記号処理45－3
（1988. 3. 23）

コンビネータリダクションチップ その方式とアーキテクチャ

和田　英一　　児玉　祐悦
東京大学工学部

グラフコピー方式によるSKコンビネータリダクションを行う専用システムについて考察する。本システムは後置型コンビネータを取り入れることにより、完全遅延評価並びにストリクトコンビネータを効率よく実行できるように、グラフコピー方式を改良している。また、リダクション実行部をチップ化し、内部複数バスによるレジスタ並列転送や、専用キャッシュ化されたレジスタによる高速スタック処理、後置型コンビネータ検出のためのフラグ、タグやコンビネータによる多方向分岐命令、スタック及びヒープポインタのオーバーフローの割り込みによる検出など本リダクション方式に適した特別の機能を備えている。このため高速なリダクションを可能にしている。

Chip for combinator reduction --- its scheme and architecture

Eiiti Wada and Yūetsu Kodama

Department of Mathematical Engineering
University of Tokyo
7-3-1, Bunkyou-ku, Tokyo, 153

A system for SK combinator reduction with graph copying mechanism is described. This graph copying scheme is improved by adopting postfixed combinators, and effectively executes fully lazy or strict combinator evaluation. Reduction executer is implemented on the customized CPU chip. The chip has special functions suitable for the system i.e., parallel register translation by inner buses, fast stack operations by special caching registers, flags for detecting postfixed combinators, multi-branch instructions according to tag contents or combinators, stack and/or heap point overflow detection by interrupt, etc. Reduction speed was estimated by simulater.

１．　はじめに

１９７９年にターナー[1]によってコンビネータリダクションによる関数型言語の実行系が提唱された。その方式の利点として、式から変数が取り除かれているため、変数と伴って必要となるスコープなどの処理が不用になり実行系が簡単になること、プログラム検証などで必要とされている遅延評価が自然に実現できること、その遅延評価を単純に行うと引数が何度も評価され効率が落ちてしまうがそれを防ぐ完全遅延評価が簡単に実現できること、評価順による結果の安全性が保証されていることなどがあげられる。

このような方式が本当に関数型言語の実行系として適しているかどうかを検証するためには、ある程度大きなプログラムを実行する必要がある。これを既存のコンピュータ上で実行しようとするとどうしてもインタープリティブに実行する必要があるため、スピード的に遅くなってしまう。しかし、これではコンビネータリダクション方式に対する評価というよりは既存のコンピュータとこの方式の親和性に対する評価となっている。

そこで、コンビネータリダクション方式に適した専用システムを開発し、その上で評価することが必要である。本論文ではこのような専用システムを考えた場合のコンビネータリダクション方式とそのためのリダクションチップのアーキテクチャについて述べてみる。

２．　コンビネータリダクション方式

まず、コンビネータリダクションの方式について述べる。今回のシステムのコンビネータとしては、ＳＫＩコンビネータを用いる。Ｓ，Ｋ，Ｉなどの各コンビネータの処理は非常に簡単であり、チップ内での実行に適している。主なコンビネータのリダクションルールを表１に示す。

また、リダクション方式としてはグラフコピー方式[2]を用いる。これは、グラフ書換え方式よりもヒープメモリへのアクセスが少なく、しかも、特殊なセルを共有することにより完全遅延評価を行うこともできる。

完全遅延評価のためコンビネータＳのリダクションを行うときに図１（b）のようにコンビネータevalとｚからなるセルをつくり、それをｚであるかのようにスタック操作を行っておく。そして、eval ｚを評価するときには図１

