. 3　Leftmost reduction for example 2.

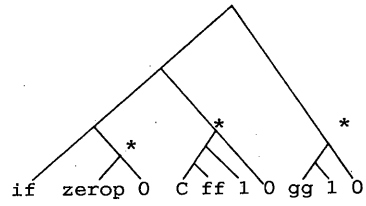

図4　Knuth-Grossリダクションで遅延性が保証されない例
Fig. 4　An example failing laziness by Knuth-Gross reduction.

表1　最左リダクションとKnuth-Grossリダクションの比較

	最左	Knuth-Gross	
実行式	ステップ数	ステップ数	リダクション数
(f 1 1 4)	138	30	196
(tarai 2 1 0)	207	28	481
(pfac 5)	224	30	440
(fib 5)	226	33	466

表2　Knuth-Grossリダクションで並列度の割に実行ステップが減少しない例

	最左	Knuth-Gross	
実行式	ステップ数	ステップ数	リダクション数
(ff 1)	39	22	235
(ss 1)	47	27	1043
(fa 2)	68	26	725

[例3]

```
ff x=if (=(h x) 0) then 0 else (h x)
 h x=if(=x 0) then 0
        else (h(h(−x 1)))
ss x=if (rr x) then 0 else (ss x)
```

rr x = if(< x 1) then true
else (rr(ss(− x 1)))
fa x = if (> x 0) then (fb(fb(− x 1)))
else (fb(fb(+ x 1)))

このように，Knuth-Grossの方法では[原則1]が満たされていないことによる次のような問題が生じる．

・無駄な計算を行う．
これによりプロセッサ等の資源を必要以上に消費する．

・自らは停止しないプロセッサが生じる．
このようなプロセッサは爆発的に増える可能性がある．従って，強制的にそれを停止させる必要があり，プロセッサの制御が煩雑となる．

このため，結合子を用いたリダクションマシンの利点が失われてしまう．そこで，次のように実際の計算に必要なリデックス(必須リデックス)を明らかにすることによって，[原則1]を満たす並列リダクションの方法を考察する．

3.3 必須リデックス

Knuth-Grossの方法で遅延性が保証されなかったのは，最左でないリデックスもリダクションしてしまったことによる．

従って，遅延性を保証するためには，リダクションされるリデックスはすべて「必須リデックス」でなければならない．「必須リデックス」とは，その式から最左リダクションを続けていくと，それがいつかは最左リデックスになるようなリデックスとしてとらえられる．

しかし，あるリデックスが必須かどうかは判定できない．実際に次のような式

if x then (+2 3) else (＊ 5 7)

においては，(+2 3)か(＊ 5 7)のいずれかは必須になるが，そのどちらであるかは決定できない．

そこで本章では，必須リデックスの中で，静的に解析できるものの形を明らかにする．

3.3.1 組込み関数の引数

＋，－等の組込み関数は，引数の値がすべて定まらないと関数の値も定まらないから，それらの引数に現れる最左のリデックスは「必須リデックス」である．従って，このようなリデックスは並列にリダクションを行っても遅延性は保証される(図5参照)．

これは一般に，組込み関数の仮引数に束縛される式が，次のように定義されるストリクトな位置にあるとき，その式に現れる最左リデックスについていえる．

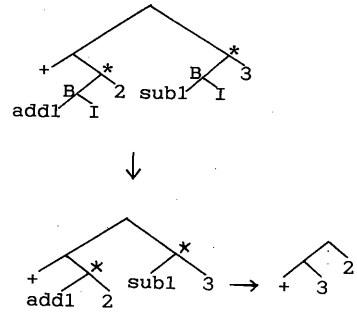

図5 組込み関数の引数に現れる必須リデックス
Fig. 5 Needed redexes in argument positions of primitive function.

