自己簡約グラフによる結合子式の評価

金子敬一†

Turner は結合子を使って関数プログラムを評価する技法を提案した．この技法では結合子式はグラフに変換され，通訳系がこのグラフを簡約して関数プログラムを評価する．このとき，評価結果でグラフのノードを更新することで，完全遅延評価を実現している．この方式をグラフ書換え簡約法と呼ぶ．一方，Takeichi は通訳系による簡約時のポインタ操作を簡単にするため，グラフを複製しthunk機構によって完全遅延性を実現するグラフ複製簡約法を提案した．本論文では，グラフ簡約法におけるスタックの先頭要素に対する型検査を必要としない評価方式を提案する．この方式で使用するグラフは自己簡約的であり，通訳系を必要としない．このため高速にグラフを簡約することができる．またグラフ簡約に基づくため，高階関数の扱いも Turner, Takeichiの方式と同等の能力を持つ．さらに本評価方式は通常のハードウェア上で簡単に実現可能であり，計算機上で行った実験結果も示す．

Self-Reduction Graphs for Evaluation of Combinator Expressions

KEIICHI KANEKO†

Turner has proposed an evaluation scheme of functional programs based on combinators. In this scheme, combinator expressions are translated into graphs, and an interpreter evaluates the original functional programs by reducing the graphs. Fully lazy evaluation is achieved by updating the nodes of the graphs with their evaluation results. This scheme is called the graph rewriting reduction scheme. Takeichi has, on the other hand, proposed the graph copying reduction scheme which copies the graphs to simplify the pointer operations by the interpreter in reduction steps. The scheme makes use of the thunk mechanism to implement full laziness. This paper proposes a new approach based on graph reduction in which type checks of the top elements of the stack are eliminated. In this approach, graphs are self-reducible and no interpreter is required. Hence, the graphs are reduced quickly. In addition, the approach has ability of handling the higher order functions because it is based on the graph reduction scheme. The approach is easily implemented on conventional hardware. The experimental results are also presented.

1. はじめに

Turner[4]が提案したグラフ書換え簡約法では，**S**, **K**, **I**, **B**, **C** などあらかじめ定めた結合子からなる式を，関数と引数の対をノードとするグラフで表現し，結合子の簡約規則に基づき簡約を実行する．この評価系はグラフのノードを指すポインタを格納するスタックとグラフを実現するためのヒープ領域を持ち，スタックの先頭要素に対する制御のもとでグラフの書換えによって簡約を行う．グラフ書換え簡約法では，常に簡約結果によってグラフが更新されるので，共有されている結合子式は高々一度しか評価されない．

† 東京大学工学部計数工学科
Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo

Takeichi[3]によるグラフ複製簡約法は，グラフ書換え簡約法における煩雑なポインタ操作を避けるためにグラフの書換えをあきらめ，簡約結果を表現するグラフの複製を構成する．このためスタックにはノードへのポインタではなく引数を載せればよい．グラフ複製簡約法では，共有される結合子式が二度以上評価される可能性を避けるために，共通部分式に対しては書換えを行う thunk 機構を導入している．

本論文では，グラフのノードに機械命令を埋め込んだ自己簡約グラフを構成することで通訳系の処理を代行し，スタックの先頭要素に対する型検査を除去して高速な簡約を実現する方式を提案する．また本方式に基づいて実現したグラフ書換え簡約法とグラフ複製簡約法の実験結果を示す．

以下，第2章においては，関数プログラムから結合子式への翻訳規則とグラフの構成，および通訳系によ

るグラフ簡約の方法とその問題点について説明する．次に第3章では，前章の問題点の解決策として，自己簡約グラフによる結合子式の実行方式を提案する．さらに，グラフ書換え簡約法とグラフ複製簡約法に対して，この方式を実際の計算機上で実現し，実験を行った結果を第4章に示す．最後にこの方式の利点と問題点について，第5章で述べる．