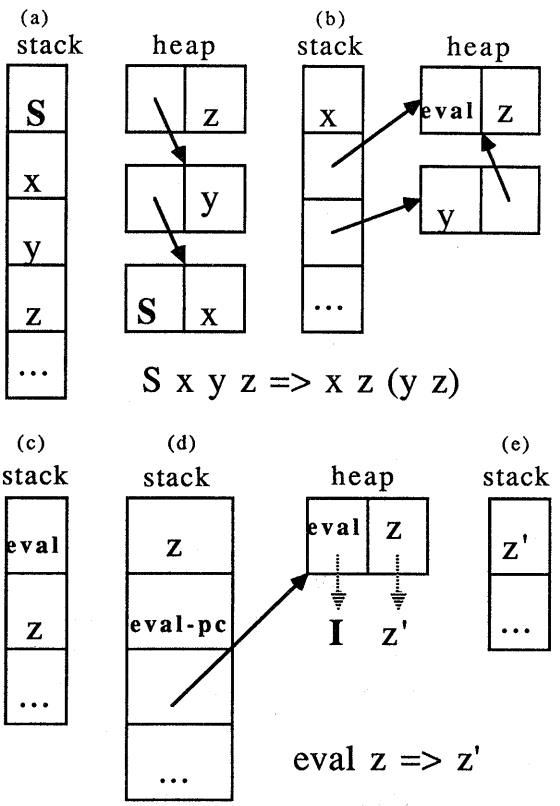


図　１　完全遅延評価

(d) のように後置型コンビネータ[3] eval-pc を用いて z を評価し セルを図 1 (e) のようにその結果で書き換えることにより、次にこのセルを評価するときにはこの結果を取り出すだけでよくなり、z の評価を高々 1 度しか実行しないようにすることができる。

また、plusのようなストリクトコンビネータの評価にも後置型コンビネータを用いることにより、引数の評価を少ないオーバーヘッドで行うことができる。コンビネータ plus の場合、まず図 2(b) のように後置型コンビネータ eval-pc を用いて x を評価し、次に (c) のように後置型コンビネータ eval-pc2 を用いて y を評価し、それから x'+y'を計算する。

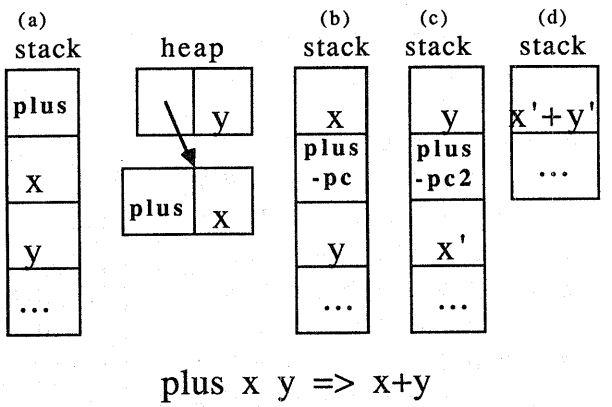


図 2 ストリクトコンビネータ

IF コンビネータはないが、条件をチェックするコンビネータ CONSP, EQ, LT はリダクションの結果、コンビネータK, Jとなる。

```
K  then-part else-part => then-part
J  then-part else-part => else-part
```

であるから、条件チェックコンビネータはそのまま IF コンビネータの代わりになる。

表 1 各コンビネータのリダクションルール (一部)

```
S  x y z => x z' y z'
        ここで z' = EVAL z
  K  x y => x
    I  x => x
B  x y z => x (y z)
C  x y z => x z y
  J  x y => y
    Y  f => f (Y f)

P  x y => P x y
        リスト構造を示す
     HD s => s HD-PC
a HD-PC => x
        ここで a = P x y
   CONSP s => s CONSP-PC
a CONSP-PC => K (if true)
              J (if false)
```

表 2 制御命令

```
INVALID =>              終了
   OUT x => x OUT-PC
a OUT-PC =>             a を出力
  INIT x => INVALID     そしてスタックポインタクリア
   GC-ON =>             チップ上のデータをすべてスタックメモリ上に
                        吐き出したあとストップ
  GC-OFF =>             スタックメモリに吐き出したデータをチップ上
                        に取り戻し、リダクションを継続
```

３．　コンビネータリダクションチップ

この方式に基づくリダクションチップを説明する。

このチップのブロック図は図３のように、マイクロコード制御を行うコントロールユニットと、レジスタなどからなるデータユニット、それにスタック操作を行うスタックユニットの３つのユニットからなっている。ヒープメモリ用のバスとスタックメモリ用のバスは分離されていて、それぞれのメモリに同時にアクセスすることができる。また、各メモリはデータ３２ビット、アドレス１５ビットとなっている。