[定義1] ストリクトな位置，ストリクトな変数

n 引数関数 (f x_1 x_2 ⋯ x_n)
x_i (i=1, ⋯, n) : 仮引数

において，第 i 引数 x_i = ！のとき，任意の x_j (j=1, ⋯, n ; j≠i) に対して

(f x_1 x_2 ⋯ x_i ⋯ x_n) = ！

となるならば，変数 x_i はストリクトな変数であると言う(ここで，！は値が定まらないことを示す)．

またこのとき，f を引数 M_1, ⋯, M_n に適用した式 (f M_1 M_2 ⋯ M_i ⋯ M_n) において，部分式 M_i は f のストリクトな位置にあるという □

[例4]

(if x y z) ; x, y, z : 仮引数 については，任意の y, z に対しても (if ！ y z) = ！であるから，x はストリクトな変数である．従って，式 L, M, N がそれぞれ x, y, z に束縛されるとき (if L M N) の L は if のストリクトな位置にある．

y = ！であっても x = false ならば (if x ！ z) = z となるので，y はストリクトな変数ではない．z についても同様にストリクトな変数ではない．従って，(if L M N) において，M, N はストリクトな位置にはない．
□

与えられた式に含まれるストリクトな変数を抽出するアルゴリズムは文献(2)等で述べられている．しかし，一般に必須リデックスの抽出と同様，ストリクトな変数をすべて抽出できる訳ではない．

なお，後述する実験結果については，文献(2)のストリクト性解析のアルゴリズムを用いた．

3.3.2 利用者定義関数の引数

組込み関数の引数部分だけでなく，利用者定義関数の引数部分に現れるリデックスの中にも，その値が必須となるものがある．

例えば，次の[例5]のように G が定義されているとき，式 $(G(+23)(*45)(*37))$ の中の $(+23)$ と $(*37)$ は必須となる．

[例5]

$$Gxyz = \text{if}\,(=0\,x)\,\text{then}\,(*\,y\,z)\quad \text{else}\,(*\,x\,z) \qquad \square$$

実際に，G の定義において仮引数 x，z はストリクトな変数であったから，それらに束縛される $(+23)$ と $(*37)$ は先ほどの式においては必須になる．つまり，G が利用者定義関数のとき式 $(f\,M_1 \cdots M_n)$ においてストリクトな位置にある M_i がリデックスであれば，それは必須リデックスである．

3.3.3 左部分木につく結合子

まず，最左の結合子が逐次実行される様子を観察してみる(図6)．図6より結合子は図7のような順序で実行されるのがわかる．

ところで，根も含めて左部分木の節に付く結合子は，○の中の数字の順番に必ず最左のリデックスとなっていく(図6，7参照)．これは，最左リダクションを繰り返すと，これらの結合子はいつか必ず最左リデックスとなる(つまり必須である)ことを意味している．

値を求めるためには関数が実行されなければならないが，関数はその関数名が最左に現れたとき，初めて関数として認識される．

ところがこの関数名は，左部分木に付く結合子が全てリダクションされなければ，最左に現れない．従って，左部分木に付く結合子は必須リデックスを生成することがわかる．

そこで，そのようにして生成されるリデックスは直ちにリダクションが可能である．

以上これらの必須リデックスは，すべて静的に解析できるものであるから，結合子式への翻訳時にこれらを抽出し，その情報に基づいて並列リダクションを行う方法が考えられる．

4. 並列遅延リダクション

3.3 では，静的に検出できる次の三つの場合に必須リデックスができることを明らかにした．

(1) 組込み関数の引数
(2) 利用者定義関数の引数
(3) 左部分木に付く結合子

これらのリデックスは，並列にリダクションしても[原則1] すなわち遅延性は保証される．

本章では，必須リデックスをできるかぎり早い時期にリダクションする戦略を述べる．これにより，[原則1][原則2]が満たされる．

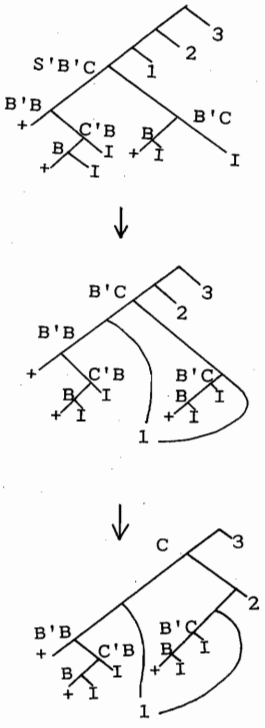

図6 結合子の最左リダクション
Fig. 6 Leftmost reduction for combinators.

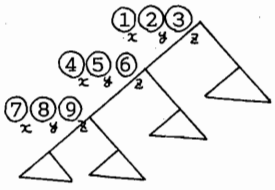

図7 結合子の最左リダクションの実行順序
Fig. 7 Reduction sequence of leftmost reduction for combinators.