2. 結合子によるグラフ簡約方式

2.1 翻　訳

結合子を用いるグラフ簡約方式は，翻訳と実行という二つの段階に分かれる．翻訳段階では入力プログラムを結合子式に翻訳し，これからグラフを構成する．実行段階では通訳系を用いるなどして，グラフを簡約して評価結果を得る．本節では第一段階である翻訳とグラフの構成について説明する．

以下で定義される二つの結合子 **S** と **K** の組合せによって，任意の λ 式を実現可能であるということは，結合子論理においてよく知られた事実である．

$\mathbf{S}\,f\,g\,x = f\,x\,(g\,x)$

$\mathbf{K}\,x\,y = x$

Turner は結合子式の大きさを抑えて簡約速度を向上させるために，以下に示す結合子 **I**, **B**, **C** を提案した．

$\mathbf{I}\,x = x$

$\mathbf{B}\,f\,g\,x = f\,(g\,x)$

$\mathbf{C}\,f\,g\,x = f\,x\,g$

さらに条件式や算術演算のための結合子も導入している．

$\mathbf{IF}\,e\,t\,f = t$ if $e = \mathbf{true}$, f otherwise

$+\,x\,y = x+y$, etc.

$=\,x\,y = \mathbf{true}$ if $x=y$, **false** otherwise, etc.

これらの結合子は入力プログラム中でも，組み込み関数として使用される．

一般の λ 式を結合子式に翻訳するには以下の規則をこの順の優先度で用いる．

$\lambda x.\,x \Rightarrow \mathbf{I}$

$\lambda x.e \Rightarrow \mathbf{K}\,e$, if $x \notin e$

$\lambda x.\,(e_0\,x) \Rightarrow e_0$, if $x \notin e_0$

$\lambda x.\,(e_0\,e_1) \Rightarrow \mathbf{S}(\lambda x.\,e_0)(\lambda x.\,e_1)$, if $x \in e_0$ and $x \in e_1$

$\lambda x.\,(e_0\,e_1) \Rightarrow \mathbf{B}\,e_0(\lambda x.\,e_1)$, if $x \notin e_0$ and $x \in e_1$

$\lambda x.\,(e_0\,e_1) \Rightarrow \mathbf{C}(\lambda x.\,e_0)e_1$, if $x \in e_0$ and $x \notin e_1$

階乗関数 *fac* および Fibonacci 関数 *fib* に対する λ 式

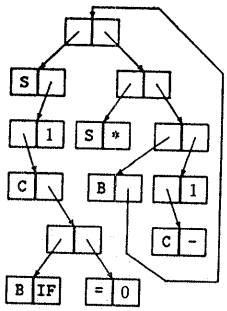


図 1 *fac* に対するグラフ
Fig. 1 Graph representation for *fac*.

$fac = \lambda n.\,(\mathbf{IF}(=\,0\,n)1(*\;n\,(fac(-\,n1))))$

$fib = \lambda n.\,(\mathbf{IF}(<n\;2)1$

$\qquad (+(fib(-\,n\,1))(fib(-\,n\,2))))$

を結合子式に翻訳したものを以下に示す．

$fac = \mathbf{S}(\mathbf{C}(\mathbf{B}\;\mathbf{IF}(=\;0))1)(\mathbf{S}\,*(\mathbf{B}\,fac(\mathbf{C}-1)))$

$fib = \mathbf{S}(\mathbf{C}(\mathbf{B}\;\mathbf{IF}(\mathbf{C}<2))1)$

$\qquad (\mathbf{S}(\mathbf{B}+(\mathbf{B}\,fib(\mathbf{C}-1)))(\mathbf{B}\,fib(\mathbf{C}-2)))$

結合子式からグラフを構成するには，結合子式の各要素を葉に見立てて，関数と引数の結合順序に従って二分木を構成していけばよい．このとき，関数適用は左結合的であることに注意する．すなわち $(f\;x\;y)$ という式は，$((f\;x)\;y)$ のように解釈する．先の *fac* に対する結合子式から構成したグラフを**図1**に示す．グラフの各ノードは関数部と引数部からなる．

