ソフトウェア基礎論　18−2
（１９８６・９・３０）

結合子による並列リダクション

堀　有，　広川　佐千男
静岡大学工学部情報工学科

本稿では、遅延評価でかつ効率のよい（eager）な並列結合子リダクションを実現する戦略を与える。我々はこれを結合子にプロセッサの起動を制御する機能を持たせることにより実現した。

並列にリダクションを行う方法は Knuth-Grossの方法が良く知られている。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様々な問題が生じる。我々は、”必須リデックス”の形を明らかにするとともに、これらの問題点も同時に解決した。

LAZY AND EAGER PARALLEL COMBINATOR REDUCTION

(in Japnese)

Yutaka HORI and Sachio HIROKAWA
Department of Computer Science, Shizuoka University
Hamamatsu 432, JAPAN

This report presents a "lazy and eager" strategy for parallel combinator reduction. In this strategy each combinator controls not only graph reductions but also the activation of processors.

The simplest strategy for parallel combinator reduction is the "Knoth-Gross" strategy. This strategy decreases the number of reduction steps, but it contains unnecessary reduction which causes some inefficiency and difficulties. To get rid of these problems, we apply the strictness analysis for given functional program and we make clear when a necessary redex is produced. The reduction of such a redex is performed immediately after receiving activation signal from the combinator that produces the redex.

1．はじめに

関数型言語の処理系の実現に結合子を用いる方法が［7］で示された。そこにおいても並列処理の可能性はすでに指適されていた。しかし、現れるリデックスを全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］。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）の記述を持たせる方法［1］も可能ではある。

しかし本稿では、言語ひいてはプログラマーにその様な負荷を負わせずとも無駄もなく効率の良い並列リダクションの一方式を述べる。それはストリクト性解析［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持たせることにより、単なるグラフの1ステップの書き換えだけでなく、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方法である。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）は最早大域的には不用となり、全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。同様な方法は［3］でも述べられているが、そこでは引数がそろうまでリダクションは待たされ、結合子を用いるメリットが失われている。

並列 リダクションの戦略で最も単純なものは、Knuth-Grossの方法である。

我々はリダクションを次のように実現するものと仮定する。

一つのプロセッサにリデックスを渡し、そのプロセッサは独立にリダクションを行なう。この仮定のもとで、Knuth-Grossの方法はリダクション ステップ数の減少に関してはかなりの効果が期待できる。

しかし、この方法では”必須”でないリデックスをリダクションするために次のような問題点が生じる。

・無駄な計算をおこなう。
　－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。

そこで我々は”必須リデックス”とは何かを明らかにし、これらの問題を解決した。また、実行時に”必須リデックス”となるか否かは翻訳時に知ることができ、そこで得られた情報を結合子に持たせるようにした。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。

この様にした結果、遅延でかつeagerな並列結合子リダクションが可能になることを示した。

1.1 結合子

Turnerは基本的な結合子として次のものを用いている［7］。

```
S＝λｘｙｚ．ｘｚ(ｙｚ)
K＝λｘｙ．ｘ
Ｉ＝λｘ．ｘ
Ｂ＝λｘｙｚ．ｘ(ｙｚ)
Ｃ＝λｘｙｚ．ｘｚｙ
```

```
S'＝λａｘｙｚ．ａ(ｘｚ)(ｙｚ)
Ｂ'＝λａｘｙｚ．ａｘ(ｙｚ)
Ｃ'＝λａｘｙｚ．ａ(ｘｚ)ｙ
```

関数型のプログラムを変数の除去された、これらの結合子を含む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。

例1

```
f x y z =
      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 結合子式は（図1）のようになる。