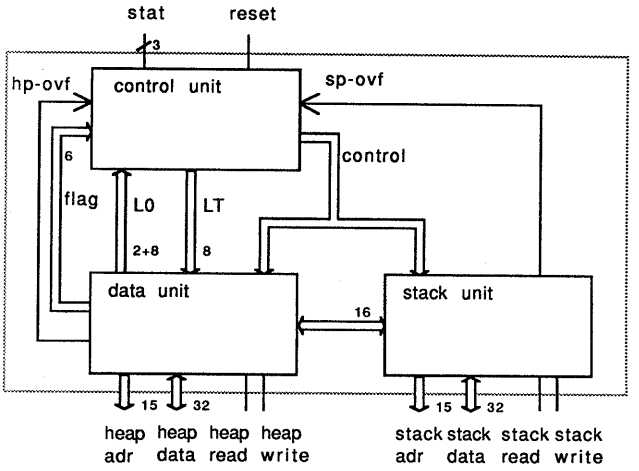


図３　　ブロック図

ここではリダクションチップはバックエンド型で稼働し、関数型言語からコンビネータへの変換はホスト側で行い、リダクションチップはリダクションだけに専念できるものとする。そのコンビネータ式の受渡しは共有メモリであるヒープメモリを介して行い、評価するコンビネータ式の先頭番地や作業領域として使用可能なヒープエリアの先頭番地は特定番地（チップ側アドレス０）を介して行うことにする。　そのほかにリダクションの開始を指示する reset や現在のチップの状態を示す stat が制御信号として用意されている。　stat は３ビットで上位が１のときチップは停止していることを示し、その理由は下位２ビットで示される。上位が０のときはリダクションを実行中であることを示す。　詳しくは表３に示す。

ヒープメモリは２つのデータをもつセルからなる。それぞれのデータ構成を図４に示す。　タイプとしてはセルポインタ、コンビネータ、整数の３タイプで、セルポインタは最上位ビットが１、コンビネータ、整数はそれぞれ上位２ビットが００、０１というタグを持つ１６ビットデータである。

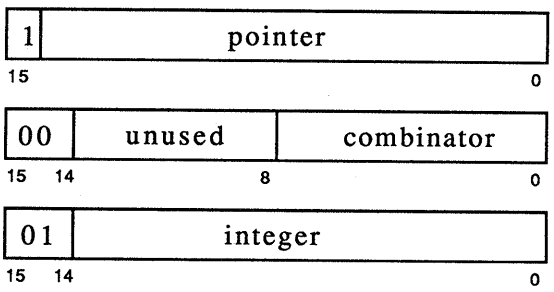


図４　データタイプ

表３　ｓｔａｔ信号

０＊＊	リダクション実行中
１００	ヒープポインタオーバーフローのため停止
１０１	スタックポインタオーバーフローのため停止
１１０	ｓｔｏｐ命令により停止
１１１	ｈａｌｔ命令により停止

コントロールユニットのブロック図を図５に示す。コントロールユニットは普通のマイクロ制御型ＣＰＵと同様であるが、そのほかにヒープポインタオーバーフローが起きたときにガベージコレクションを行うためにマイクロコードポインタを保存するレジスタＧＣＰと、スタックポインタやヒープポインタからのオーバーフローの割り込みを処理するＭＩＮＴなどがある