4.1 リデックスの生成

結合子がリデックスとなるのは実引数が一つ渡されたときである．このとき，渡された実引数と結合子式を結合するために，必ず新しい節が一つ作られる(図8参照)．

しかし，実引数を分配するのは結合子であるから，この節を作るのも結合子である．結合子は，その実引数を必要とする部分式にしか分配しないので(翻訳時にそのように結合子式が生成されている)このような新しい節はリデックスになっている．つまり，結合子が新しくリデックスを作ると言える(図9参照)．

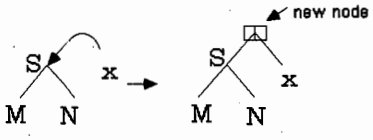

図8　新しい redex の生成
Fig. 8　Creation of a new redex.

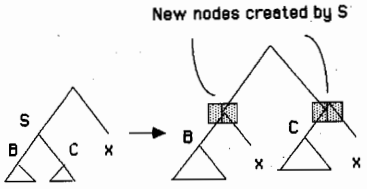

図9　結合子 S による新しい節の生成
Fig. 9　Creation of new nodes by combinator S.

そこで，結合子によって作られたリデックスが必須リデックスであるとあらかじめわかっていれば，そのリダクションを直ちに開始させることが可能となる．

4.2　左部分木に付く結合子のリダクション

左部分木上にできるリデックスは，**3.3** で述べたように必須リデックスである．このリデックスは実引数を左部分木に分配する結合子，S，S'，C，C' によって作られる．従って，これらの結合子に，実引数を分配する（グラフの書換えを行う）だけでなく，それによってできた左部分木のリデックスのリダクションを開始させるようにすれば，いち早く計算が始められる（図 10 参照）．

一般に結合子列表現において，左部分木に付く結合子がリデックスとなるのは，その結合子が結合子列（結合子列表現の節に付けられた複数の結合子の列）の最左に現れていて，かつ実引数の数が揃ったときである（このとき最左リデックスになっているとは限らない）．

ところで，ある結合子が結合子列の最左に現れるのは，その左側の結合子のリダクションがすべて終わったときである．また実引数は，一つ上の節に付けられた同一変数に由来する結合子によって渡される．従って，図 11 のように，同一番号の付いた結合子のリダクションが終わると，新たなリデックスが一つできることになる．

このようにしてできた，左部分木に付いた結合子のリデックスは，そのリデックスを作った結合子によって直ちにリダクションを開始させることができる．

このことは，引数の数に比例して有効な並列性が増すことを意味している．

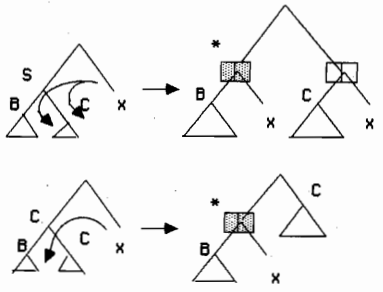

図10　左部分木におけるリダクションの開始
Fig. 10　Activation of reduction in the left leaf.

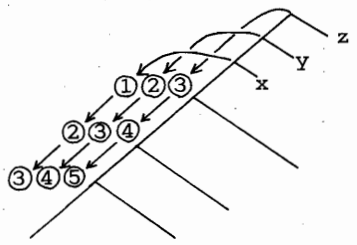

図11　左部分木におけるリデックスの生成
Fig. 11　Creation of redexes in the left leaves.

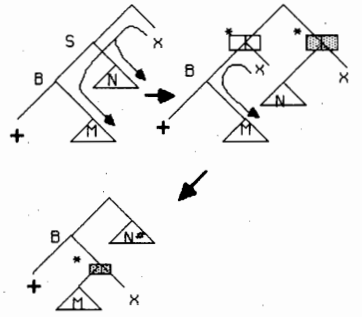

図12　右部分木におけるリダクションの開始
Fig. 12　Activation of reduction in the right leaf.

4.3　組込み関数の引数部分のリダクション

組込み関数では，ストリクトな位置にある最左リデックスは必須であった．また，組込み関数の引数部分は右部分木であるから，引数部分にリデックスを作るのは，右部分木に実引数を分配する結合子，すなわち S，S'，B，B' である．従って，これらの結合子によって実引数が分配される右部分木がストリクトな位置にあれば，**4.2** と同様に，結合子に実引数の分配だけでなく，そこにできるリデックスのリダクションも開始させることができる（図 12 参照）．これはもちろん組込み関数が最左に現れるのを待たずに行って良い（call by value）．但し，それがストリクトな位置になければ実引数を分配するだけでリダクションの開始は始めてはならない

(call by need).

