部分評価を使った自己反映言語のコンパイル

中川　理恵　　　浅井　健一
お茶の水女子大学大学院人間文化研究科
rie@pllab.is.ocha.ac.jp
asai@is.ocha.ac.jp

概 要

近年、新しいプログラミング言語のパラダイムの１つとして、アスペクト指向プログラミングが注目を集めている。アスペクト指向プログラミングでは、横断的関心事を一ヶ所にまとめて記述し、それをプログラムに織り込む事でモジュール性を高めている。一方、自己反映言語を使っても、一部の横断的関心事については同様の事をメタレベルを書き換える事で実現でき、かつ、部分評価器でコンパイルしメタレベルを潰す事がアスペクト指向プログラミングでの織り込みに対応する。そこで、本稿では、いくつかの横断的関心事を自己反映型言語を使って実現し、更に、部分評価による実行の高速化を図る実験を行った。本稿では、関数型言語 Scheme に基づく自己反映言語を使ってコンパイルをしたが、ここでの結果がアスペクト指向プログラミングでのコンパイルにも示唆を与える事を目指している。

1　はじめに

近年、新しいプログラミング言語のパラダイムの１つとして、アスペクト指向プログラミング (AOP)[8] が注目を集めている。AOP では、横断的関心事を一ヶ所にまとめて記述し、それをプログラムに織り込む事でモジュール性を高めている。一方、自己反映言語 [12] を使っても、一部の横断的関心事については同様の事を実現できる [13]。ここで考えている自己反映言語とは、メタレベルへ上がる事ができ、メタレベルインタプリタを書き換える事ができる言語の事である。一部の横断的関心事はインタプリタのレベルでは一ヶ所にまとまっており、そのためユーザプログラムを解釈実行しているメタレベルを書き換える事でアスペクト指向プログラミングで実現されている機能を自己反映言語でも実現できる。

自己反映言語を使う利点は解釈実行という基本的な概念のみでアスペクトのような概念も説明できることである。複雑な構造を持つアスペクトでもそれを自己反映言語の枠組みで記述できればそこでの知見から効率的な実現方法を見つけることができるかもしれない。実際、この方向性でアスペクトを理解する研究もなされている [9]。

一方、自己反映言語を使う欠点は、実行時間が非常に掛かることである。自己反映言語、特に behavioral な自己反映言語は、インタプリタが書き換えられると余分な解釈実行のレベルが入り、通常 100 倍のオーダで遅くなる。しかし近年、部分評価の技法の発展により、この効率の欠点が少しずつ解消されるようになってきた。部分評価 [6] とは、プログラムに部分的な入力を与えて、その入力に最適化されたプログラムを生成する技法である。

そこで本稿では、いくつかの横断的関心事を関数型言語 Scheme [7] に基づく自己反映言語を使って実現し、それを部分評価することで AOP の効率的な実現方法を探っていく。関数型言語に基づく自己反映言語を使用しているのは、部分評価との親和性が高いためである。関数型言語をベースにおく自己反映言語を使うと、多くの部分を副作用なしで書けるため（副作用の問題を完全に回避するのはまだ難しいものの）扱いは楽になる。同様のアプローチは、すでに増原ら [9] によってなされているが、本研究はベースレベル言語、メタレベル言語がともに関数型である点、部分評価の起動自体をインタプリタに組み込んでいるためコンパイルまで含めた言語セマンティクスを議論する土台となり得る点が異なっている。

このようなシステムのもとで種々のアスペクトを記述、コンパイルの実験を行なった。現段階ではまだ一部のアスペクトしか実現できていないが、ユーザ定義のインタプリタのもとでの部分評価するという非常に一般的な枠組みでコンパイルに相当することが可能となってきている。この結果は部分評価による織り込みが不可能でなく、AOP のコンパイル方法の１つになりうることを示している。将来的には、このような実験を通してアスペクト指向プログ

ラミングでのコンパイルにも示唆を与える事を目指している。また、言語セマンティクスとの関連などの見通しが良いため、新しいアスペクト指向言語の機能や言語要素を構築するための実験環境にも適していると思われる。

以下、2 節で自己反映言語 Black の紹介を行い、3 節で 4 種類のアスペクトを Black の上で実装する。4 節では Black の内部構造について説明し、5 節で部分評価を使ったコンパイルの枠組を述べる。6 節では、3 節で実装したアスペクトを部分評価を使ってコンパイルし、どのくらいの速度向上が得られたかを報告する。7 節で関連研究に付いて述べ、8 節でまとめる。

2 Black の構造

本研究で使用する自己反映言語は Black[2] という Scheme ベースの自己反映言語である。メタレベルに Scheme で書かれた Scheme のインタプリタ（メタサーキュラーインタプリタ）が動いており、ユーザはそれを自由に書き換えることができるようになっている。

2.1 メタレベルのインタプリタ

メタレベルのインタプリタを書き換えるためには、インタプリタの構造を知っておく必要がある。メタレベルインタプリタは、標準的な継続渡し形式 (Continuation Passing Style; 以下 CPS と略す) で書かれている。Black では、このインタプリタがユーザーから見えている。

```
(define (base-eval exp env cont)
  (cond ((number? exp) (cont exp))
        ((symbol? exp) (eval-var exp env cont))
        ((eq? (car exp) 'if) (eval-if exp env cont))
        ((eq? (car exp) 'define) (eval-define exp env cont))
        ((eq? (car exp) 'set!) (eval-set! exp env cont))
        ((eq? (car exp) 'lambda) (eval-lambda exp env cont))
        ((eq? (car exp) 'begin) (eval-begin (cdr exp) env cont))
        ((eq? (car exp) 'exec-at-metalevel)  (eval-EM exp env cont))
        ((eq? (car exp) 'pe) (eval-pe exp env cont))
        (else (eval-application exp env cont))))
```

メイン関数である base-eval が Scheme の構文に合わせて dispatch し、それぞれを処理する関数へと引き渡す。それらの関数は例えば以下の様な形をしている。eval-application は関数を処理する関数で、eval-var は変数を処理する関数である。

```
(define (eval-application exp env cont)
  (eval-list exp env          ; 関数と引数を順に評価し
    (lambda (l) (base-apply (car l) (cdr l) env cont)))) ; 関数を適用する
(define (eval-var exp env cont)
  (let ((pair (get exp env))) ; get は env から exp の値を取ってくる関数
    (if (pair? pair)          ; 返ってきたものが pair なら、値が定義されている
        (cont (cdr pair))
        (my-error  (list 'eval-var: 'unbound 'variable: exp) env cont))))
```

base-eval に出てくる pe という命令は、後に部分評価をするときに使う命令である。また exec-at-metalevel 文については次に触れる。

2.2 書き換えのための構文

インタプリタを書き換えるには、以下の命令を実行する。

(exec-at-metalevel 式)
exec-at-metalevel でメタレベルへ上がり、そこで式を実行する。exec-at-metalevel 文を複数回繰り返して使えば任意のレベルに行くことが可能である。(しかし、本稿では 2 レベル以上、上がることはしない。)