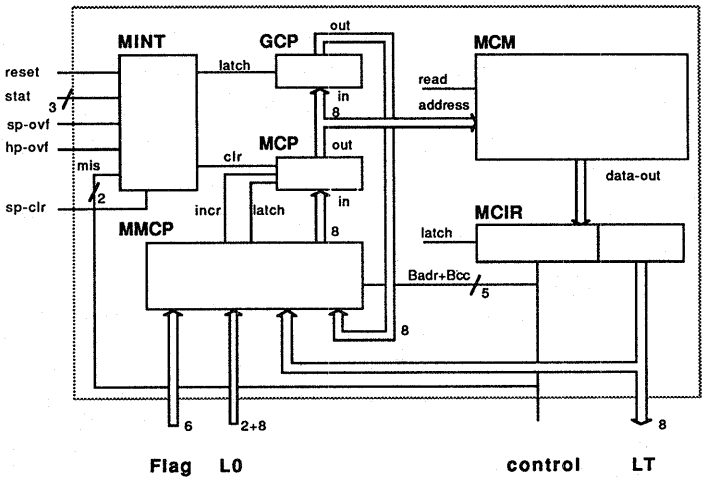


図 5 コントロールユニット

ヒープポインタオーバーフローが起きたときは次のマイクロ命令まで行いその後で停止しガベージコレクションを行う。これはメモリアクセスには２マイクロサイクル必要であり、オーバーフローは第一サイクル後に発生するが、現在のメモリアクセスには影響を与えないため、そのメモリアクセスを終了させるためである。ガベージコレクションはこのチップでは行わずにホストあるいは他のチップで行うが、チップ上のデータもガベージコレクションの対象であるのでこれをメモリ上に退避することが必要であり、これを行うための命令と回復するための命令が用意されている。

データユニットのブロック図を図６に示す。データユニットにはスタックの値を保持する４個のノードレジスタＢ０，Ｂ１，Ｂ２，Ｂ３とヒープメモリアクセス用のアドレスレジスタＡ，データレジスタＤＨ，ＤＬ、新しいセルとして使用可能なヒープアドレスを指すＨＰレジスタ、算術演算回路ＡＬＵやフラグレジスタＦＲなどの機能素子がある。それらは３本のバスＬ０，Ｌ１，Ｌ２によってつながっている。また、ノードレジスタＢ３にはスタックユニットへのバスがつながっているし、マイクロコードのＬＴ領域も３本のバスにつながっている。この３本のバスを利用して同時に３つのレジスタ

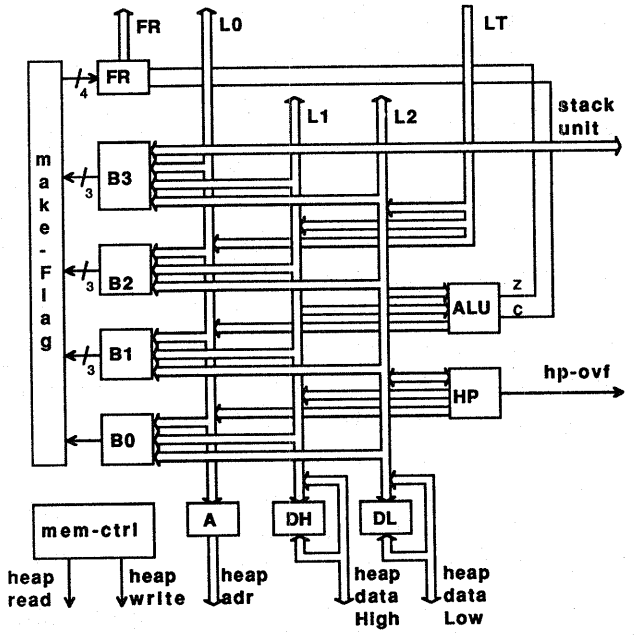


図 6 データユニット

転送を行うことができる。また、メモリアクセスには２マイクロサイクル必要とするが、その間も空いているバスを利用してレジスタ転送を行うこともできる。フラグとしてはＡＬＵの結果のゼロフラグやキャリーフラグの他に、Ｂ０がセルポインタであるかどうかをしめすフラグＣ０、Ｂ１に有効な引数が入っているかどうかを示すフラグＦ１、Ｂ１とＢ２に有効な引数が入っているかどうかを示すフラグＦ２、Ｂ１とＢ２とＢ３に有効な引数が入っているかどうかを示すフラグＦ３がある。有効な引数とは後置型コンビネータでない値ということであり、後置型コンビネータはそこより上が現在評価すべきデータであることを示しているわけである。