```
(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)))
```

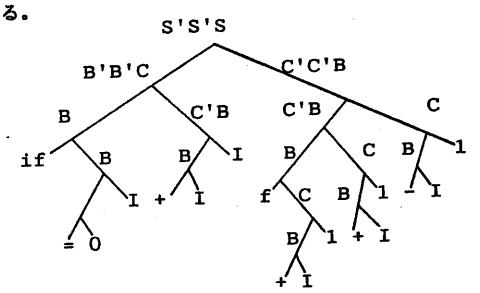
（図1）例1を翻訳して得られた結合子式

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。本稿ではこれを各節に結合子の列が付いた二分木（結合子列表現［4］）を用いて表す。

結合子列表現及びその翻訳アルゴリズムは［4］を参照して頂きたい。

例1を結合子列表現を用いて表すと（図2）のようになる。



（図2）例1を結合子列表現を用いて翻訳した結果

2．リダクション戦略における原則

本稿では”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ッセサとは独立にリダクションを行なう”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なわないだけでなく、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や、プロセッサの制御を簡単にすることが可能となる。

さらに、並列リダクションのリダクション ステップ数を短くする（並列度を上げる）ためには、あるリデックスが実際に必要な リデックスならばできるだけ早い時点でリダクションを行う必要がある（eagerness）。

そこで、次の２つの原則を満たす並列リダクションの戦略を考えていく。

（原則１）遅延性を保証する。
（原則２）遅延性が保証されている限りにおいて eagerである。

３．並列リダクションの戦略

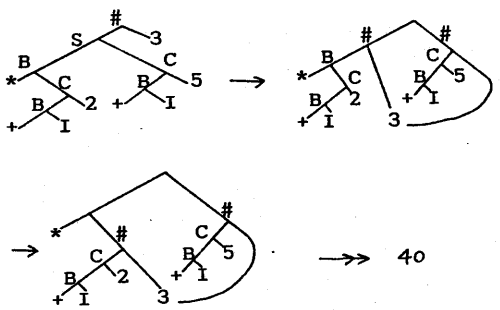
3.1 逐次最左リダクション

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

最左リダクションはその計算に本当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に示す簡単なプログラム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　F(x) = (* (+ x 2) (+ x 5))



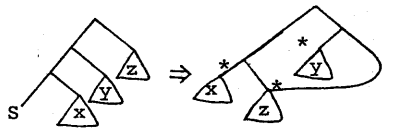
（図３）最左リダクションによる例２のプログラムの実行

（図３）の中には、実行途中に現れるリデックスを＃で示してある。これらのリデックスが複数存在している時、それらを同時にリダクションすれば、当然リダクションのステップ数は減少するはずである。

尚、議論を簡単にするために、本稿では各結合子及び組込み関数の実行に要する時間は全て等しいものと仮定する。

3.2 Knuth-Gross並列リダクション

リダクションを並列に行う方法として最も単純な考え方は、現在注目しているグラフに含まれるすべてのリデックスについてリダクションを同時に行うと言うものである（図４参照）。これはKnuth-Grossの方法として一般に知られている。



（＊のグラフ中のリデックスは全てリダクションする）
（図４）Knuth-Grossの方法の概念図

例１を最左リダクションとKnuth-Grossリダクションで実行した結果を比較すると次表のようになる。

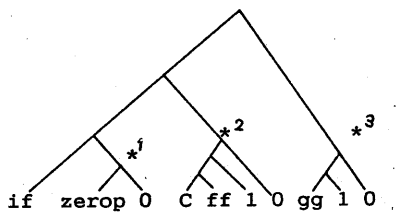
例１．（f 1 1 4）の実行結果

最左	Knuth-Gross	
ステップ数	ステップ数	リダクション数
１３８	３０	１９８

（表１）最左リダクションとKnuth-Grossリダクションの比較

この結果からリダクションのステップ数はかなり減少しているのが分かる。従って実行速度は確かに向上したと言える。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本当に必要な計算しか行っていないのであるから、これは明らかに無駄な計算を行っていること、即ち［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すでに始っているような時である。（図５参照）



（＊のついた部分式はリデックスなので同時に計算が行われる。ところが３の部分式はifの実行後必要でなくなる。）
（図５）knuth-Grossの方法で遅延性が保証されない典型例

このような無駄な計算は、無視できない場合が存在する。上の例でggが次のように定義されていたとしよう。

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

上の例のように(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、幾らでもプロセッサの数が増大する。（但し、本稿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、（表2）でも分かるように、並列度が上がるわりには実行ステップ数は短くならない。

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)))
```

例3　(ff 1)の実行結果

最左	Knuth-Gross	
ステップ数	ステップ数	リダクション数
39	22	235

（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

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様々な問題が生じる。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［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 並列遅延リダクション (non-eager version)