図 1 はメタレベルのインタプリタ内の関数 eval-application が書き換わった時の様子を表している。ここで、実線の箱はデフォルトのインタプリタ関数で直接実行されているもの、点線の箱はユーザ定義の関数で、もう一つ上のレベルのインタプリタによって解釈実行されるものである。図の例ではユーザプログラムは元の eval-application ではなく、レベル 2 のインタプリタで解釈実行されているユーザ定義の eval-application で解釈実行されている。

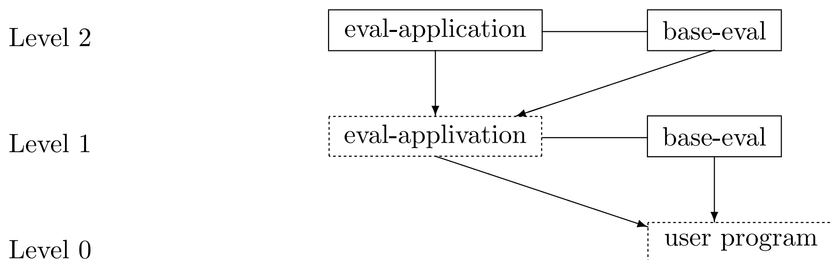

図 1: Black のタワー

3　プログラム例

メタレベルに上がりインタプリタを書き換えると、ある種のアスペクトを実現することができる。ここではいくつか簡単なアスペクトを Black の上で実装する。

3.1　変数のトレース

最初に変数のトレースをとる命令を考える。メタレベルのインタプリタを書き換えることで、Black に (trace 変数) という新しい命令を加える。これで、指定された変数にアクセスするたびに、その値を表示することにする。この命令を使えるようにするためにインタプリタを以下の様に書き換える。(0-1>等は Black のプロンプトである。)

```
 1 0-1> (exec-at-metalevel (begin
 2        (define original-application eval-application)
 3        (define (eval-trace exp1 env1 cont1)
 4          (let ((old-var eval-var) ; オリジナルの eval-var をとっておく
 5                (var (car exp1)))
 6            (set! eval-var (lambda (exp env cont)
 7                  (if (eq? exp var) ; exp が指定した変数だったら
 8                      (begin (write (cdr (get exp env))) (newline))) ; その値を表示する
 9                  (old-var exp env cont))) ; もとの eval-var の処理を行う
10            (cont1 'trace-changed)))
11        (set! eval-application ; eval-trace への dispatch を追加
12              (lambda (exp env cont)
13                (if (eq? (car exp) 'trace) (eval-trace (cdr exp) env cont)
14                  (original-application exp env cont))))))
15 0-1: eval-application
```

11 行目以降で eval-application を書き換えて trace 文への dispatch を加えている。trace 文は 3 行目から定義されている eval-trace で解釈実行される。内部では eval-var を書き換えて、目的の変数がアクセスされたら、その値を表示するようにしている。

この変更は、すべてユーザプログラムであることに注意しよう。インタプリタの書き換えは、あらかじめ Black に用意されている機能であり、単にその機能を使って trace を実現している。

ここで階乗を求める関数 fac を定義しよう。

```
0-2> (trace n)
0-2: trace-changed
0-3> (define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
0-3: fac
```

fac は内部で変数 n を使用しているため、以下の様に n の値を（n にアクセスするたびに）出力していく形になる。

```
0-4> (fac 2)
2
2
2
1
1
1
0
0-3: 6
```

上の定義中に n は 3 度でているため 3 回ずつ表示される。

3.2 before / after

AOP の技法の一つに、結合点モデルがある。あらかじめ、式を挿入できる結合点を決めておいて、そこに好きな式を挿入していく技法の事である。ここでは、その中で before と after を実装する。メタレベルのインタプリタを書き換えることで、Black に以下のようなふたつの新しい命令を付け加える。

(before 関数名 式) 指定された関数が呼ばれる前に、式を実行する。実行される式は (lambda (引数) ...) の形をしていて、関数呼び出しの引数を参照することができる。

(after 関数名 式) 指定された関数が呼ばれた後に、式を実行する。実行される式は (lambda (結果) ...) の形をしていて、関数呼び出しの結果を参照することができる。

例えば、

```
(before 'fac (lambda (n) (display "enter : ") (write n) (newline)))
```

とすると、関数 fac が呼び出される前にその関数の引数を表示する。また、

```
(after 'fac (lambda (result) (display "result: ") (write result) (newline)))
```

で関数を処理した後にその結果を表示する。

この before, after 文を使える様にするためにインタプリタを次の様に書き換える。

```
 1 0-1> (exec-at-metalevel (begin
 2        (define original-application eval-application)
 3        (define (eval-before exp1 env1 cont1) ; before 文を処理する関数
 4          (eval-list exp1 env1 (lambda (l)
 5            (let ((func (car l))        ; 捕まえる関数名
 6                  (ex1 (car (cdr l))))   ; 実行する関数
 7              (set! original-application ; original-application を書き換える
 8                    (lambda (exp env cont)
 9                      (eval-list exp env (lambda (l)
10                        (if (eq? (car exp) func)        ; func への呼び出しなら
```

```
11                              (apply ex1 (cdr l))) ; ex1 を実行
12                          (base-apply (car l) (cdr l) env cont))))))
13              (cont1 'before-changed))))
14          (define (eval-after exp1 env1 cont1) ; after 文を処理する関数
15            (eval-list exp1 env1 (lambda (l)
16              (let ((old-application original-application)
17                    (func (car l))         ; 捕まえる関数名
18                    (ex1 (car (cdr l))))   ; 実行する関数
19                (set! original-application ; original-application を書き換える
20                      (lambda (exp env cont)
21                        (old-application exp env (lambda (m)
22                          (if (eq? (car exp) func) ; func への呼び出しなら
23                              (ex1 m))             ; ex1 を実行
24                          (cont m))))))
25              (cont1 'after-changed))))
26          (set! eval-application ; eval-before/after への dispatch を追加
27                (lambda (exp env cont)
28                  (cond ((eq? (car exp) 'before) (eval-before (cdr exp) env cont))
29                        ((eq? (car exp) 'after) (eval-after (cdr exp) env cont))
30                        (else (original-application exp env cont)))))))
31 0-1: eval-application
```

