結合子式の評価系の実現法について

武市正人

（電気通信大学計算機科学科）

I. 結合子式の評価系の比較（概要）

関数型言語の実用的な処理系を実現するにあたって、結合子（combinator）の簡約（reduction）に基づいて関数を評価するいくつかの方法を提示し、実験結果をもとにこれらを比較する。

結合子を用いて関数を評価する方式は、1979年にTurnerによって示されてから注目されてきている。Turnerによる標準的な結合子を用いるもののほか、Hughesによって、関数に応じて選んだ超結合子（super-combinator）を用いる方法も提案されている。しかし、これらの結合子で構成された結合子式（combinator expression）の評価系（evaluator）の実現法について論じたものは見当らない。

ここでは、Turnerのグラフ書換え簡約法（Graph Rewriting Reduction）に対して、新たにグラフ複製簡約法（Graph Copying Reduction）と不変コード（fixed code）による評価方式を示し、これらの方式を比較し、検討する。評価系は、現在の通常の（conventionalな）計算機の上で実現するものと仮定している。

（これは、理化学研究所シンポジウム：関数的プログラミング（1984-07-21）で発表したものである。）

II. 不変コードによる完全遅延評価（概要）

関数型言語に対して、結合子式を簡約する完全遅延評価法（fully lazy evaluation）には高階関数を評価する際に、通常の遅延評価にはみられない望ましい性質がある。グラフの簡約によって完全遅延評価を行なうには、TurnerやHughesによる結合子式の簡約法があるが、通常の計算機では、グラフの簡約よりも不変コードによるものの効率がよい。

遅延評価法を実現する不変コードの生成についてはJohnssonに与えられているが、そこでは完全遅延性は考慮されていない。ここでは完全遅延評価に適した不変コードの生成法の概略を述べる。

I. 結合子式の評価系の比較

1. はじめに

筆者らは1983年に関数型言語MLの処理系をVAX11/780からMelcomに移植した[5]が、その経験から、関数型言語の実用的な処理系の移植性を高めるには、手続き型言語の場合と同様に、コード生成部の設計が重要であることを認識した。MLでは、仮想のスタック機械FAM(Functional Abstract Machine)の命令を中間コードとして採用しているが、このコードから実在する計算機の機械語を生成する方式では計算機に依存する部分が大きく、実行時支援系(runtime routine)との関係も複雑なものになってしまっている。この問題は、手続き型言語の処理系の移植に関して多年にわたって研究されてきているものであるが、関数型言語の処理系に対しては言語の特徴的な性質を反映させたコード生成の方式を考えることも有効であろう。その可能性を追究するために、ここでは結合子の簡約に基づく方式を考察する。

評価系を実現する上で特定の関数型言語の機能に密接に関連させることは避け、遅延評価方式(lazy evaluation)を標準のものとし、直積データ(レコード)、直和データ、抽象データ型なども扱う言語の処理系を実現するものとする。遅延評価方式は結合子と相性がよいばかりでなく、評価系の核となる部分を小さくすることができるので、評価系の移植にも適していると期待される。また、例外処理(exception handling)のような特殊な制御をもつ関数型言語の処理系の実現にも有効であると考えられる。このように拡張することができる評価系は、さまざまな関数型言語の処理系の後置部(back-end)として用いることができる。すなわち、言語ごとに用意される前置部(front-end)とともに計算機ごとに用意される評価系を用いて処理系を構築できることになる。このとき、結合子式は前置部と後置部の接合点に置かれる中間コードであると考えられる。

以下では、結合子式の評価系を実現する方式を比較する。3種類の方式を示すが、形式的で詳細な記述は行なわず、図を用いて、背景にある基本的な考え方を述べることとする。最後に、これらの評価系を用いて標準的な結合子による結合子式と、超結合子による結合子式の簡約を行なった実験結果を考察する。

2. グラフ書換え簡約法(GRR)