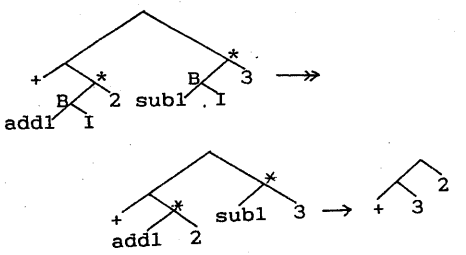
Knuth-Grossの方法で遅延性が保証されなかったのは、最左でないリデックスもリダクションしてしまったことによる。

従って遅延性を保証するにはリダクションを行うリデックスが”必須なリデックス”でなければならない。”必須なリデックス”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。

本章ではそのようなリデックスの形を明らかにする。

3.3.1 引数の並列実行

＋，＊等の組込み関数は引数の値が全て定まらないと関数の値も定まらない（引数はストリクトである）から、それらの引数に現れる最左のリデックスは”必須”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6参照）。



（＊のreduxは同時にリダクションしてよい）

（図6）組込み関数の引数の並列実行

Knuth-Grossの方法で遅延性が保証されなかったのは、最左リデックスにならない 部分式に引数が渡されてしまったことによる。引数の分配は結合子が行っているわけであるから、結合子のリダクションは最左リデックスに対してのみ行えば遅延性は保証される。

これらのことから、組込み関数の引数を並列に リダクションし、結合子のリダクションは最左リデックスになっているものだけについて行えば、遅延性を保証できることになる。

この章では、組込み関数が、その引数部分に現れる最左リデックスを他のプロセッサに割り当てることによって、それを並列にリダクションするものとする。この場合、並列リダクションの制御は組込み関数が行うことになる。

この方法による例1の実行結果を（表3）に示す。

例1　(f 1 1 4) の実行結果

最左	組込み関数の引数を並列実行	
ステップ数	ステップ数	リダクション数
138	121	138

（表3）組込み関数の引数を並列に実行した結果

（表3）の結果を見ると遅延性は保証されているが、組込み 関数の引数を並列にリダクションしても、最左リダクションを行った場合と比較してさほど実行ステップは短くなっていない。

そこで、さらに並列実行の可能性を考察する。

組込み関数の引数だけでなく、利用者定義の関数の引数部分に現れるリデックスの中にも必ず最左になるものが存在する。それは、利用者定義関数の定義に用いられている変数の中で、実行時に必ず評価が必要となる変数に対応しているものである。

例えば次のように定義された関数Fでは、変数x，zは実行時に必ずその評価値が必要となる。

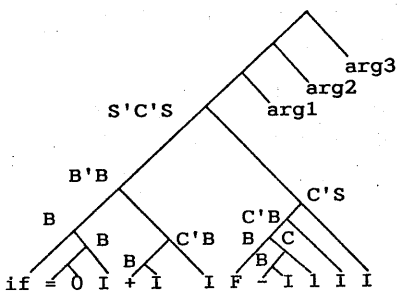
例4

```
F x y z =
    if (= 0 x) then (+ y z)
      else (F (- x 1) z z)
```

このような変数は以後ストリクトな変数と呼ぶことにする。ストリクトな変数はプログラムの ソースコードにストリクト性解析［2］を施すことによって発見できる。

ストリクトな変数に束縛される式は、＋や＊の引数と同様に同時に評価することができる（上の関数Fは第一引数と第三引数が同時に評価可能）。ところが、翻訳して生成された結合子式は全て変数が除去されているので、このままでは同時にリダクションしてよいリデックスを知る手立てがない。

ここで、上の関数Fの結合子式に引数を渡した状態を見てみると、（図7）のようになっている。



（図７）例４の結合子式に引数が渡された様子

ここに現れている引数 arg1,arg2,arg3 は、結合子式の根についた結合子によって各部分式に分配されていく。従って、この根につく結合子のうち、ストリクトな変数に由来するものには、引数の分配と同時に引数自身のリダクションを始めるように指示する情報を、何等かの形で持たせておけばよい。