26 行目以降で eval-application を書き換えて before/after への dispatch を加えている。before/after 文はそれぞれ 3 行目、14 行目から定義される eval-before/eval-after で解釈実行される。内部ではそれぞれ関数名をチェックして、目的の関数ならアドバイスを実行している。

先ほどの fac の例を実行してみよう。

```
0-2> (before 'fac (lambda (n) (display "enter : ") (write n) (newline)))
0-2: before-changed
0-3> (after 'fac (lambda (result) (display "result: ") (write result) (newline)))
0-3: after-changed
0-4> (define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
0-4: fac
```

すると以下のように fac の入出力のトレースを得ることができる。

```
0-5> (fac 3)
enter : 3
enter : 2
enter : 1
enter : 0
result: 1
result: 1
result: 2
result: 6
0-5: 6
```

3.3 cflow

cflow とは、関数の呼出関係を記述する方法で、ある関数が指定された関数の中から呼び出されていた場合にアドバイスを実行する。ここでは、その簡略版として (cflow 'f 'g) としたら、関数 f の中で関数 g が呼び出されたときのみ、g の返り値を表示するようにインタプリタを変更してみる[1]。この命令を使える様にするためには、以下のようにすれば良い。(この実装例は増原ら [9] によるものをベースにしている。)

```
 1 0-1> (exec-at-metalevel (begin
 2        (define state 0)         ; f が何回、再帰的に呼ばれたかを示す変数
 3        (define (add-state! flg) ; state を変化する
 4          (set! state (+ state flg)))
 5        (define (see-state)      ; state を見る
 6          state)
 7        (define original-application eval-application)
 8        (define (eval-cflow exp1 env1 cont1)
 9          (eval-list exp1 env1 (lambda (l)
10            (let ((func1 (car l))        ; f
11                  (func2 (car (cdr l)))) ; g
12              (set! original-application (lambda (exp env cont)
13                (cond ((eq? (car exp) func1)
14                       (add-state! 1)    ; f が呼ばれると state に 1 加える
15                       (eval-list exp env (lambda (l)
16                         (base-apply (car l) (cdr l) env (lambda (m)
17                           (add-state! -1) ; f からぬけたら state から 1 引く
18                           (cont m))))))
19                      ((and (eq? (car exp) func2) ; g が呼び出されて
20                            (> (see-state) 0))    ; state が 0 より大なら
21                       (eval-list exp env (lambda (l)
22                         (base-apply (car l) (cdr l) env (lambda (m)
23                           (write func2) (display " : ") (write m) (newline)
24                           (cont m)))))) ; g の計算結果を出力
25                      (else
26                        (eval-list exp env (lambda (l)
27                          (base-apply (car l) (cdr l) env cont)))))))
28              (cont1 'cflow-changed)))))
29        (set! eval-application ; eval-cflow への dispatch を追加
30              (lambda (exp env cont)
31                (if (eq? (car exp) 'cflow) (eval-cflow (cdr exp) env cont)
32                    (original-application exp env cont))))))
33 0-1: eval-application
```

29 行目以降で eval-application を書き換えて cflow への dispatch を加えている。cflow 文は 8 行目から定義される eval-cflow で解釈実行される。ここで state は f を実行中かを示すフラグである。再帰している場合に対応するため state はカウンタになっており、f に入るたびに state に 1 が足され、f をぬけると 1 が引かれる。そして g に入ったときに $state > 0$ であれば f の中で g が呼ばれたので g の演算結果を出力している。

```
0-2> (cflow 'f 'g)
0-2: cflow-changed
0-3> (define (g x y) (+ x y))
```

[1] g の返り値を表示するだけでなく、そこで任意の式を実行するようにすることも可能であると思われるが、現段階では実装できていない。

```
0-3: g
0-4> (define (f x y z) (* (g x y) z))
0-4: f
0-5> (f 1 2 3)
g : 3        ; f の中で g が呼ばれたので g の結果が出力されている。
0-5: 9
0-6> (g 3 5)
0-6: 8    ; f の中から呼ばれていないのでメッセージは出力されない。
```

3.4 ThisJoinPoint

ThisJoinPoint とは、アドバイスが実行されるときのさまざまな情報を取得するものである。ここではそのもっとも簡単なものとして、現在、実行されている関数名の情報を変数をトレースしているときに取ってこられるようにすることを考える。新しい命令として、`(thisjoinpoint 変数)` というのを導入し、これで変数をトレースするようにするが、同時にその変数がどの関数内で参照されたのかも表示するようにする。

この命令を使える様にするためには、以下のようにすれば良い。

```
0-1> (exec-at-metalevel (begin
       (define func '()) ; 再帰的に呼び出されてきた関数名を表すリスト
       (define original-application eval-application)
       (define (eval-thisjoinpoint exp1 env1 cont1)
         (let ((old-var eval-var) ; オリジナルの eval-var を取っておく
               (var (car exp1)))  ; トレースする変数名
           (set! eval-var (lambda (exp env cont)
             (let ((pair (get exp env)))
               (if (pair? pair)
                   (begin (if (eq? exp var) ; exp が指定した変数だったら
                              (begin (write func) (display ":")     ; 関数名と
                                     (write (cdr pair)) (newline))) ; 変数の値を出力
                          (cont (cdr pair)))
                   (my-error (list 'eval-var: 'unbound 'variable: exp) env cont)))))
           (set! original-application (lambda (exp env cont)
             (eval-list exp env (lambda (l)
               (set! func (cons (car exp) func)) ; 関数名をリストに加える
               (base-apply (car l) (cdr l) env (lambda (m)
                 (set! func (cdr func)) ; 関数から出るとリストから外す
                 (cont m)))))))
           (cont1 'thisjoinpoint-changed)))
       (set! eval-application (lambda (exp env cont) ; eval-thisjoinpoint への dispatch を追加
             (if (eq? (car exp) 'thisjoinpoint) (eval-thisjoinpoint (cdr exp) env cont)
                 (original-application exp env cont))))))
0-1: eval-application
```

ここで func は再帰的に呼び出されてきた関数（ジョインポイントの情報）のリストである。上のプログラムでは、関数に入る度にリストに関数名が加えられ、関数から抜けると関数名がリストから取り外される。そして、変数アクセス時にリストにある関数名を表示している。

以下のように n をトレースすることにすると

```
0-2> (thisjoinpoint n)
```

```
0-2: thisjoinpoint-changed
0-3> (define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
0-3: fac
```