Turner[7]に述べられているグラフ書換え簡約法(Graph Rewriting Reduction、GRR)は、S、K、I、B、Cなどから構成される結合子式を、関数と引数の対を節点(node)とするグラフで表現し、結合子の簡約規則にしたがって簡約を繰り返すものである(図1)。この評価系は、グラフの節点を指す指標(pointer)を格納するスタックLAS(Left Ancestor Stack)と、グラフを実現するヒープ領域(heap)とをもち、簡単な制御機構のもとでグラフの書換えによって簡約を行なう。GRRでは結合子式の簡約結果を、常に、もとの結合子式を表わしている節点に残すようにグラフを書き換えるので、共有されている結合子式は一度しか評価されない。共通式に対してこのような評価を行なう(すなわち、call-by-needを実現する)ために、LASには結合子式そのものではなく、結合子式を表わしている節点への指標を載せる必要がある。

GRRをHughes[2]の超結合子に対して適用することもできるが、この場合は、一般に、簡約結果のグラフがもとの結合子式のものに比べてかなり大きくなり、新たに多くの節点が作られることが予想される(図2)。

3. グラフ複製簡約法(GCR)

グラフ複製簡約法(Graph Copying Reduction、GCR)[6]はGRR評価系とは異なり、原則的にはグラフの節点を書き換えることはしないで、簡約結果を表現するグラフの複製を作る方式である(図3)。GCRの評価系では、スタックARGS(ARGument Stack)に結合子式の値を載せる。ヒープ領域とARGSとのあいだでデータ(結合子式の表現)を移動するときに、数語を一括して転送する命令を有効に使うことを考えて、ヒープ領域の節点は対だけではなく、任意個の連続した語の塊(chunk)を用いて表現する。このような記憶領域の使用法は、ごみ集め(garbage collection)が複雑になることはあるが、直積データを効率よく実現するためにも必要なものである。

簡約の際に単純にグラフの複製を作るだけでは、GRRによれば一度しか評価されない部分式が2回以上評価されることもあり得る。これを避けるために、共通部分式に対しては例外的に書換えを行なう結合子THUNKを用意し、一度評価して得られた結果をもとの結合子式と置き換えるものとする(図4)。図では、THUNKは2語で表現し、eval/deliverの印をつけることにしている。THUNKを通じて式を評価したときには、evalをdeliverに変えて、もとの式のかわりにその結果を置く。deliverの印のついたTHUNKが評価されるときには単に、その値を簡約結果とする。

このようなTHUNKは、不変コードに対して、Jones[3]によって用いられている。また、Henderson[1]のLispKit LispでSECD機械を拡張したものには、これと類似の機能をもつrecipeが使われている。

GCRで簡約の際に、THUNKを用意する必要があるのは、簡約結果の結合子式の形式の中に2回以上現われている引数に対するものだけである。したがって、標準的な結合子S、K、I、B、Cのうちで THUNKが使われるのは、Sだけである(図4)。このように、THUNKを必要とする引数は簡約規則から直接求められるので、GCRは超結合子に対しても容易に適用することができる(図5)。また、GCRにおける関数と引数との結合関係の表現は、超結合子のように不定個の引数が現われるものには自然なものであるといえる。

4. 不変コード方式(FC)

不変コード(fixed code)による結合子式の評価方式FCでは、GRRやGCRのように結合子式をグラフで表現するのではなく、通常の手続き型言語の処理系における目的コードと同様に、簡約を行なうための機械語の命令に制御の流れを置くことによって表現する。Jones[3]には、標準的な結合子SKIBCを用いる不変コードの方式が述べられているが、ここでは、より一般的に、超結合子にも適用することのできる方式を示す。

ここで扱う不変コード方式の考え方は、結合子の簡約によるものと対比して扱われる関数の評価方式で用いられる閉包(closure)を利用するものである。すなわち、結合子式は、それを簡約するための機械語のコードC(code)とその式に現われる引数の値からなる環境E(environment)の対で表現される。簡約を行なう際には、コードと環境の対を載せるスタックCES(Code-Environment Stack)を用いる。結合子の簡約では、スタックにある環境をヒープ領域に移して、それを簡約結果の結合子式のすべての部分式の環境とする(図6)。超結合子の簡約においては、簡約結果の結合子式のすべての部分式が同じ環境をもつものとして扱うことができるので、環境を関数の呼出しごとに作る必要はない。