スタックユニットのブロック図を図７に示す。スタックユニットは、スタックメモリが３２ビット構成であるのに対し、スタック操作はその半分の１６ビット単位で指定されるため、その調整を行っている。また、ヒープメモリ同様２マイクロサイクル必要なスタックメモリ操作を１マイクロサイクルで行えるように工夫している。これは、スタック操作をスタックユニット内部のレジスタ操作に置き換え、スタック操作の指定がないときにスタックメモリ操作を行うことにより実現している。このためにはスタック操作が指定されていないときに、次のスタック操作の要求に答えられるように、スタックユニット内のレジスタの値を準備して置かなければいけない。ユニット内のレジスタの状態とスタック操作による状態遷移を示しているのが図８のオートマトンである。これより、スタック操作の指定がないときにスタックユニット内のレジスタの状態がこの４つのうちのどれかになるようにスタックメモリ操作を行えばよいことがわかる。ただし、必要なデータは現在のスタックポインタ上及び前後１ワードとし、また、その後には２マイクロサイクル以上のスタック操作無指定ステップがあ

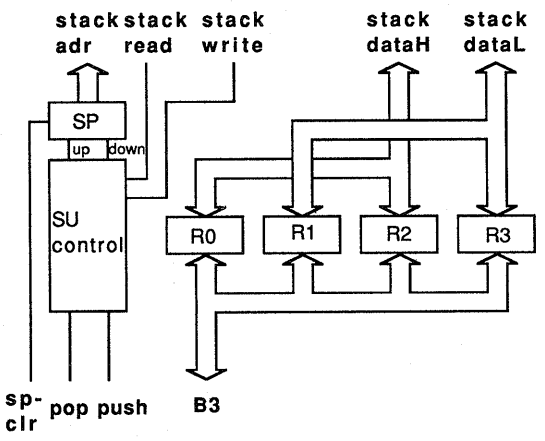
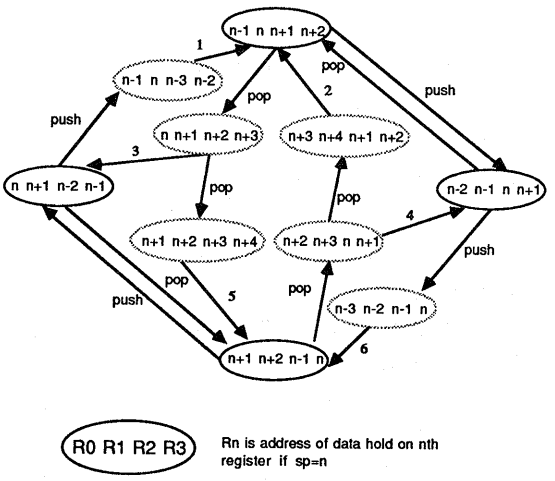


図 7　スタックユニット



operation

1 st R2R3 (SP - 1)　4 ld R0R1 (SP - 1)

2 ld R0R1 (SP)　5 ld R2R3 (SP)

3 ld R2R3 (SP - 1)　6 st R0R1 (SP - 1)

図 8　スタック操作オートマトン

るものとする。

図９にマイクロコードの形式を示す。マイクロコードは４４ビット幅で８ビットのアドレス付けがされている。１８の領域に分かれそれぞれの領域で符号化されている。Ｌ０領域からＬ２領域はバスに出力するソースを指定する。Ｂ０領域からＢ３領域は各レジスタに入力されるソースを指定する。Ａ領域からＨＰ領域は各レジスタのラッチを指定していて、ＨＰ領域はさらにインクリメントの指定も行う。ＭＥＭ領域はヒープメモリ操作を指定し、第１サイクルと第２サイクルの指定も行う。ＡＬＵ領域はＡＬＵの開始タイミングと加算か減算かを指定する。ＲＳ領域はスタックユニット操作を指定する。Ｂａｄｒ，Ｂｃｃ領域は次に実行するマイクロコードポインタを指定する。ここでは上位２ビットのタグによる４方向分岐や下位８ビットのコンビネータの値による間接分岐、フラグの値による条件分岐などいろいろな分岐を指定することができる。ＬＴ領域は分岐先のアドレスや、レジスタにいれるデータなどを示すリテラル部である。８ビットあるために直接マイクロコードアドレスやコンビネータを指定することができる。