以下のように n が現在いる関数を得ることができる。

```
0-4> (fac 3)
(fac):3
(fac):3
(fac):3
(fac fac):2
(fac fac):2
(fac fac):2
(fac fac fac):1
(fac fac fac):1
(fac fac fac):1
(fac fac fac fac):0
0-4: 6
```

ここでは関数のリストそのものを出力しているので、現在実行中の関数名だけでなく、過去の関数名も出力されている。リストの先頭の要素だけ出力するように変更すれば、現在実行中の関数名だけを出力することもできる。

4 Black の実際の構造

前節では、いくつかのアスペクトが自己反映言語で書けることを見てきたが、このままでは解釈実行されているため非常に遅い。AOP での織り込みに相当するコンパイルを部分評価器を使って行いたい。しかし、インタプリタを部分評価器でコンパイルするためには（ユーザから見えているメタレベルの状態ではなく）Black の構造そのものを知らなければならない。そこで、ここでは Black の実際の構造について説明する。

ここで紹介する Black の構造は、部分評価器を使ったコンパイルの仕組みを作るには必要であるが、一度、そのような仕組みができあがってしまえば、その後、アスペクトを Black 上で実現するだけなら知っている必要はない。また、作ったアスペクトをコンパイルする（織り込む）際にも、基本的には必要ない。しかし、うまくコンパイルができなかった場合、その原因を探ろうとするときには必要である。

Black の構造を理解する上でキーとなるのが、インタプリタが書き換わったかどうかを判断する meta-apply というフック関数と各レベルへ自由に行き来ができる様に、各レベルの環境と継続を保持しているメタ継続 Mcont である。Black のメイン関数である base-eval では、基本的には 2.1 節で示したのと同じように、構文に従って dispatch している。

```
(define (base-eval exp env cont)
  (cond ((number? exp) (cont exp))
        ((symbol? exp) (meta-apply 'eval-var exp env cont))
        ((eq? (car exp) 'if) (meta-apply 'eval-if exp env cont))
        ((eq? (car exp) 'define) (meta-apply 'eval-define exp env cont))
        ((eq? (car exp) 'set!) (meta-apply 'eval-set! exp env cont))
        ((eq? (car exp) 'lambda) (meta-apply 'eval-lambda exp env cont))
        ((eq? (car exp) 'begin) (meta-apply 'eval-begin (cdr exp) env cont))
        ((eq? (car exp) 'exec-at-metalevel) (meta-apply 'eval-EM exp env cont))
        ((eq? (car exp) 'pe) (meta-apply 'eval-pe exp env cont))
        (else (meta-apply 'eval-application exp env cont))))
```

2.1 節でのインタプリタとの違いは、eval-var などの他の関数を呼び出す際に、直接 (eval-var exp env cont) と呼び出すのではなく、(meta-apply 'eval-var exp env cont) と meta-apply 経由で呼び出していることである。これは、

eval-var がユーザによって別の関数に書き換えられているかも知れないので、実際に呼び出す前に eval-var の定義を（メタレベルの環境から）取り出し、その時点での eval-var を呼び出すことで、ユーザによる変更に対応している。

meta-apply は以下の様になっている。処理する関数が書き換わっていれば書き換わった関数で、書き換わっていなければ元の関数で処理を実行する。

```
 1 (define (meta-apply proc-name . operand)
 2   (lambda (Mcont) ; メタ継続
 3     (let* ((meta-env (car (head Mcont)))
 4            (meta-cont (car (cdr (head Mcont))))
 5            (meta-Mcont (tail Mcont))
 6            (operator (cdr (get proc-name meta-env)))) ; proc-name の値を meta-env から
 7       (cond ((procedure? operator)                    ; 取ってくる
 8              (if (pair? (member operator primitive-procedures))
 9                  ((meta-apply 'base-apply operator operand meta-env meta-cont)
10                   meta-Mcont)
11                  ((apply operator operand) ; デフォルトの関数ならそのまま適用
12                   Mcont)))
13             (else ((meta-apply 'base-apply operator operand meta-env meta-cont)
14                    meta-Mcont))))))) ; ユーザ定義の関数に書き換わっていたら、一つ上の
15                                     ; レベルのインタプリタを使って解釈実行する
```

ここに出てくる Mcont はメタ継続と呼ばれ、各レベルの環境と継続を保持しているものである。meta-apply は Mcont の中からメタレベルの環境を取り出し (6 行目)、eval-var などの他のインタプリタ関数が書き換わっているかどうかを調べる (7 行目)。書き換わっていなかったらそのまま実行する[2] が (11 行目)、書き換わっていたら一つ上のレベルのインタプリタを使ってそれを解釈実行している (13 行目)。

Black の実装では、ほとんど全ての関数がメタ継続を受け取り、レベル遷移はメタ継続への操作で実現されている。しかし、レベル遷移を伴わない関数ではメタレベルの情報は必要ないため Mcont は η 簡約されて外見上は見えなくなっている。例えば、上の base-eval の定義は Mcont を使って書けば

```
(define (base-eval exp env cont) (lambda (Mcont)
  (cond ((number? exp) ((cont exp) Mcont))
        ((symbol? exp) ((meta-apply 'eval-var exp env cont) Mcont)
        ...))))
```

となるべきところだが、η 簡約されて Mcont の見えない形になっている。

5　部分評価

5.1　部分評価とは

部分評価 [6] とは、プログラムに部分的な入力を与えて、その入力に最適化されたプログラムを生成する技法である。これによって、汎用的で可読性の高いプログラムを特定の用途に最適化して実行することができる。部分評価の技法は、与えられたプログラムを、計算できる静的（static) 式と、プログラム中に残る動的（dynamic）式に分解し、静的な式のみを実行する。静的な式とは、定数、静的な引数、静的な式の値のみに依存している式である。動的な式は、それ以外、つまり動的な引数や動的な式に依存した式である。例えば、以下のべき乗を計算する式：

```
(define (power m n) (if (= m 0) 1 (* n (power (- m 1) n)))))
```

を、m が静的、n が動的であるとすると、式 (= m 0)、条件分岐、式 (- m 1)、power の再起呼出が静的であり、then 節の値１、式 (* n (power (- m 1) n)（の乗算部分）が動的である。そして、静的な式だけを実行し、その際に実行さ

[2] ここで apply は Scheme にもとからある apply で、base-apply というのは Black で定義されている関数である。base-apply は CPS で書かれ、ユーザ定義の関数の適用などを行なう。