2.2 実　行

本節では Turner の提案したグラフ書換え簡約用の通訳系による実行段階について説明する．グラフ簡約にはスタックを使う．簡約は，まず評価すべきグラフへのポインタをスタックにプッシュすることで開始する．通訳系はスタックの先頭に積まれているものがポインタである限り，そのポインタの指すノードの関数部をスタックにプッシュし続ける．スタックの先頭に結合子が現れたときは，その結合子に応じた規則でグラフを簡約する．結合子 **B** に対する簡約の様子を**図2**に示す．スタックは下に向かって成長し，新た

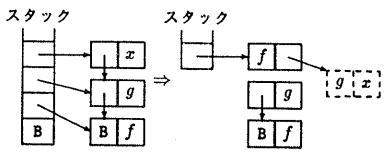


図 2 結合子**B**に対する簡約
Fig. 2 Reduction for combinator **B**.

表 1 各要素のスタックの先頭への出現回数
Table 1 Occurrence counts of each element on top of stack.

	S	K	I	B	C	IF	*	+	−	=	<	ポインタ
fac 10	21	0	29	21	21	11	10	0	10	11	0	313
fib 10	265	0	351	441	530	177	0	88	176	0	177	5472

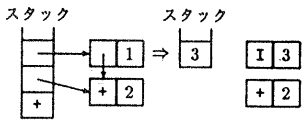


図 3 間接ノードの生成
Fig. 3 Generation of indirection node.

に割り当てられたノードは破線で示されている．簡約した結果，スタックの先頭に再びポインタが現れた場合は，先ほどと同様の操作を繰り返す．一方，簡約した結果，スタックの先頭に値が積まれていれば，その値を返す．条件式や算術演算のための結合子の簡約に対しては，通訳系は再帰的に呼び出される．また，簡約の結果が複合式とならないような結合子 (**K**, **IF**, +, etc.) に対しては，結合子 **I** を使って間接ノードを生成する．この様子を**図 3** に示す．

表 1 に (*fac* 10) と (*fib* 10) の簡約時に各結合子とポインタがスタックの先頭に現れた回数を示す．これよりポインタの出現回数が無視できないほど多いことがわかる．したがって，各結合子に対する簡約操作に加えて，スタックの先頭要素の型検査を行い，その結果に応じて制御を分岐する機構も高速な簡約のためには無視できないことがわかる．次章では自己簡約グラフと呼ばれ，スタックの先頭要素に対する型検査を必要としない方式を提案する．

3. 自己簡約グラフ

3.1 グラフ書換え簡約法

まずグラフ書換え簡約法のための自己簡約グラフについて説明する．自己簡約グラフとは，通訳系の役割を代行するために，前章に示したグラフのノードに機械命令を埋め込んだものである．グラフ書換え簡約法の場合，通常のハードウェアが備えているサブルーチンの呼び出し命令を，ノードの関数部に埋め込む．例えば，図 1 のグラフを自己簡約グラフに変換すると，**図 4** に示すものとなる．各ノードにおいて関数部と引数部は，それぞれ上下に位置している．ここで **CALL** は，サブルーチンの呼び出しのための仮想命令であり，**PTR** と **INT** はデータの属性を表すためのタグ