こうすることによって，すべての実引数がそろわなくても組込み関数のリダクションが開始できるので，部分評価に対する制限なしに並列リダクションが可能となる．

4.4 利用者定義関数の引数部分のリダクション

利用者定義関数の場合も，基本的には組込み関数の場合も同じように考えることができる．すなわち，関数 g，f が例6のように定義されていたとすれば，g の中で f が呼び出される際に，その引数部分に実引数を分配する結合子（図13中の(i)）に対して，組込み関数のそれと同様な動作をさせればよい．

[例6]

$g\,xy=(f\,y(+x\,y)\,x)$

$f\,xyz=(+z\,(*\,y\,x))$ □

しかし，そのようにすると，f のストリクトな変数に束縛される実引数は，それがストリクトな位置に分配されるまでリダクションが待たされることになる．このことは，実引数のリダクションに要する時間が，f 本体のリダクションに要する時間に比べて短い場合はさほど問題とはならない．

ところが，実引数のリダクションが，本体のリダクションより時間を要する場合や，利用者定義関数の呼出が行われないような場合には，実引数のリダクションを待たせただけ，確実に全体の実行に要する時間が増す．

ところで，利用者定義関数に与えられた実引数は，図13中の(ii)のように，その本体の根についた結合子によって，最初に各部分式に分配される．そこで，これらの結合子のうち，ストリクトな変数に由来するものに対しては，実引数の分配だけでなく，実引数自身のリダクションも開始させるようにすればよい（図14参照）．

これにより，実引数のリダクションを待たせる必要はなくなり，利用者定義関数の実引数も[原則1]，[原則2]を満たしながら，リダクションすることが可能となる．また，関数本体のリダクションも実引数がそろうのを待つことなく開始できるので，部分評価に対する制限も生じない．

4.5 並列制御結合子

これまでに結合子は，それが作るリデックスが必須リデックスであればそのリダクションも行い，また，組込み関数本体の根に付く結合子がストリクトな変数に由来するものであれば，更にそれが分配する実引数

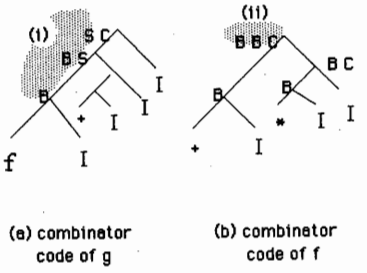

図13 利用者定義関数のストリクト性
Fig. 13 strictness for user-defined function.

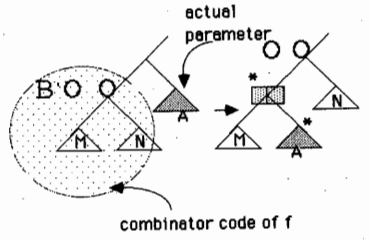

図14 引数に対するのリダクションの開始
Fig. 14 Activation of reduction for argument.

のリダクションを行っても[原則1]は満たされることを見てきた．

そこで，これらの機能をリダクション規則としてもつ並列制御結合子（Parallel Reduction Control Combinator : PRCC）が次のように定義される．

[定義2] 並列制御結合子（PRCC）

$S_0xyz \longrightarrow (xz)^*(yz)$
$S_1xyz \longrightarrow (xz^*)^*(yz^*)$
$S_2xyz \longrightarrow (xz)^*(yz)^*$
$S_3xyz \longrightarrow (xz^*)^*(yz^*)^*$
$B_0xyz \longrightarrow x(yz)$
$B_1xyz \longrightarrow x(yz^*)$
$B_2xyz \longrightarrow x(yz)^*$
$B_3xyz \longrightarrow x(yz^*)^*$
$C_0xyz \longrightarrow (xz)^*y$
$C_1xyz \longrightarrow (xz^*)^*y$
$S_0'kxyz \longrightarrow k(xz)^*(yz)$
$S_1'kxyz \longrightarrow k(xz^*)^*(yz^*)$
$S_2'kxyz \longrightarrow k(xz)^*(yz)^*$
$S_3'kxyz \longrightarrow k(xz^*)^*(yz^*)^*$
$B_0'kxyz \longrightarrow kx(yz)$
$B_1'kxyz \longrightarrow kx(yz^*)$
$B_2'kxyz \longrightarrow kx(yz)^*$
$B_3'kxyz \longrightarrow kx(yz^*)^*$
$C_0'kxyz \longrightarrow k(xz)^*y$