れるべき動的な式を「特化された式」として出力する。関数呼出は、被呼出側が出力した式を呼出側が出力する式の中に埋め込んだものを出力する。例えば、power を m=3 であるとして特化すると、(= m 0)、条件分岐、(- m 1)、power の再起呼出などを実行しつつ、動的な式”(* n ■）”や”1”を出力してゆき（■は再起呼出によってつくられる特化された式が埋め込まれる領域を示す)、”(* n (* n (* n 1)))”という式を得る。

この部分評価を使い、インタプリタを特化する事により Black のコンパイルを行う。

5.2 インタプリタを部分評価する

実行速度の高速化を図るためにインタプリタを部分評価する。インタプリタのメイン関数である base-eval は式と環境（と継続）を受け取るが、このうち式が部分評価時（コンパイル時）にわかっていれば、base-eval が行っている dispatch 等は先に実行してしまうことができる。これを押し進めると、部分評価を使うことでほぼインタプリタ一段分のオーバーヘッドを取り除くことができる。これは部分評価の世界では第 1 二村射影 [4] として知られているものである。

専用のコンパイラを作らずにインタプリタを部分評価するのは、メタレベルがユーザによって自由に書き換えられるためである。コンパイラは、言語の意味が定まって初めて作ることができるが、メタレベルのインタプリタが書き変わる（言語の意味が変わる）ような場合には、あらかじめコンパイラを作っておくことはできない。このような場合、一般には、部分評価を使ってコンパイルする必要がある。なお、ユーザによる変更を制限し、その範囲内でコンパイルできるような仕組みを作ることは可能かも知れないが、ここではより一般的な状況、つまりユーザが任意にインタプリタを書き換え得るという状況を考えている。

Black のコンパイルに使用する部分評価器は [1] によるものである。これは通常の Scheme インタプリタに (pe λ式) という命令が加わったもので、これでλ式を部分評価した結果が得られる。この部分評価器の上に Black を実行し、その上でユーザプログラムを動かす。この部分評価器自体も Scheme で実装されているため、ユーザプログラムは通常の Scheme インタプリタの上で動く pe を実行するインタプリタ（部分評価器）の上で動く Black インタプリタの上で動くことになる。

部分評価は次のように行う。Black における base-eval は pe 命令を受け取ると eval-pe を呼び出すようになっている。eval-pe は以下のように定義されている。

```
 1 (define (eval-pe exp env cont)
 2   (base-eval (car (cdr exp)) env (lambda (x) (lambda (Mcont) ; コンパイル時のメタ継続
 3     (let ((lambda-params        (car (cdr x)))
 4           (lambda-body     (car (cdr (cdr x))))
 5           (lambda-env (car (cdr (cdr (cdr x))))))
 6       ((meta-apply cont ;          ↓環境は未知              実行時のメタ継続↓
 7                     ((pe (lambda (unknown-env) (lambda operand (lambda (Mcont2)
 8                             (let ((meta-env (car (head Mcont2)))
 9                                   (meta-cont (car (cdr (head Mcont2))))
10                                   (meta-Mcont (tail Mcont2)))
11                               ((meta-apply 'eval-begin ; 式を評価
12                                            lambda-body
13                                            (extend unknown-env lambda-params operand)
14                                            meta-cont) Mcont)))))) ; Mcont を使う
15                      lambda-env))
16        Mcont))))))
```

ユーザがコンパイルしたいλ式に対して (pe λ式) と実行すると、コンパイルされるべきλ式が 2 行目で closure になり、3 行目から 6 行目でその中身が取り出される。そして、7 行目から 14 行目までで部分評価を行っている。

部分評価は、基本的にはメタレベルインタプリタを式 (lambda-body) で特化する形で行われる。この際、環境とメタ継続について特別な扱いをしている。7 行目を見ると (pe (lambda (unknown-env) ...)) となっているので、環

境は部分評価中、未知であることがわかる。これはベースレベルのプログラムが将来 set!で書き換えられた場合に対応できるようにするためである。(ただし cons や+などの primitive については大域環境を特別扱いすることで既知としている。)

一方、メタ継続については、部分評価中、既知であるようにしている。上のコードでは、これは 14 行目で実行時のメタ継続である Mcont2 を使うのではなく、コンパイル時のメタ継続である Mcont を使っていることからわかる。これは、メタ継続が未知になってしまうと全く部分評価できなくなってしまうためである。メタ継続は、メタレベルの情報を保持しており、base-eval などインタプリタを構成する関数が eval-var など他のインタプリタ関数を呼び出す際に必ず参照する。メタ継続が未知になってしまうと、他のインタプリタ関数の情報が全くわからない、言い換えればインタプリタが全て未知のもとで部分評価することになり、全く特化が行われないまま部分評価が終了することになる。

5.3 静的なアスペクトと動的なアスペクト

メタ継続を既知として部分評価するというのは、部分評価する上ではやむを得ないことであるが、一方で、これは「一旦、コンパイルされた後は、メタレベルを書き換えても、その影響はコンパイルされたコードには及ばない」ということを意味している。アスペクトの言葉で言い換えれば「一旦、アスペクトを織り込んだ後に、新しいアスペクトを定義しても、その影響は(もとのプログラムを新しいアスペクトがインストールされたもとで再び織り込まない限り)織り込みずみコードには及ばない」ということになる。これはコンパイルされたコードは解釈実行されていない、と考えれば自然であるが、コンパイルされているかされていないかを意識せずに使いたい場合には不便かも知れない。しかし、インタプリタのあらゆる変更を許す、という一般的な枠組のもとでは、なんらかの形でコンパイルされているコードと解釈実行されているコードを区別することは不可欠であり、逆に区別しないのであれば、全くコンパイルはできないものと思われる。

ここで述べたことは、アスペクトが静的であるか、動的であるか、ということに関係する。コンパイルする前はユーザプログラムは解釈実行されているので、実行時に動的にアスペクトが付け加わっても問題ない。ユーザプログラムはそのアスペクトのもとで実行されることになる。しかし、一旦、コンパイルされると、その後に加わったアスペクトはもはやコンパイルされたコードには及ばない。まとめると、本稿で紹介しているシステムは、解釈実行されているコードは動的なアスペクトに対応しているが、コンパイルされたコードは、その時点で静的にわかっているアスペクトにしか対応できない、ということになる。なお、前者は、アスペクトの中で自己反映機能が使われるようなケースにも対応できていると思われるが、現段階ではその実験はできていない。