5. 実験結果と評価

3種類の評価系GRR、GCR、FCを用いて、いくつかの関数に対して結合子SKIBCと超結合子SCのそれぞれによる結合子式を簡約した実験結果を示す。

Hughes[2]には、結合子SKIBCと超結合子との比較実験の結果が簡略に述べられているが、その実験には、BCPLが使われている。評価系の具体的な実現法は明らかではない。Peyton Jones[4]には、結合子の方式と閉包を用いる方式との比較が行なわれているが、そこでもBCPLが使われている。Jones[3]には、結合子SKIBCによる結合子式をLispの不変コードに変換して評価する方式が述べられている。これらはどれも評価系を実現しているBCPLやLispの処理系に依存しているが、ここでは、現在の計算機で実用的な評価系を構築する方式を比較するのが目的であるので、評価系を実現する言語処理系に依存するのは望ましくない。実験に用いた評価系は、すべて、筆者が同じ計算機(Melcom 70/250)に、同じ程度の最適化を行なって、アセンブラで書いたものである。評価系の主要部はいずれも小さく、1Kステップ以内である。

5.1 実験1: f

整数値をとる関数で、簡約回数がきわめて多いものとして、図7の関数fを選び、nの値を変えながら(f 0 n 2n)の値2nを求めた。評価に要した時間とヒープ領域の大きさを図8に示す。いずれもnの2乗に比例しているが、同じ関数を遅延評価によらないで計算すると30のn乗に比例する。

n=30の場合に評価系で簡約した結果は表1のようになった。

表1

時間 (msec.)

結合子	GRR	GCR	FC
標準	8646	7428	-
SKIBC	(2.85)	(2.45)	-
超結合子	-	3030	1572
SC	-	(1)	(0.52)

ヒープ領域 (語)

結合子	GRR	GCR	FC
標準	149432	120626	-
SKIBC	(1.32)	(1.06)	-
超結合子	-	113446	66432
SC	-	(1)	(0.59)

関数型言語MLの評価方式は、標準的には遅延評価ではないが、関数fの引数に対して評価を遅らせることとして、超結合子によるものと実行時間を比較したものが表2である。MLの関数fは、fそのものを結合子として簡約する場合に対応しているので、fを超結合子として簡約を行なった。

表2

GCR(SC)	FC(SC)	ML
1910	1060	1048
(1.82)	(1.01)	(1)

時間(msec.)

5.2 実験2: nth-prime

遅延評価の特徴的な例である無限リスト(ストリーム)を用いる関数として図9に示すようなn番目の素数を求める関数nth-primeに対して実験を行なった。

n=30に対して簡約に要した時間とヒープ領域の大きさは表3の通りである。

表3

	GRR(SKIBC)	GCR(SC)	ML
時間	1950	836	492
(msec.)	(2.33)	(1)	(0.59)
ヒープ領域	30572	14916	-
(語)	(2.05)	(1)	-

5.3 実験3: nFib

Henderson[1]には、遅延評価を取り入れたSECD機械のインタプリタのベンチマークの結果が述べられている。関数

```
nFib n = if n<=1 then 1 else nFib(n-1)+nFib(n-2)+1
```

を用いて1秒間に実行できる関数呼出しの回数を計ったものである。GRRとGCRに対するものを表4に示す。

表4

Melcom 70/250		ICL Perq		VAX11/780	
GRR(SKIBC)	GCR(SC)	Pascal	Microcode	Pascal	Assembler
561	1430	75	2K	200	900
		Henderson[1]		Henderson[1]	

5.4 評価

上の結果から、GCR評価系では、結合子SKIBCを用いた場合に比べて、超結合子SCによるものの方が2倍以上の速さであることがわかる。Hughes[2]にも述べられているが、超結合子の方が簡約の単位が大きく、簡約回数が少なくて済むことによるものと考えられる。

SKIBC結合子に対しては、GRR評価系とGCR評価系とのあいだに有意な差は認められないが、GCRの方が超結合子に対する拡張性の点で優れているものと考えられる。

超結合子に対するGCR評価系とFC評価系では、FCの方が2倍程度高速であることがわかる。また、必要とされるヒープ領域の大きさもFCの方が小さいので実用的な評価系を構成する点では不変コードの方式が優れているといえる。

MLの処理系で生成するコードは、必ずしも遅延評価に適しているものではないが、超結合子を用いた不変コードによる評価系では、それと同じ程度の効率が期待できる。コード生成の方式や拡張性の点では超結合子によるものの方が有利であると考えられる。