t	t* = (x) t
x	I
M	K M　　x∉M
M N	S_i M* N*　　x∈M, N
	if Nが組込み関数のストリクトな位置 then i = 2 else i = 0
M N	B_i M* N*　　x∉M, x∈N
	if Nが組込み関数のストリクトな位置 then i = 2 else i = 0
M N	C_i　i = 0 M* N*　　x∈M, x∉N
$P_1..P_n$ M N	$S_i'P_1...P_n$ M* N*　　x∈M, N
	if Nが組込み関数のストリクトな位置 then i = 2 else i = 0
$P_1..P_n$ M N	$B_i'P_1...P_n$ M* N*　　x∉M, x∈N
	if Nが組込み関数のストリクトな位置 then i = 2 else i = 0
$P_1..P_n$ M N	$C_i'P_1...P_n$　i = 0 M* N*　　x∈M, x∉N

図15　PRCCへの翻訳アルゴリズム
Fig. 15　Compile algorithm for PRCC.

$C_1'kxyz \longrightarrow k(xz^*)^*y$

（*はリダクションの開始を示す．）□

結合子に付けられた添字0，1，2，3は，それらを二進数で表したとき次のような意味をもつ．

第1ビット

0：右部分木に対して何もしない．

1：右部分木のリダクションを開始させる．

第0ビット

0：実引数に対して何もしない．

1：実引数のリダクションを開始させる．

例えば添字2，すなわち$10_{(2)}$は，右部分木のリダクションは開始させるが，実引数に対しては何もしないことを意味している．

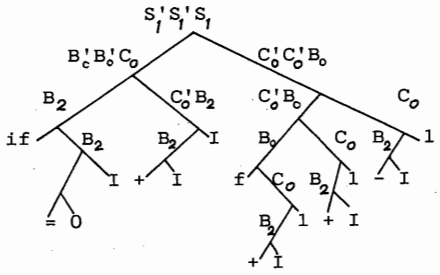

図16　例1の並列制御結合子表現
Fig. 16　PRCC code for example 1.

また，左部分木上にできるリデックスは必須リデックスであるから，左部分木に実引数を分配する結合子に対しては，左部分木のリダクションを無条件に開始させるようになっている．これらの並列制御結合子への翻訳アルゴリズムを図15に示す．但し，(M, N)が根でかつxがストリクトな変数であるときはiの値を図中の各iより1増すものとする．

例1のプログラムを並列制御結合子に翻訳した結果は図16のようになる．

これまで見てきたように並列制御結合子がリダクションを開始させるのは，

（1）　組込み関数の引数部分のリデックス

（2）　利用者定義関数のストリクトな位置に来るリデックス

または，

（3）　左部分木につく結合子により生成されたリデックス

のいずれかである．これらはすべて必須リデックスであるから次のような定理が成り立つことがわかる（詳細な証明は稿を改めて行いたい）．

[定理]　並列制御結合子によるリダクションは，必須リデックスのみをリダクションし，遅延性を保証する．

5.　他の方法との比較

例1他のプログラムを最左，Knuth-Gross，PRCCのそれぞれの方法と比較した結果を表3に示す．参考のため，結合子は最左リダクションで実行し関数のストリクトな実引数のみを並列に実行した結果も表中のstrictnessの欄に示す．

ここで注目しなければならないのはリダクションを行ったリデックスの総数（リダクション数）である．**4.**で示したPRCCによる方法で行ったものと，逐次に最

表3 PRCCによる方法と他との実行結果の比較

実行式		最左	Knuth-Gross	Strictness	PRCC
(f 1 1 4)	s.n.	138	30	89	66
	r.n.	138	196	138	138
(tarai 2 1 0)	s.n.	207	28	110	73
	r.n.	207	481	207	207
(pfac 5)	s.n.	224	30	108	79
	r.n.	224	440	224	224
(fib 5)	s.n.	226	33	61	61
	r.n.	226	466	226	226
(ff 1)	s.n	39	22	29	29
	r.n	39	235	39	39
(ss 1)	s.n	47	27	36	36
	r.n	47	1043	47	47
(fa 2)	s.n	68	26	49	49
	r.n	68	725	68	68