そこで、我々は次のようなリダクション規則を持つ結合子を新しく定義した。

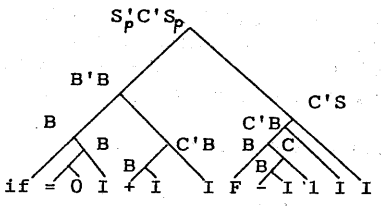
定義１

```
Sp x y z      ⇒   x z*(y z*)
Bp x y z      ⇒   x (y z*)
Cp x y z      ⇒   x z* y
Sp'k x y z    ⇒   k x z*(y z*)
Bp'k x y z    ⇒   k x (y z*)
Cp'k x y z    ⇒   k x z* y
```

（＊はこの部分グラフのリダクションが既に始まっていることを示す。）」

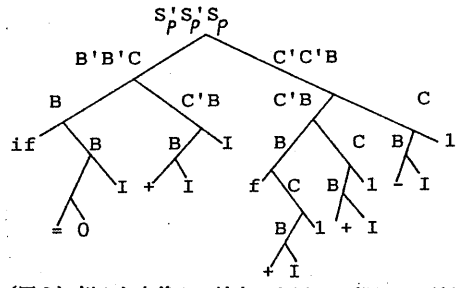
例えば定義１の結合子 Sp はグラフの書き換えの他に他の部分グラフ（この場合　z）を別のプロセッサに渡し、そのプロセッサに起動をかけるというプロセッサ制御の機能を持つ。

先の例４ではx,zがストリクトな変数であるから、定義１の結合子を用いるとその結合子式は（図８）のようになる。



（図８）例４を定義１の結合子を用いて翻訳した結果

また例１のプログラムではストリクトな変数はx,y,zであるから、その結合子式は（図９）のようになる。



（図９）例１を定義１の結合子を用いて翻訳した結果

これらの式（図８，９）を3.3.1節の方法（組込み関数の引数を並列に実行する）も合わせて取り入れて実行した結果は（表４）のようになる。

	最左	ストリクトな引数を並列実行	
	ステップ数	ステップ数	リダクション数
a	138	89	138
b	107	86	107

a：例１（f 1 1 4）の実行結果
b：例４（F 4 1 1）の実行結果

（表４）ストリクトな変数と組込み関数の引数を並列に実行した結果

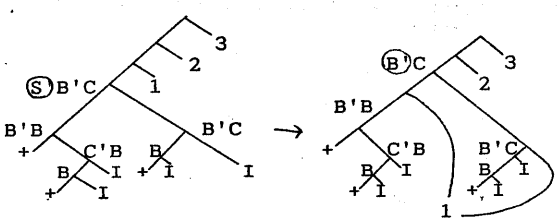
これを見るとやはり遅延性が保証されているのは分かるが、まだステップ数はそれほど減少していない。

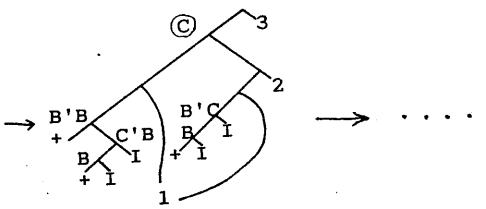
3.4 遅延でかつ Eager な並列リダクション

3.3節の方法では、最左リダクションと比べても、そのリダクションのステップ数は僅かしか向上していない。これは、実行ステップの殆んどが結合子のリダクションに占められていて、（しかもこれは逐次に実行されている。）関数の引数を並列に実行している部分は全実行部分の僅かな割合にしかならないからだと考えられる。

そこで、結合子のリダクションを［原則１］，［原則２］を満たすように並列実行することを次に考える。

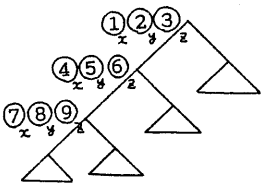
まず、最左の結合子が逐次実行される様子を観察してみる（図１０）。





（図１０）結合子の最左リダクション

（図１０）より、結合子は次のような順序で実行されるのがわかる（図１１）。



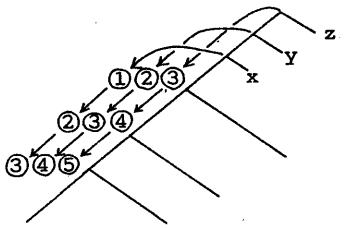
○は結合子を表し、その中の数字は結合子の実行順序を示し、右下の文字は結合子の由来する変数を示す。