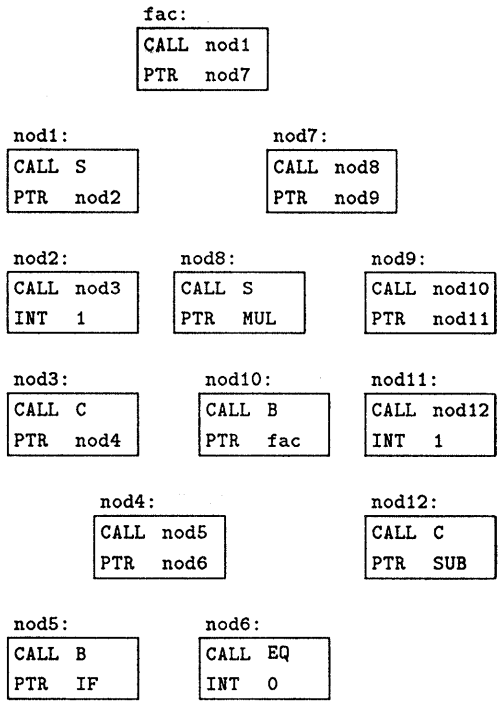


図 4 *fac* に対する自己簡約グラフ
Fig. 4 Self-reduction graph for *fac*.

である．各結合子には，対応するラベルとグラフ簡約を実行する命令列を用意しておく．通常のハードウェアでは **CALL** 命令の実行後，スタックにはその直後のアドレスが戻り番地としてプッシュされる．スタックに積まれたこのアドレスを，ノードへのポインタとして簡約時に利用するわけであるが，これらはノードの途中を指している．これからノードの先頭のアドレスを得るためには，変位 **Dsp** を使って補正する．結合子 **B** に対する仮想命令列は，**図 5** のようになる．**Fun**, **Arg** は，それぞれノードの先頭から，その関数部，引数部が保持するポインタやデータを参照するための変位である．また **R 0**, …, **R 3** はレジスタを表す．**図 6** は結合子 **B** に対する自己簡約の様子を示している．この簡約後，ノード **nod 1** への分岐命令 **JMP** で簡約を再開する．

間接ノードを生成する結合子のうち必ず値を返すもの（+, =, etc.）に対応するコード列の最後はスタックをほどいて復帰する命令からなる．他の結合子のうち値を返す可能性のあるもの (**K, I, IF**) は型検査を行い，値であれば同様に復帰し，ポインタであればそれが指す先へ分岐するような命令からなる．型検査を必要とするのは，この場合のみである．結合子 **K** に対する仮想命令列は**図 7** のようになる．

```
B:      ALLOC   R0                      |R0 <- pointer to an allocated node
        POP     R3
        POP     R2
        MOV     (R2-Dsp+Arg), (R0+Fun)  |(R0+Fun) <- (R2-Dsp+Arg)
        POP     R1
        MOV     (R3-Dsp+Arg), (R1-Dsp+Fun)|(R1-Dsp+Fun) <- (R3-Dsp+Arg)
        MOV     (R1-Dsp+Arg), (R0+Arg)  |(R0+Arg) <- (R1-Dsp+Arg)
        MOV     R0, (R1-Dsp+Arg)        |(R1-Dsp+Arg) <- R0
        JMP     (R1-Dsp)
```

図 5 結合子**B**に対する仮想命令列
Fig. 5 Sequence of virtual instructions for combinator **B**.

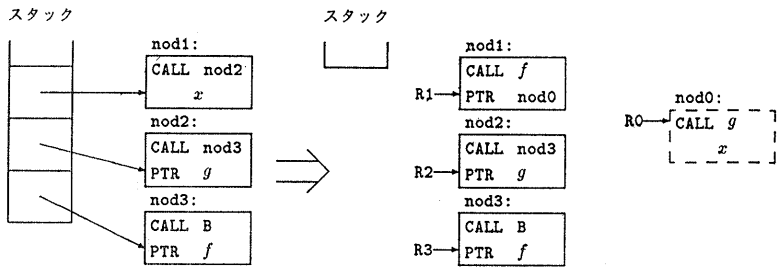


図 6 結合子**B**に対する自己簡約
Fig. 6 Self reduction for combinator **B**.

```
K:      POP     R2
        POP     R1
        MOV     #I, (R1-Dsp+Fun)        |(R1-Dsp+Fun) <- address of I
        MOV     (R2-Dsp+Arg), (R1-Dsp+Arg)
        IFI     (R2-Dsp+Arg), K1        |if (R2-Dsp+Arg) is an integer, goto K1
        MOV     (R2-Dsp+Arg), R0
        JMP     (R0)
K1:     MOV     (R2-Dsp+Arg), R0
        UNLK
        RET
```

図 7 結合子**K**に対する仮想命令列
Fig. 7 Sequence of virtual instructions for combinator **K**.