s.n: ステップ数
r.n: リダクション数

表4 PRCCとHankinの方法との比較

例1 (f 1 1 4) の実行結果

	PRCC	Hankin
s.n.	66	143
r.n.	138	204
s.n. / r.n.	47.8% → 43.3% [(f a b n),n → ∞]	70.1% → 65.9% [(f a b n),n → ∞]

s.n.: ステップ数
r.n.: リダクション数

左リダクションを行ったものとは正確に一致している．このことからもPRCCは遅延性が保証されているのが確認される．また実行ステップ数も，Knuth-Grossほどではないが，関数のストリクトな実引数だけを並列に実行したものより更に並列性が向上している．

ストリクト性の情報を含む結合子はHankin[3]でも述べられている．しかし，その方法では引数がすべてそろわなければリダクションが実行されない．一般には引数の個数をkとすると，結合子式の各節において，最大kステップだけ待ち合わせる必要がある．このため，速度向上比(並列処理時間／逐次処理時間)で，我々の方式ほどの値が達成できない．これは，結合子を用いることの利点である，部分評価の機能が失われていることによる．

各方法の詳細は稿を改めて報告したい．本論文では，例1のプログラムについての比較を表4に掲げておく．なお，Hankin[3]らの方法とは生成される結合子式が異なり，直接の比較ができないため，最左リダクションに対する並列リダクションのステップ数の比で比較した．

6. むすび

4.で述べたPRCCによるリダクション戦略は最左となるリデックスのみを実行しているから遅延性は保証されている．またそのようなリデックスはすべて独立に実行されているので，有効な並列性も実現されている．

更に，並列リダクションを制御するための情報は結合子がもっているので，組込み関数は実引数のリダクションの開始を指示(制御)する必要はなく，ただ引数の値が決定するのを待っていればよい．これはハードウェアを実現する場合，並列に動作するプロセッサを制御する機構が簡単になることを示唆している(生成された機械語が，自動的に並列リダクションを制御する形になるものと考えられる)．

また関数は，引数の数がすべてそろわなくてもその評価を開始することができるので，部分評価に対する制限もない．

以上，本稿で提案する並列制御結合子を用いることにより，結合子を用いた逐次的リダクションマシンの利点を何ら失うことなく，並列機械に発展させることが可能となったといえる．

文　献

(1) F. W. Burton : "Annotations to control parallelism and reduction order in the distributed evaluation of functional programs", ACM Trans. Program. Lang. &

Syst., **6**, 2, pp. 159-174 (April 1984).
(2) C. Clack and S. L. Peyton Jones : "Strictness analysis —A practical approach", Springer LNCS 201, pp. 35-49 (1986).
(3) C. L. Hankin, G. L. Burn and S. L. Peyton Jones : "A safe approach to parallel combinator reduction", ESOP86, Springer LNCS 213, pp. 99-110 (1986).
(4) S. Hirokawa : "Complexity of combinator reduction machine", Theoret. Comput. Sci., 41, pp. 289-303 (1985).
(5) 広川佐千男: "Parallel combinator reduction の効率", 理研シンポジウム関数型プログラミング, pp. 1-7 (1986).
(6) D. A. Turner : "A new implementation technique for applicative languages", Softw. Pract. Exper., 9, pp. 31-49 (1979).

(昭和 62 年 1 月 5 日受付, 3 月 26 日再受付)

堀　　有

昭 60 静岡大・工・情報卒. 昭 62 同大大学院修士課程了. 現在県立清水工業高等学校教諭. 関数型言語, リダクションマシン, 並列計算等に興味をもつ. 情報処理学会会員.

広川佐千男

昭 54 九大大学院修士課程了(数学専攻). 同年 4 月より静岡大・工・情報助手. ラムダ計算, 関数型言語処理系, リダクションマシンを研究している. 証明論にも興味をもつ. 情報処理学会, ソフトウェア科学会, LA, 数学会, EATCS 各会員.

関本　彰次

昭 34 京大・理・数学卒, 昭 35 三菱電機入社, 昭 59 静岡大・工・情報教授. 工博. この間, 計算機基本ソフトウェアについての研究・開発, 形式言語とオートマトン, 計算理論に関する研究に従事. 情報処理学会会員.