6. おわりに

ここで扱った評価系のうちで、超結合子に対するGCR評価系は、核になる部分が小さく、結合子の簡約コードは翻訳時に生成することができる。標準的な結合子や特殊な結合子、および既に得られている結合子に関する簡約コードをライブラリとして蓄えておくのも簡単である。結合子式と簡約規則は計算機に依存しないので、関数型言語の処理系の移植に適しているといえよう。

不変コード方式もGCR評価系と同じように扱うことができるが、結合子式そのものは、陽には表現されないので、簡約規則に基づくコード生成に依存する部分が大きい。このコード生成系は、GCRのものと同じ程度のものと考えられるが、対象とする計算機の命令に依存する割合は、当然、不変コード方式の方が高い。

これらのことから、不変コード方式が、時間と記憶容量の点で優れているといえるが、GCR評価系は容易に高水準言語によって実現することもできるし、評価系はインタプリタによるものなので、デバッグを行なうための処理系には便利である。

参考文献

[1] Henderson, P., Jones, G.A., and Jones, S.B.: The LispKit Manual, Technical Monograph PRG-32, Oxford University Computing Laboratory, 1983.

[2] Hughes, R.J.M.: Super-combinators, Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming, pp.1-10.

[3] Jones, Neil D., and Muchnick, Streven S.: A Fixed-Program Machine for Combinator Expression Evaluation, ibid., pp.11-20.

[4] Peyton Jones, S.L.: An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions, ibid., pp.150-158.

[5] 武市正人: ソフトウェア評論-関数型言語ML、コンピュータソフトウェア、vol.1、No.2、岩波書店、1984.

[6] Takeichi, M.: An Alternative Scheme for Evaluating Combinator Expressions, to appear.

[7] Turner, D.A.: A New Implementation Technique for Applicative Languages, Software-Practice and Experience, 9(1979), pp.39-49.