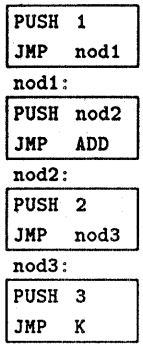


図 8 結合子式 (**add**(**K** 3 2) 1) に対するグラフ複製簡約法における自己簡約グラフ
Fig. 8 Self-reduction graph for combinator expression (**add**(**K** 3 2)1) in graph copying reduction scheme.

3.2 グラフ複製簡約法

グラフ複製簡約法では，ノードへのポインタではなく引数をスタックに載せていく．したがって今回はサブルーチンへの呼び出し命令は使用できない．グラフのノードに埋め込む命令は，引数をスタックに載せる **PUSH** 命令と関数部の保持するポインタの先へ制御を移す **JMP** 命令からなる．この場合，各ノードでは関数部が下，引数部が上となっている．結合子式 (+ (**K** 3 2) 1) を自己簡約グラフにすると**図 8** となる．また各結合子に対する命令列も，グラフ複製簡約を実行するように変更する．

4. 実験結果

インテル社の i 80286, 8 MHz を CPU とする計算機上で，自己簡約グラフを実現して実験を行った．以下グラフ書換え簡約法のための実現について説明するが，グラフ複製簡約法においても同様である．グラフ書換え簡約法のための実現では，グラフの各ノードは 8 バイトからなる．i 80286 にはセグメント内の絶対番地によるサブルーチン呼び出し命令がないので，まずレジスタに絶対番地をロードし，レジスタによる呼び出し命令を実行することとした．このため初めの 1 バイトはロード命令であり，これに続く 2 バイトは呼び出し先の絶対番地を保持している．次の 2 バイトはレジスタによるサブルーチン呼び出し命令である．その次の 2 バイトは，整数値あるいはポインタを格納するための領域である．最後の 1 バイトは 0 を保持しており，これはノードを整列させるためのものである．整数値をポインタと区別するために，最上位ビットをセットすることとした．このため整数値は 15 ビットの符号付き表現となっている．たとえば図 4 に示した *fac* に対する自己簡約グラフは

```
fac:   mov   ax, offset nod1
       call  ax
       dw    nod7
       db    0
nod1 : mov   ax, offset S
       call  ax
       dw    nod2
       db    0
       ...
nod11 :mov   ax, offset nod12
       call  ax
       dw    INT+1
       db    0
nod12 :mov   ax, offset C
       call  ax
       dw    SUB
       db    0
```

となる．

結合子に対する命令列は，先に述べた仮想命令列をマクロ展開し，さらにハードウェアに依存した最適化を施して生成した．結合子 **B** に対する命令列を示す．

```
B:    or    cx, cx
      jnz   B1
      call  gc
      jcxz  spacerr
B1 :  mov   bx, cx
      mov   cx, word ptr [bx+Fun]
      pop   si
      pop   di
      mov   ax, word ptr [di]
      mov   word ptr [bx+Fun], ax
      pop   di
      nov   ax, word ptr [si]
      mov   word ptr [di−Dsp+Fun], ax
      mov   ax, word ptr [di]
      mov   word ptr [bx+Arg], ax
      mov   word ptr [di], bx
      sub   di, Dsp
      jmp   di
```

簡約の結果，値を返す場合はレジスタ **ax** を使用する．また記憶管理のため，レジスタ **cx** が自由リスト

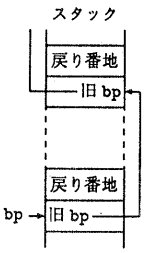


図 9 再帰的な自己簡約のためのスタックフレーム
Fig. 9 Stack frames for recursive self-reduction.