以上が部分評価を使ったコンパイルの概要と特徴であるが、実際にインタプリタを部分評価するには様々な問題点がある。次節からはその問題点と解決方法について述べる。

5.4 filter

特化をする際、式によっては無限ループに陥ってしまう場合がある。無限ループを防ぐために、インタプリタ内に filter[3] というユーザによる注釈を設ける。if 文の条件文が動的でかつ再帰関数を使っている場合、無限ループになる可能性がある。例えば、x という式(変数)を特化するとする。その際、インタプリタ内では環境テーブルから変数の値を取ってくる以下の get 関数が使われる。

```
(define (get var env)
  '(filter (static? env))
  (if (null? env)
      '()
      (let ((pair (assq var (car env))))
        (if (pair? pair)
        pair
        (get var (cdr env))))))
```

しかし、変数 x の値が入っている環境が動的（未知）である場合、上の get 関数を特化（インライン展開）しようとすると、if 文の条件部分が動的なので、分岐している式を全て特化しようとして、無限ループに陥ってしまう。そこで、filter を使う。filter は'(filter(条件)) と書き、条件を満たさない場合にはその filter を入れた関数は展開せずに関数呼び出しの形で返す。上の get 関数の場合、環境テーブルが静的だったらこの get 関数を特化し、そうでない場合には何もせずに関数呼び出しのまま返す。そうする事によって特化時に無限ループになる事を防ぐ。

全ての再帰関数で無限ループに陥る可能性がある。そこで、再帰変数がわかっている場合にのみ展開するという指示に相当する filter を全ての再帰関数に入れる。この挿入は現在のところ、Black の実装に直接手を加える形で行っている。部分評価の知識がなくても、コンパイラのインライン展開を知っていれば容易に挿入できる類のものである。将来的には、適当なヒューリスティクスを使った自動化を考えている。

5.5 束縛時の改善 (Binding-time Improvement)

部分評価される関数の中に if 文が出てきた時、if 文の条件式が動的な場合、if 文全体は動的になり、各分岐部分を部分評価していく。前述したとおり、base-eval では Mcont を後ろで回しており、各関数にも適用されている。しかし、if 文全体が動的だった場合、Mcont が静的であっても if 文の各分岐部分には適用されない。するとその時点でメタレベルの状況が未知になり、その後、一切コンパイルができなくなってしまう。具体的に見てみよう。eval-if は次のように定義されている。

```
(define (eval-if exp env cont)
  (let ((pred-part (car (cdr exp)))
        (then-part (car (cdr (cdr exp))))
        (else-part (cdr (cdr (cdr exp)))))
    (meta-apply 'base-eval pred-part env (lambda (p)
      (if p (meta-apply 'base-eval then-part env cont)
            (meta-apply 'base-eval (car else-part) env cont))))))
```

この状態では Mcont は η 簡約されて現れてきていないが、Mcont を一部、明示的に書くと次のようになる。

```
(define (eval-if exp env cont)
  (let ((pred-part (car (cdr exp)))
        (then-part (car (cdr (cdr exp))))
        (else-part (cdr (cdr (cdr exp)))))
    (meta-apply 'base-eval pred-part env (lambda (p) (lambda (Mcont)
      ((if p (meta-apply 'base-eval then-part env cont)
             (meta-apply 'base-eval (car else-part) env cont))
       Mcont))))))
```

ここで問題なのは if 文である。今、条件部分を評価した結果である p は実行時になるまでわからない。従って if 文は実行できず、条件分岐が特化結果としてそのまま残る。そのような場合、条件分岐自体は実行できなくとも、then 部分、else 部分はそれぞれ特化して欲しい。しかし、条件分岐の then 部分、else 部分を見てみると、(meta-apply ...) という形をしており、これは Mcont を渡されないとほとんど特化を行えない。

そこで、明示的に Mcont を渡すようにプログラムを書き換える。具体的には以下のようにする。

```
(define (eval-if exp env cont)
  (let ((pred-part (car (cdr exp)))
        (then-part (car (cdr (cdr exp))))
        (else-part (cdr (cdr (cdr exp)))))
    (meta-apply 'base-eval pred-part env (lambda (p) (lambda (Mcont)
      (if p ((meta-apply 'base-eval then-part env cont) Mcont)
            ((meta-apply 'base-eval (car else-part) env cont) Mcont)))))))
```

	コンパイル時間			実行時間	倍率
	eval-application	eval-var	fac		
コンパイルなし	-	-	-	307.3	1.0
メタレベルをコンパイル	16.9	61.4	-	30.1	10.2
ベースレベルもコンパイル	16.9	61.4	15.3	0.87	353.2

表 1: trace 付きの (fac 10) にかかる実行時間（秒）

先ほどとの違いは Mcont を if 文全体に渡すのではなく、then 部分、else 部分それぞれに別々に渡している点である。そうする事により if 文の分岐部分は Mcont が適用された形で部分評価される。

同様の事を例えば eval-and, eval-or などの if 文の条件文が動的になる可能性のある関数にもする。この変更は、部分評価のことをよく知っていなくてはできないが、一度、Black を実装する時にしておけば、その後、ユーザがプログラムを組む際に必要となることはない。

6　部分評価の実験

以下の実験は全て SPARC 750MHz , メモリ 512MB のマシンを使用した。またプログラムの実行に使用した処理系は Petite Scheme 6.0a である。

6.1　変数のトレース

(trace 変数）が使える様にするために書き換えたインタプリタを部分評価する。

```
0-1> (exec-at-metalevel (begin
       (define original-application eval-application)
       (define (eval-trace exp1 env1 cont1)
         (let ((old-var eval-var) ; オリジナルの eval-var をとっておく
               (var (car exp1)))
           (set! eval-var (pe (lambda (exp env cont) ; <===================== pe
                 (if (eq? exp var) ; exp が指定した変数だったら
                     (begin (write (cdr (get exp env))) (newline)) ; その値を表示する
                 (old-var exp env cont)))) ; もとの eval-var の処理を行う
           (cont1 'trace-changed)))
       (set! eval-application ; eval-trace への dispatch を追加
             (pe (lambda (exp env cont) ; <================================== pe
               (if (eq? (car exp) 'trace) (eval-trace (cdr exp) env cont)
                 (original-application exp env cont)))))))
0-1: eval-application
```

3.1 節との違いは pe 命令が 2 ヶ所に入っている事だけである。ここで eval-trace 自体の部分評価を行っていないのは、eval-trace が副作用 (set!) を使用しているため、部分評価できないためである。これは使用している部分評価器の制限である。

上のプログラムを実行すると、まず eval-application がコンパイルされる。そして

```
0-2> (trace n)
0-2: trace-changed
```

を実行すると、その時点で書き換えられた eval-var がコンパイルされる。コンパイルにかかった時間は表 1 に示されている。それぞれ 16.9 秒、61.4 秒である。