（図１１）結合子の最左リダクションの実行順序

ところで、根も含めて左部分木の節につく結合子は、○の中の数字の順番に必ず最左のリデックスとなっている（図１０，１１）。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。従って、この様な結合子はそれがリデックスとなり次第（最左でなくとも）リダクションして良いことになる。

それでは、これらの結合子がリデックスとなるのはいつであろうか。ある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、かつ引数の数が揃った時である（最左リデックスとは限らない）。

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全て終わった時である。また、引数は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１２）の様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。

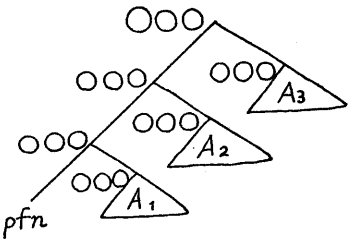


（図１２）左部分木の節に付く結合子がリデックスとなる様子

この様にしてできた、左部分木に付いた結合子のリデックスは、最左になるのを待つことなく直ちにリダクションできる。

次いで結合子列表現の右の分枝について考察する。

結合子列表現に＋，＊等の、引数を並列に実行できる組込み関数が現れている場合を考えてみる。



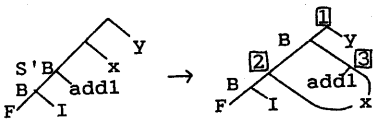
（図１３）組込み関数に引数を渡す結合子

（図１３）で組込み関数 pfn の引数が全てpfnの値を求めるのに必要であれば、部分木 A1,A2,A3 の根についた結合子は、最左リダクションを行った時必ず最左リデックスとなる。従ってそれらの結合子のリダクションは、それがリデックスとなり次第、pfnが最左に現れるのを待たずに実行してよい。つまり、A1,A2,A3 の部分木に引数を渡す各結合子が、それらの部分木に引数を渡すと、直ちにその部分木のリダクションを開始させることができる。これにより、pfnはその引数のリダクション開始を制御する事から開放される。また、部分評価が可能となるので、pfnの全ての変数がそろわなくてもリダクションが始められる。

利用者定義の関数についても同様な事が言える。

以上のように、結合子は引数の受け渡しだけでなく、引数を渡してできる新たなリデックスに対し、もしその リデックスが必須（ストリクトな位置にある）ならばそのリダクションを開始させるという制御までさせる事が可能となる。実際に必須かどうかの解析は翻訳時にできる。

例えばSのリダクションを行った後は（図１４）のようになり、



（図１４）結合子Sのリダクション

リデックスとなる所が1,2,3の３ヵ所ある。他に引数 x がストリクトならば、その部分についてのリダクションも開始できる。これらのリデックスのリダクションを制御するために次のように各結合子を定義する。

定義２　並列制御結合子　(PRCC)

S0 x y z　→　(x z)*(y z)
S1 x y z　→　(x z*)*(y z*)
S2 x y z　→　(x z)*(y z)*
S3 x y z　→　(x z*)*(y z*)*
B1 x y z　→　x (y z*)
B2 x y z　→　x (y z)*

```
B3 x y z      →    x (y z*)*
C0 x y z      →    (x z)* y
C3 x y z      →    (x z*)* y

S0'k x y z    →    k (x z)* (y z)
S1'k x y z    →    k (x z*)* (y z*)
S2'k x y z    →    k (x z)* (y z)*
S3'k x y z    →    k (x z*)* (y z*)*
B1'k x y z    →    k x (y z*)
B2'k x y z    →    k x (y z)*
B3'k x y z    →    k x (y z*)*
C0'k x y z    →    k (x z)* y
C3'k x y z    →    k (x z*)* y
```