表 2 (*fib* 20) の実行時間 (単位：秒)
Table 2 Execution time of (*fib* 20) (unit is second).

書換え	通訳系	6.58
	自己簡約	5.50
複製	通訳系	6.46
	自己簡約	5.15

表 3 (*nfib* 19) による1秒間の関数呼び出し回数
Table 3 Numbers of function calls per second by (*nfib* 19).

書換え	通訳系	3.07×10^3
	自己簡約	3.68×10^3
複製	通訳系	3.11×10^3
	自己簡約	3.92×10^3

の先頭を保持している．ノードの割り当て時に自由リストが空になっている場合はゴミ集めを起動する．

再帰的に自由簡約を実行するために，**図 9** のようなスタックフレームを導入した．ここでもスタックは下に向かって成長するように表されている．フレームを構成するための仮想命令

LINK addr

に対しては

```
mov     ax, offset addr
push    ax
push    bp
mov     bp, sp
```

のようにマクロ展開する．

(*fib* 20) に対する各方式による実行時間を**表 2** に示す．グラフ書換え簡約法，グラフ複製簡約法いずれの場合も自己簡約グラフを用いたほうが通訳系による簡約よりも2割程度実行時間が少ないことが確かめられた．また以下で定義される関数 *nfib* を使って，1秒間の関数呼び出し回数を測定した．

$$nfib=\lambda n.(\mathbf{IF}(\leqq n\,1)1(+\,1(+(nfib(-\,n\,1))\\ (nfib(-\,n\,2)))))$$

その結果を**表 3** に示す．この性能評価でも，いずれの簡約法も自己簡約グラフを用いたほうが2割以上良い結果を示している．

5. 議　　論

自己簡約グラフでは，グラフのノードに埋め込まれた命令列が通訳系を代行して，スタック上の要素に対する型検査を除去する．通訳系を使ったグラフ書換え簡約法において，通訳系の核部分は**図 10** に示すような命令列からなる．グラフ複製簡約法では，**PUSH** 命令が **R 0** ではなくて，**(R 0+Arg)** をオペランドとするように変更される．自己簡約グラフでは，これらの命令列が行う操作を一つの命令 **CALL**，あるいは **PUSH** と **JMP** といった二つの命令で実行しており高速な簡約を実現している．

Jones[2] は **S**, **K**, **I**, **B**, **C** を用いた結合子式を不変コードに翻訳して実行する方式を提案している．この方式は実行速度や記憶容量においてグラフ簡約法よりも優れているが，Takeichi[3] が指摘しているように高階関数の扱いを考慮していない点に問題がある．これに対して本方式も通訳系を必要としないコードに翻訳するが，グラフ簡約の性質を失わないので高階関数も問題なく扱うことができる．

高階関数の具体例として，以下で定義される高階関数 *twice* とそれを使った関数 *tst* を考える．

$$twice=\lambda f.(\lambda x.(f\ (f\ x)))$$
$$tst=twice\ twice\ twice(+\ 1)$$

関数 *twice* を結合子式にすると

```
NEXT:   PUSH    R0              |push the pointer onto the stack
        MOV     (R0+Fun), R0    |R0 <- (R0+Fun)
        IFP     R0, NEXT        |if R0 is a pointer, goto NEXT
        JMP     (R0)            |jump into the predefined codes
```

図 10 グラフ書換え簡約法の通訳系の核部分
Fig. 10 Kernel part of interpreter for graph rewriting reduction.

表 4 自己最適化による関数 *tst* の改善
Table 4 Improvement of a program by self optimisation.