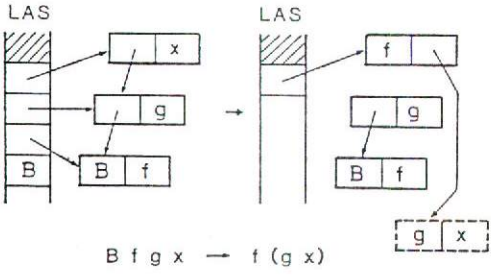

Fig. 1

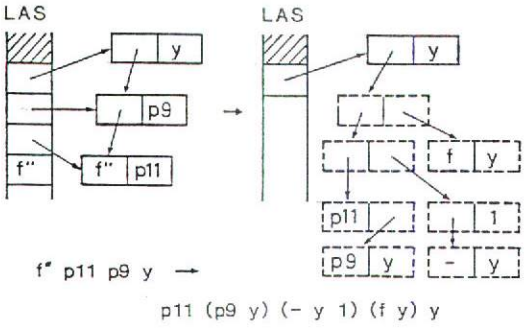

Fig. 2

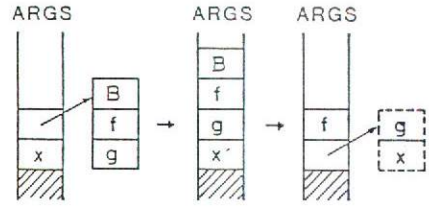

Fig. 3

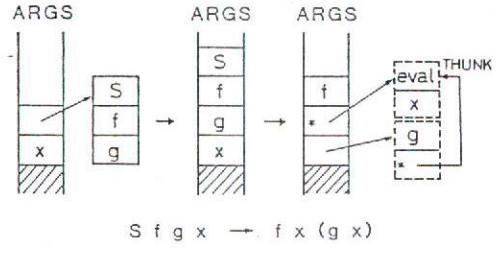

Fig. 4

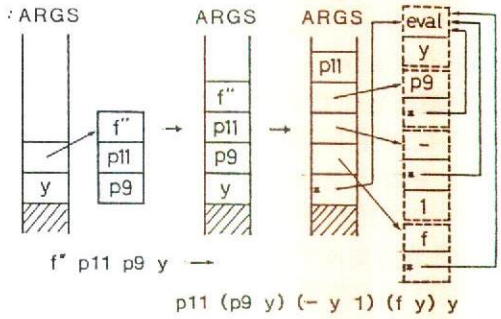

Fig. 5

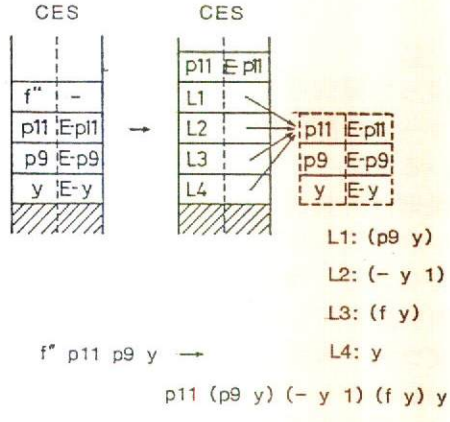

Fig. 6

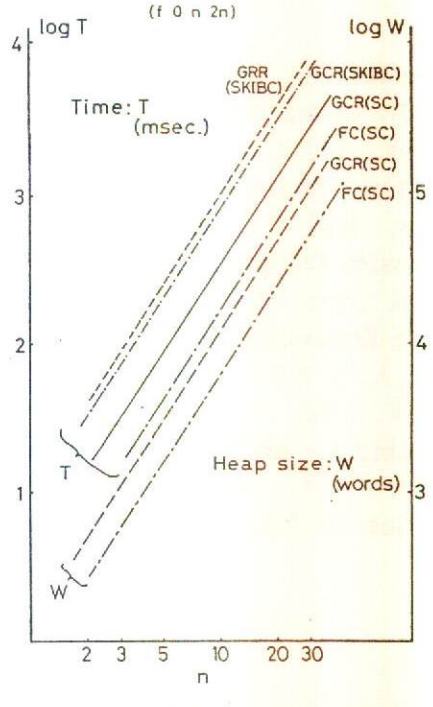

Fig. 8

```
Function f:
        f x y z = if z>y then
                              f (f y z (x-1))
                                (f z x (y-1))
                                (f x y (z-1))
                            else
                              y

SKIBC combinator expression for f:
        (S
         ( B
           S
           (B C)
           ( B
             (S (B S (B (B IF) (C >))))
             ( S
               ( B
                 S
                 ( B
                   (B S)
                   ( S
                     (B S (B (B S) (B (B f)) (B (C (B C f)) (C - 1)))))
                     (C (B B (B C (C f))) (C - 1)) ) ) )
                 (C (B C (B (B B) f)) (C - 1)) ) ) ) )
          I )

Super-combinators f', f'', and f''' for f:
        f  ->  f'
        f' x  ->  f'' (f''' x (- x 1)) (f x)
        f'' p11 p9 y  ->  p11 (p9 y) (- y 1) (f y) y
        f''' p7 p6 p10 p8 p5 p4 z  ->
                    IF (> z p4) (f (p5 z p6) (f z p7 p8) (p10 (- z 1))) p4
```

Fig. 7

```
Functions:
    nth-prime n  =  nth n primes
    primes  =  sieve (from 2)
    from n  =  n : (from (n+1))
    sieve (a:x)  =  a : (sieve (filter (λn. n%a<>0)) x)
    filter p (a:x)  =  if p a then a : (filter p x) else filter p x
    nth n (a:x)  =  if n=1 then a else nth (n-1) x

SKIBC combinator expressions:
    primes: (sieve (from 2))
    from:   (S P (B from (C + 1)))
    sieve:  (U (S (B B P)
                  (B (B sieve)
                     (B filter (C (B C (B (B <>) (C %))) 0)) ) ))
    filter: (B U
             (S
              (B C (B (B S) (S (B S (B (B B) (B IF))) (B (C (B B P)) filter))))
    nth:    (B U (S (B C (B (B B) (B IF (C = 1)))) (B nth (C - 1))))

Super-combinators:
    primes  ->  sieve (from 2)
    from  ->  from'
        from' n  ->  cons n (from (+ n 1))
    sieve  ->  sieve'
        sieve' (a:x)  ->  cons a (sieve (filter (sieve'' a) x))
        sieve'' p9 n  ->  <> (% n p9) 0
    filter  ->  filter'
        filter' p  ->  filter'' (filter p) p
        filter'' p13 p12 (a:x)  ->  IF (p12 a) (cons a (p13 x)) (p13 x)
    nth  ->  nth'
        nth' n  ->  nth'' (IF (= n 1)) (- n 1)
        nth'' p7 p8 (a:x)  ->  p7 a (nth x p8)
```