(* はリダクションの開始を示す。) 」

これらの結合子から成る 結合子式を生成するアルゴリズムの概略は次の通りである（図１５）。

t	t*
x	I
M	K M　　x∉M
M N	S i M* N*　　x∈M, N
	if x が ストリクトな変数 then if N が ストリクトな位置 then i = 3 else i = 1 else if N が 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 x が ストリクトな変数 then if N が ストリクトな位置 then i = 3 else i = 1 else if N が ストリクトな位置 then i = 2
M N	C i M* N*　　x∈M, x∉N
	if x が ストリクトな変数 then i = 1 else i = 0
M N	Si'Pi..Pn M* N*　　x∈M, N
	if x が ストリクトな変数 then if N が ストリクトな位置 then i = 3 else i = 1 else if N が ストリクトな位置 then i = 2 else i = 0
M N	Bi'Pi..Pn M* N*　　x∉M, x∈N
	if x が ストリクトな変数 then if N が ストリクトな位置 then i = 3 else i = 1 else if N が ストリクトな位置 then i = 2
M N	Ci'Pi..Pn M* N*　　x∈M, x∉N
	if x が ストリクトな変数 then i = 1 else i = 0

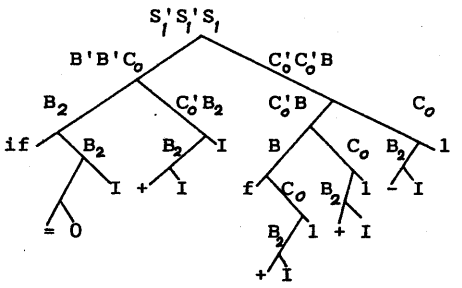
但し、

(x i,..., x n-1, x n) t = (x i,... x n-1)((x n) t)

M N において、Mの最左葉が+,－等の組込み関数で、かつNがそのストリクトな引数の位置にあるときNをストリクトな位置にあると言う。

(図１５) PRCCへの翻訳アルゴリズム

これらの結合子を用いると例１のプログラムは次のような結合子式に翻訳される（図１６）。



(図１６) 例１をPRCCへ翻訳した結果

これを実行した結果を最左，Knuth-Gross及び3.3節の方法で実行したものとあわせて下に示す（表５）。

例1 (f 1 1 4) の実行結果

	最左	Knuth-Gross	Strictness	PRCC
s.n.	138	30	89	66
r.n.	138	196	138	138

s.n: ステップ数
r.n: リダクション数

(表5) PRCCによる方法と他の方法との実行結果の比較

ここで注目してもらいたいのはリダクションを行ったリデックスのtotalである。本節で示した PRCCによる方法で行ったものと逐次に最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PRCCは遅延性を保証しているのが分かる。また実行ステップ数も3.3節の方法よりずっと改善されている。

4. 結論

3.4節で述べた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のであるから明らかに遅延性は保証されている。またそのようなリデックスは全て独立に実行されているので、eagernessも実現されている。

さらに、並列リダクションを制御するための情報は結合子が持っているので、組込み関数は引数のリダクションを指示(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。しかもこれらの情報は、全て翻訳時に解析できる。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する。 (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、引数の数が全て揃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。

これらのことは、これまでの結合子によるリダクションマシンを構成する場合のメリットを失うことなく、それをPRCCを用いる事により並列な機械に発展させる事が可能になったと言える。

5. 謝辞

日頃御指導頂く関本彰次教授に感謝致します。

尚、この研究の一部は、昭和61年度科学研究費補助金奨励研究 (A) 61750326 の援助を受けた。

6. 参考文献

(1) F.W.Burton: Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs, ACM vol.6, no.2, 159-174, April 1984.

(2) C.Clack and S.L.Peyton Jonse: Strictness analysis - a practical approach, Springer LNCS 201, 35-49, 1986.

(3) C.L.Hankin and G.L.Burn and S.L.Peyton Jonse: A Safe Approach to Parallel Combinator Reduction, ESOP86 Springer LNCS 213, 99-110, 1986.

(4) S.Hirokawa: Complexity of Combinator Reduction Machine, Theoretical Computer Science, vol.41, 289-303, 1985.

(5) 広川: Parallel Combinator Reduction の効率, 理研シンポジウム 関数型プログラミング, 1-7, 1986.

(6) T.Johnsson: Lambda Lifting Transforming Programs to Recursive Equations, Functional Programming Languages and Computer Architecture Springer LNCS 201, 190-203, 1985.

(7) D.A.Turner: A New Implemantation Technique for Applicative Languages, Software Practice and Experience, vol.9, 31-49, 1979.