	S	I	B	+	ポインタ
一度目	7	19	19	16	129
二度目	0	15	15	16	92

twice=**S B I**

となり，*tst* に引数を与えてグラフ簡約により評価すると，*tst* の定義は内部展開されて以下のようになる．

```
tst=B(B(B(B(+ 1)(I(+ 1)))
          (I(B(+ 1)(I(+ 1)))))
       (I(B(B(+ 1)(I(+ 1)))
          (I(B(+ 1)(I(+ 1)))))))
    (I(B(B(B(+ 1)(I(+ 1)))
          (I(B(+ 1)(I(+ 1)))))
       (I(B(B(+ 1)(I(+ 1)))
          (I(B(+ 1)(I(+ 1)))))))
```

このうち多くの部分がポインタによって共有されているため，実際にはそれほど多くの記憶領域を必要とはしていない．この結果，再び *tst* に引数を与えて評価させると，一度目より簡約段数が少なくなる．Turner[4]は，このことを自己最適化特性と呼んでいる．**表 4** に，(*tst* 0) の評価時にスタックの先頭に現れる要素が一度目と二度目でどのように変化するかを示す．Jones[2]による不変コード方式では，*tst* の評価結果を通常の関数と同様に保持することはできず，二度目の評価に使用するためには特別な機構を必要とする．

一方，自己簡約グラフのように頻繁に分岐を行う実行は，CPU が持つ命令キャッシュをうまく利用できない点が問題である．これを克服するためには Clarke[1]に示されるような専用機械を開発するという方法もある．さらに今回の実現では，グラフ書換え簡約法，グラフ複製簡約法のいずれにおいても通訳系を用いた場合は，各ノードが 4 バイトからなるのに比べて，自己簡約グラフでは，さらに 4 ないし 6 バイトも必要とする点は問題である．これは，実験に用いた計算機の CPU である i80286 が，絶対アドレスによるサブルーチン呼び出し命令や分岐命令，あるいは即値をスタックに積む命令を持たないため，これらの値を一旦レジスタにロードするための命令を必要とするからである．たとえばモトローラ社の MC68000 など，このような命令を持つ CPU を備えた計算機では，各ノードについてより少ない記憶領域の増加で自己簡約グラフを構成でき，実行速度もさらに向上すると考える．

6. 結　論

グラフのノードに命令列を埋め込むことで通訳系を使わずにグラフ簡約が可能であることを示した．この方式は以下の利点を持つ．

- スタック上の要素に対する型検査を実行する必要がないので高速である，
- グラフ簡約に基づくので高階関数の利用が簡単である，
- 単純な仮想命令で記述できるので多くのハードウェア上で実現可能である．

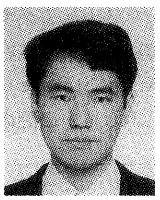
他の計算機における実現と実験は今後の課題である．

謝辞 東京大学工学部機械情報工学科の寺田 実助教授には，自己簡約グラフを記述する際の仮想命令の使用についてご指導いただいた．また東京大学教育用計算機センターの岩崎英哉助教授には，本論文の原稿について何度も貴重なご意見をいただいた．お二人に深く感謝する．なお本研究の一部は文部省科学研究費補助金（奨励研究(A)05780231：完全遅延評価による関数プログラムの部分計算）の援助による．

参 考 文 献

1) Clarke, T. J. W., Gladstone, P. J. S., MacLean, C. D. and Norman, A. C.: SKIM—The S, K, I Reduction Machine, *Conf. Rec. of the 1980 LISP Conf.*, pp. 128-135 (1980).
2) Jones, N. D. and Muchnick, S. S.: A Fixed-Program Machine for Combinator Expression Evaluation, *Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming*, pp. 11-20 (1982).
3) Takeichi, M.: An Alternative Scheme for Evaluating Combinator Expressions, *J. Inf. Process.*, Vol. 7, No. 4, pp. 246-253 (1984).
4) Turner, D. A.: A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.*, Vol. 9, pp. 31-49 (1979).

(平成 5 年 9 月 6 日受付)
(平成 6 年 2 月 17 日採録)



金子 敬一（正会員）
1962 年生．1985 年東京大学工学部計数工学科卒業．1987 年同大学大学院工学系研究科情報工学修士課程修了．現在，同大学計数工学科助手．関数プログラミング，並列プログラミング，部分計算等の研究に従事．日本ソフトウェア科学会，ACM 各会員．