Fig. 9

II. 不変コードによる完全遅延評価

1. 遅延評価法

以下では、図1に示すような簡単な関数型言語を対象として考える。図1の意味記述に忠実にラムダ式の簡約を行なう解釈系を実現すれば、名前呼び(call-by-name)の意味にあった評価系が得られる。このとき、ラムダ式に対する引数は、実際に値が必要とされるまでは評価しないという遅延評価(lazy evaluation)の方式がとられる。単純な遅延評価の方式では、同じ引数の評価が何度も行なわれるときには、値呼び(call-by-value)に比べて効率が悪くなる。これを解決するために、必須呼び(call-by-need)の方法が用いられる。必須呼びの実現に適した言語の定義は、図2のように与えられる。引数に対して、それが最初に参照されるときに名前呼びで処理を行ない、それからあとでは値呼びで値を得るものである。図2では環境の書換えによる状態の変化と評価の順序を明示するために、状態Sと接続(continuation)K、Cを用いている。

2. 完全遅延評価法

遅延評価の方式には、結合子式の簡約に用いられるグラフ書換えの方法や、拡張SECD機械のコードのような不変コード方式がある。不変コードの方法では、簡約の際に、変数に対する値を得るために環境を参照するが、グラフの書換えでは環境は用いない。

遅延評価法では、関数の引数として渡される式は、その値が必要になったときに初めて評価される。グラフの書換えによる方法ではグラフで表わしている式をその値を表わすグラフで置き代えることによって必須呼びを実現している。また、すべての式が、一度評価されるとそれ以降は値に置き代えられるという性質を持っている。Hughes[1]はこのような評価法を完全遅延評価(fully lazy evaluation)と呼んでいる。グラフの簡約法では、プログラムもグラフであるので、プログラムの一部分をその値で置き代えることも自然であるが、不変コード方式では、完全遅延評価を実現するためには、新たな機構が必要である。

3. 不変コードによる結合子式の完全遅延評価

結合子式に対して完全遅延評価を行なう不変コードとしては、図2に示したような必須呼びを実現する際の環境の書換えを利用することが考えられる。図1の言語の局所的な宣言を一般化した言語の式に対して以下のような変換を施して、超結合子を用いた結合子式を得ることができる。

(1) fn I: E

Eに含まれる極大部分式(maximal free expression)e1, ..., ekを抜き出し、この部分を超結合子を用いた式に変換する:

$$(\overline{fn}\ x1\ \ldots\ xk\ I : E')\ e1\ \ldots\ ek$$

この変換はHughes [1]のラムダ式に対するものと同じである。超結合子

$$\varphi = (\overline{fn}\ x1\ \ldots\ xk\ I : E')$$

は、Eに現われるeiをxiで置き代えて得られる式E'と、外側のfnによる束縛に対してe1 ... ekの極大部分式ができるだけ大きくなるように、それらに対応するx1, ..., xkの順序を定めることによって構成される。

(2) E where I1=E1 and ... Im=Em

I1, ..., Imに対するEの極大部分式e1, ..., ekを抜き出してx1, ..., xkで置き代える:

$$(\overline{fn'}\ y1\ \ldots\ yp : E')\ f1\ \ldots\ fp$$

$$yj \in \{I1, \ldots, Im, x1, \ldots, xk\}$$

$$fj \in \{E1, \ldots, Em, e1, \ldots, ek\}$$

このときにも外側の束縛に対して極大部分式が大きくなるようにyj、fjの順序を定める。

超結合子

$$\varphi' = (fn'\ y1\ \ldots\ yp : E')$$

に対しては常に十分な個数の引数が与えられるので、これは極大部分式の入れ換えをするための

結合子であると言える。

(3) E whererec I1=E1 and ... Im=Em

EおよびE1, ..., Emの、I1, ..., Imに対する極大部分式e1, ..., ekを求めて次のような結合子式に変換する:

$$(\overline{fn}^{*}\ x1\ \ldots\ xk\ :\ E'\ \text{whererec}\ I1=E1'\ \text{and}\ \ldots\ Im=Em')\ e1\ \ldots\ ek$$

超結合子

$$\varphi^{*}=(\overline{fn}^{*}\ x1\ \ldots\ xk\ :\ E'\ \text{whererec}\ I1=E1'\ \text{and}\ \ldots\ Im=Em')$$