コンパイル結果は次の様になっている。(エラー処理部分等を省略しているため短くなっているが、実際の出力は575 行であった。) 多少複雑になっているが、eval-var を CPS にしたものに対して直接出力命令 (write v) (newline) が埋め込まれていることがわかる。結果が CPS になるのは Black のインタプリタが CPS で書かれているためである。また、old-var を実行する部分が複製されているのは、部分評価時に if 文で継続を then 部分と else 部分で複製してより多くの部分評価を行っているからである。またベースレベルの環境を未知としてコンパイルしているので var や get、old-ver などが unbound な場合のエラー処理も入っている。var や old-var は定義されているはずだが、現段階ではこれを消去する方法は考えていない。これは、定義されているはずの関数を調べるのが自明ではないためである。(メタレベルインタプリタをユーザが操作して、さっきまで定義されていた関数が未定義になってしまうような場合も考えられる。) しかし、このエラー処理によるオーバーヘッドを考えると、何らかの方法を考えることが望ましい。

```
(lambda (exp env cont)
  (let ([var-pair (get 'var lambda-env)])
    (if (pair? var-pair)
        (let ([var (cdr var-pair)])
          (if (eq? exp var) ; 変数が var かどうかをチェック
              (let ([get_pair (get 'get lambda-env)])
                (if (pair? get_pair)
                    (let ([get (cdr get_pair)])
                      (base-apply get (list exp env) meta-env
                        (lambda (v-pair)
                          (let ([v (cdr v-pair)])
                            (write v) ; 変数の値を表示
                            (newline)
                            (let ([old-var-pair (get 'old-var lambda-env)])
                              (if (pair? old-var-pair)
                                  (let ([old-var (cdr old-var-pair)])
                                    (base-apply old-var (list exp env cont)
                                                meta-env meta-cont))
                                  エラー)))))) ; old-var が unbound
                    エラー)) ; get が unbound
              (let ([old-var-pair (get 'old-var lambda-env)])
                (if (pair? old-var-pair)
                    (let ([old-var (cdr old-var-pair)])
                      (base-apply old-var (list exp env cont) meta-env meta-cont))
                    エラー)))) ; old-var が unbound
        エラー))) ; var が unbound
```

次のプログラムで、部分評価した場合としなかった場合の実行速度の計測を行った。

```
0-3> (define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
0-3: fac
0-4> (fac 10)
10
...
0-4: 3628800
```

実行時間は表 1 に示されている。コンパイル時間を含めなければ、メタレベルをコンパイルすることで約 10 倍高速化した。コンパイル時間を含めても 3 倍弱の速度向上が見られる。

更に fac をこの変更された eval-var のもとで部分評価してみよう。

```
0-5> (set! fac (pe fac))
0-5: fac
```

このコンパイルには約 15.3 秒かかり、コンパイル結果は以下の様になっている。（実際の出力は 156 行。）

```
(lambda (n)
  (write n) ; n を表示
  (newline)
  (if (= n 0)
      (meta-apply cont 1)
      (begin
        (write n) ; n を表示
        (newline)
        (let ([fac-pair (get 'fac lambda-env)])
          (if (pair? fac-pair)
              (let ([fac (cdr fac-pair)])
                (write n) ; n を表示
                (newline)
                (base-apply fac (list (- n 1)) env
                            (lambda (x) (meta-apply cont (* n x)))))
              エラー))))) ; fac が unbound
0-5> (fac 10)
0-5: 3628800
```

これを見ると、確かに n にアクセスするときに表示するようなコードにコンパイルされている。pe により変数をトレースをするアスペクトが織り込まれていることがわかる。

コンパイルした fac の実行は 0.87 秒で終了し、全くコンパイルしなかった場合と比べると 353 倍の高速化を得ることができた。しかし一方で、ユーザが以下の様に fac の定義の中に n を出力するコードを書き、これを（Black は動かさず、直接、部分評価器の上で）実行すると 0.12 秒になり、更に 7.3 倍速くなる。

```
(define (fac n)
  (if (= (begin (write n) (newline) n) 0)
      1
      (* (begin (write n) (newline) n) (fac (- (begin (write n) (newline) n) 1)))))))
```

これは、このコードには自己反映にするための処理（meta-apply によるフックや Mcont の受渡しなど）が一切、含まれていないためであると思われる。上のコンパイルされたコードにはまだ変数が未定義だった場合のエラー処理などが残っているが、ほぼコンパイルできているととらえれば、ここでの差が自己反映にしているオーバーヘッドであると考えられる。

ここでの例は、部分評価がアドバイスを完全に織り込んでしまうことを示している。しかし、織り込まれるコードが大きく、かつ織り込まれる場所がたくさんある場合には、コードの爆発が問題となる場合がある。また、Hilsdale ら [5] はオブジェクト指向言語のアクセス権の問題から、織り込みまでは行なわない、というアプローチをとっている。上の例では、織り込まれるコードは小さいため、織り込んでしまっても問題はないが、織り込みたくない場合はアドバイスのコードが展開されないように filter を入れれば良い。

6.2 before / after

前節と同様、pe 命令を入れてコンパイルする。

```
0-1> (exec-at-metalevel (begin
       (define original-application eval-application)
       (define (eval-before exp1 env1 cont1) ; before 文を処理する関数
```

	コンパイル時間				実行時間	倍率
	eval-application	before	after	fac		
コンパイルなし	-	-	-	-	484.5	1.0
メタレベルをコンパイル	31.4	379.1	54.4	-	62.2	7.8
ベースレベルもコンパイル	31.4	379.1	54.4	49.3	3.9	124.2

表 2: before/after 付きの (fac 10) にかかる実行時間（秒）

```
    (eval-list exp1 env1 (lambda (l)
      (let ((func (car l))         ; 捕まえる関数名
            (ex1 (car (cdr l))))   ; 実行する関数
        (set! original-application ; original-application を書き換える
              (pe (lambda (exp env cont) ; <=========================== pe
                (eval-list exp env (pe (lambda (l) ; <================== pe
                  (if (eq? (car exp) func)        ; func への呼び出しなら
                      (apply ex1 (cdr l))) ; ex1 を実行
                  (base-apply (car l) (cdr l) env cont))))))))
      (cont1 'before-changed))))
  (define (eval-after exp1 env1 cont1) ; after 文を処理する関数
    (eval-list exp1 env1 (lambda (l)
      (let ((old-application original-application)
            (func (car l))         ; 捕まえる関数名
            (ex1 (car (cdr l))))   ; 実行する関数
        (set! original-application ; original-application を書き換える
              (pe (lambda (exp env cont) ; <=========================== pe
                (old-application exp env (pe (lambda (m) ; <============ pe
                  (if (eq? (car exp) func) ; func への呼び出しなら
                      (ex1 m))             ; ex1 を実行
                  (cont m))))))))
      (cont1 'after-changed))))
  (set! eval-application ; eval-before/after への dispatch を追加
        (pe (lambda (exp env cont) ; <================================= pe
          (cond ((eq? (car exp) 'before) (eval-before (cdr exp) env cont))
                ((eq? (car exp) 'after) (eval-after (cdr exp) env cont))
                (else (original-application exp env cont))))))))
0-1: eval-application
```