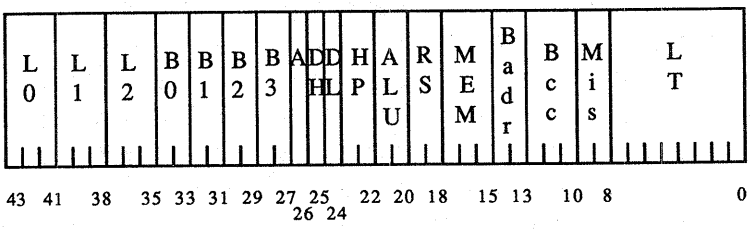


図９　マイクロコード

このチップの特徴をまとめると、

１）３本のバスによるレジスタ間並列転送が可能である
２）タグによる分岐やコンビネータの値による分岐など特別な分岐制御によりすばやく所定の処理を行うアドレスに飛べる
３）コンビネータの引数チェックがフラグ参照だけで行える
４）スタック操作がメモリサイクルの半分の１マイクロサイクルで実現できる
５）ヒープポインタやスタックポインタのオーバーフローは割り込みによって検出されるのでそれらのチェックを行う必要がない

などがあげられる。

これらの特徴により、コンビネータリダクションの各コンビネータ以外の処理には１３ステップのマイクロコードしか必要としない。

４．　評価

このチップをシミュレーションによって評価した結果を次に示す。これによるとサイクルを５Ｍｈｚとすると１秒間に約５０万リダクションが可能であり、これはインタープリタによるリダクションの１００倍から５０倍くらいのスピードアップとなっている。実行されたコンビネータの内訳はＳ，Ｋなどの基本コンビネータが５６％、後置型コンビネータが２３％、その他のコンビネータが２１％となっている。基本コンビネータは引数を各関数に受け渡すために必要なコンビネータであり、引数の数が少く算術計算中心のプログラムでもこれらのコンビネータの割合が多いので高速化が特に必要な部分である。また、後置型コンビネータも２割程度ありこれを再帰呼び出しで行いレジスタ退避などを行っていてはオーバーヘッドが大きすぎて大変である。

ヒープメモリ操作は全マイクロサイクルのうち４３％で行われているが、その

うちの６２％はトラバース操作であり、コンビネータ操作でのヒープメモリ操作の割合は１６％にすぎない。これはヒープメモリアクセスの少ないグラフコピー方式によるものである。また、全マイクロサイクルのうち２９％でスタック操作指定が行われているが、実際のスタックメモリ操作は２３％でしか行われていない。ｐｕｓｈ／ｐｏｐを直接スタックメモリ操作で行うとすると、メモリ操作は２マイクロサイクル必要であるから、増加するマイクロサイクルも考えて、スタック操作には４５％必要となるところをスタックユニットの利用により１７％ですませている。これはｐｕｓｈ／ｐｏｐ操作のうちの３８％しか実際にはメモリ操作を行っていないことを示している。つまり、６２％はスタックユニット内の処理だけで完結し、メモリを参照しなくてもすんでいる。

結果として、このチップの特徴がコンビネータリダクションの高速実行に有効であったといえる。

さらに高速化を図るためには、全体の２７％を占めるトラバース処理をパイプライン化すること。現在はセルポインタの１種類しかないポインタに、リストポインタを加えてリスト処理を速くすること。また、ストリクトコンビネータでは引数が定数であることがわかっていても、後置型コンビネータを利用した変換を行うのでこの辺の処理を、不変コード方式での最適化の方法を取り入れるなどの方法が考えられる。

しかし、これらの方法による高速化はクロックの高速化と合わせても数倍が限度であり、それ以上の高速化のためには並列化が必要となる。そのときにはノイマン型ではない例えばデータフローのようなシステムが有利になると期待される。

参考文献

(1) D.A.Turner, "A New Implementation Technique for Applicative Languages", Software-Practice and Experience, vol. 9, pp. 31-49, 1979

(2) Masato Takeichi, "An Alternative Scheme for Evaluating Combinator Expressions", Journal of Information Processing, vol. 7, no. 4, pp. 246-253, 1984

(3) W.R.Stoye,T.J.W.Clarke and A.C.Norman, "Some Practical Methods for Rapid Combinator Reduction", 1984 ACM Conference on Lisp and Functional Programming, pp. 159-166, 1984