は再帰的な定義を扱うものであり、E', E1', ..., Em'にはI1, ..., Im, x1, ..., xk以外の変数は現れない。

図3に変換の例を示してある。

これらの超結合子に対する簡約規則は図2の関数bind、rebindを拡張したものを用いて図4のように定めることができる。ただし、図4の簡約の関数Rは、図2のものとは異なり、定義領域に関しては非形式的な記述になっている。図4による簡約の規則は、状態と接続とを用いているので、不変コードとして実現するのに適した表現であり、これをもとに不変コードを得るのは容易である。

4. 結合子式の完全遅延評価と部分評価

結合子式を得るための変換の際に検出している極大部分式は、関数の特定の引数が定まることによってその値を決めることのできる部分式の極大のものである。したがって、高階関数の部分評価(partial evaluation)の際には、与えられる引数に依存する部分式の評価をたかだか一度ですませられるようになる。

図4のRによる簡約は、Johnsson [2] のものと同じく、式の最も外側に簡約できる部分がなくなるまで繰り返される。Johnssonは高階関数に対しては、すべての引数が与えられないと簡約を行なわないので、部分評価によって得られる関数を用いて評価を行なうときにももとの関数を用いるものと変わらない。すなわち、関数

```
f x1 ... xn
```

の部分評価として、関数

```
g = f e1 ... em
```

を定め、gを用いた二つの式

```
g a1 ... ap,      g b1 ... bp      (p=n-m)
```

を評価するときにはe1, ..., emが2回評価されることになる。

また、条件の判定に用いるIFを不変コードの制御の流れに変換することも部分評価においては問題になる。(IF e)が部分評価による効果を受けられるようにするには、IFを結合子として扱うのが便利である。ここで取りあげた言語でIFなどを特別のものとしていないのはこのことによる。

参考文献

[1] Hughes, R.J.M.: Super-combinators, Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming, pp.1-10.

[2] Johnsson, T.: Efficient Compilation of Lazy Evaluation, Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, pp.58-69, 1984.

Syntactic Domains

$B \in Bas$ Basic values

$I \in Ide$ Identifiers

$E \in Exp$ Expressions

Syntax

$E ::= B \mid I \mid \text{fn } I : E \mid E\ E \mid E \text{ where } I = E \mid E \text{ whererec } I = E$

Semantic Domains

$\mathbb{B}$ Basic values

$\varepsilon \in \mathbf{E} = \mathbb{B} + \mathbf{D} \to \mathbf{E}$ Expressible values

$\delta \in \mathbf{D} = \mathbf{E}$ Denotable values

$\rho \in \mathbf{U} = Ide \to \mathbf{D}$ Environment

Semantic Functions

$\mathcal{B} : Bas \to \mathbb{B}$ (omitted)

$\mathcal{E} : Exp \to \mathbf{U} \to \mathbf{E}$

$\mathcal{E}[\![B]\!]\rho = \mathcal{B}[\![B]\!]$

$\mathcal{E}[\![I]\!]\rho = \rho[\![I]\!]$

$\mathcal{E}[\![\text{fn } I : E]\!]\rho = \lambda\delta.\ \mathcal{E}[\![E]\!](\rho[\delta/I])$

$\mathcal{E}[\![E_1\ E_2]\!]\rho = (\mathcal{E}[\![E_1]\!]\rho)(\mathcal{E}[\![E_2]\!]\rho)$

$\mathcal{E}[\![E_1 \text{ where } I = E_2]\!]\rho = \mathcal{E}[\![E_1]\!](\rho[\mathcal{E}[\![E_2]\!]\rho/I])$

$\mathcal{E}[\![E_1 \text{ whererec } I = E_2]\!]\rho = \mathcal{E}[\![E_1]\!](fix(\lambda\rho'.\ \rho[\mathcal{E}[\![E_2]\!]\rho'/I]))$

Fig. 1

Semantic Domains

$\mathbb{B}$ Basic values

$e \in \mathbf{E} = Exp \times \mathbf{U} + \mathbf{E} \to \mathbf{K} + \{\phi\}$

$u \in \mathbf{U} = Ide \to \mathbf{L}$ Environment

$s \in \mathbf{S} = \mathbf{L} \to \mathbf{E}$ State

$l \in \mathbf{L}$ Location

$c \in \mathbf{C} = \mathbf{S} \to \mathbf{A}$ Continuation

$k \in \mathbf{K} = \mathbf{E} \to \mathbf{C}$

$d \in \mathbf{D} = \mathbf{U} \to \mathbf{C}$

Auxiliary Functions

$new : \mathbf{S} \to \mathbf{L}$ allocates new location

$bind : \mathbf{U} \to (Ide \times \mathbf{E}) \to \mathbf{D} \to \mathbf{C}$

$bind\ u\ (I, e)\ d\ s = d(u[l/I])(s[e/l])$ where $l = new\ s$

$rebind : \mathbf{U} \to (Ide \times \mathbf{E}) \to \mathbf{C} \to \mathbf{C}$

$rebind\ u\ (I, e)\ c\ s = c(s[e/uI])$

$find : \mathbf{U} \to Ide \to \mathbf{K} \to \mathbf{C}$

$find\ u\ I\ k\ s = k(s(uI))\ s$

Semantic Functions

$\mathcal{B} : Bas \to \mathbf{E}$ (omitted)

$\mathcal{R} : \mathbf{E} \to \mathbf{K} \to \mathbf{C}$

$\mathcal{R}(B, u)\ k = k(\mathcal{B}[\![B]\!])$

$\mathcal{R}(I, u)\ k = find\ u\ I\ (\lambda e'.\ \mathcal{R}\ e'\ (\lambda e''.\ rebind\ u\ (I, e'')(k\ e'')))$

$\mathcal{R}(\text{fn } I : E, u)\ k = k\ (\lambda e'k'.\ bind\ u\ (I, e')(\lambda u'.\ \mathcal{R}(E, u')\ k'))$

$\mathcal{R}(E_1\ E_2, u)\ k = \mathcal{R}(E_1, u)\ (\lambda e'.\ e'\ (E_2, u)\ k)$

$\mathcal{R}(E_1 \text{ where } I = E_2, u)\ k = bind\ u\ (I, (E_2, u))(\lambda u'.\ \mathcal{R}(E_1, u')\ k)$

$\mathcal{R}(E_1 \text{ whererec } I = E_2, u)\ k = bind\ u\ (I, \phi)(\lambda u'.\ rebind\ u'\ (I, (E_2, u'))(\mathcal{R}(E_1, u')k))$

Fig. 2

$(el\ 2)$ whererec $el = \text{fn } n : (\text{fn } s : (IF(= n\ 1)(hd\ s)(el\ (-\ n\ 1)(tl\ s))))$

⇩

$\varphi_1^* = \overline{\text{fn}}^*\ () : (el\ 2)$ whererec $el = \varphi_2\ el$

$\varphi_2 = \overline{\text{fn}}\ \$0\ n : (\varphi_3\ (IF\ (=\ n\ 1))(\$0\ (-\ n\ 1)))$

$\varphi_3 = \overline{\text{fn}}\ \$2\ \$1\ s : (\$2\ (hd\ s)\ (\$1\ (tl\ s)))$

Fig. 3

$\mathcal{R}(\varphi\ e_1 \cdots e_k, u)\ k = bind\ u\ (\langle x_1, \cdots, x_k, I\rangle, \langle (e_1, u), \cdots, (e_k, u), \phi\rangle)$
$(\lambda u'.\ k\ (\lambda e'k'.\ rebind\ u'\ (I, e')(\mathcal{R}(E', u')\ k')))$

$\mathcal{R}(\varphi'\ f_1 \cdots f_p, u)\ k = bind\ u\ (\langle y_1, \cdots y_p\rangle, \langle (f_1, u), \cdots, (f_p, u)\rangle)(\lambda u'.\ \mathcal{R}(E', u')\ k)$

$\mathcal{R}(\varphi^*\ e_1 \cdots e_k, u)\ k = bind\ u\ (\langle x_1, \cdots, x_k, I_1, \cdots, I_m\rangle, \langle (e_1, u), \cdots, (e_k, u), \phi, \cdots, \phi\rangle)$
$(\lambda u'.\ rebind\ u'\ (\langle I_1, \cdots, I_m\rangle, \langle (E_1', u'), \cdots, (E_m', u')\rangle)(\mathcal{R}(E', u')k))$

Fig. 4