3.2 節との違いは、5ヶ所に pe が入っていることだけである。trace のときと同様、eval-before/after は中で副作用を使っているため部分評価していないが、eval-before/after の中で書き換えられる original-application についてはコンパイルしてから書き換えている。上のプログラムのコンパイル時間、及び以下のふたつの命令の実行（コンパイル）時間を表 2 に示す。

```
0-2> (before 'fac (lambda (n) (display "enter : ") (write n) (newline)))
0-2: before-changed
0-3> (after 'fac (lambda (result) (display "result: ") (write result) (newline)))
0-3: after-changed
```

さらに (fac 10) をコンパイルせずに実行した場合と、コンパイルして実行した場合の結果も表 2 に示す。fac をコンパイルした結果は、前節と同様、 fac への再帰呼び出しの前後で write 文が挿入された（織り込まれた）形となって

いる。

6.3 cflow

次に cflow のコンパイルを行なう。使っている部分評価器が副作用命令を扱えないため、副作用によって書き変わる変数 `state` へのアクセスを部分評価から見えなくする必要がある。そのために `add-state!` と `see-state` に対して filter を加え、それらの関数が部分評価時に展開されないようにしている。その上で、以下のように 7ヶ所に pe を入れる。

```
0-1> (exec-at-metalevel (begin
       (define state 0)         ; f が何回、再帰的に呼ばれたかを示す変数
       (define (add-state! flg) ; state を変化する
         '(filter #f '(flg))    ; <========================================== filter
         (set! state (+ state flg)))
       (define (see-state)      ; state を見る
         '(filter #f '())       ; <========================================== filter
         state)
       (define original-application eval-application)
       (define (eval-cflow exp1 env1 cont1)
         (eval-list exp1 env1 (lambda (l)
           (let ((func1 (car l))        ; f
                 (func2 (car (cdr l)))) ; g
             (set! original-application (pe (lambda (exp env cont) ; <========== pe
               (cond ((eq? (car exp) func1)
                      (add-state! 1)    ; f が呼ばれると state に 1 加える
                      (eval-list exp env (pe (lambda (l) ; <==================== pe
                        (base-apply (car l) (cdr l) env (pe (lambda (m) ; <===== pe
                          (add-state! -1) ; f からぬけたら state から 1 引く
                          (cont m))))))))
                     ((and (eq? (car exp) func2) ; g が呼び出されて
                           (> (see-state) 0))    ; state が 0 より大なら
                      (eval-list exp env (pe (lambda (l) ; <==================== pe
                        (base-apply (car l) (cdr l) env (pe (lambda (m) ; <===== pe
                          (write func2) (display " : ") (write m) (newline)
                          (cont m)))))))) ; g の計算結果を出力
                     (else
                      (eval-list exp env (pe (lambda (l) ; <=================== pe
                        (base-apply (car l) (cdr l) env cont)))))))))
             (cont1 'cflow-changed)))))
       (set! eval-application ; eval-cflow への dispatch を追加
             (pe (lambda (exp env cont) ; <====================================== pe
               (if (eq? (car exp) 'cflow) (eval-cflow (cdr exp) env cont)
                   (original-application exp env cont)))))))
0-1: eval-application
```

コンパイル時間は表 3 に示す。

前節と同様、cflow 命令を実行すると、そこでコンパイルがされる。

```
0-2> (cflow 'f 'g)
```

	コンパイル時間				実行時間			
	eval-application	cflow	f	g	f	倍率	g	倍率
コンパイルなし	-	-	-	-	37.2	1.0	16.9	1.0
メタレベルをコンパイル	16.5	2769.2	-	-	5.4	6.9	1.7	9.9
ベースレベルもコンパイル	16.5	2769.2	23.5	3.9	3.2	11.6	0.23	73.5

表 3: cflow 付きの (f 1 2 3) と (g 3 5) にかかる実行時間（秒）

```
0-2: cflow-changed
```

このコンパイルには 2769.2 秒と、とても長い時間がかかる。この原因は解明中だが、条件分岐を両方、別々に（継続を複製して）部分評価しているため、分岐がたくさんあると多くの複製が起こり、その部分評価に時間がかかっている模様である。

3.3 節と同様に、関数 f と g の定義を行ない、それらをコンパイルせずに実行した場合と、コンパイルしてから実行した場合の結果を表 3 に示す。

6.4 ThisJoinPoint

最後に、ThisJoinPoint をコンパイルする。前節と同様、現在、使っている部分評価は、副作用命令を扱えないため、`func` への副作用が部分評価から見えない形で扱われるようにすべく、関数名を見る `see-func` と関数名をリストに入れる `set-func!`を定義し、さらに、それらが展開されないように指示する filter を加えている。その上で、次のように 5ヶ所に pe を入れる。

```
0-1> (exec-at-metalevel (begin
       (define func '())  ; 再帰的に呼び出されてきた関数名を表すリスト
       (define (see-func) ; 関数名を見る
         '(filter #f '()) ; filter
         func)
       (define (set-func! name) ; func に現在の関数名を入れる
         '(filter #f '(name))   ; filter
         (set! func name))
       (define original-application eval-application)
       (define (eval-thisjoinpoint exp1 env1 cont1)
         (let ((old-var eval-var) ; オリジナルの eval-var を取っておく
               (var (car exp1)))  ; トレースする変数名
           (set! eval-var (pe (lambda (exp env cont) ; <======================== pe
             (let ((pair (get exp env)))
               (if (pair? pair)
                   (begin (if (eq? exp var) ; exp が指定した変数だったら
                               (begin (write (see-func)) (display ":") ; 関数名と
                                      (write (cdr pair)) (newline))) ; 変数の値を出力
                          (cont (cdr pair)))
                   (my-error (list 'eval-var: 'unbound 'variable: exp) env cont))))))
           (set! original-application (pe (lambda (exp env cont) ; <============ pe
             (eval-list exp env (pe (lambda (l) ; <============================ pe
               (set-func! (cons (car exp) (see-func))) ; 関数名をリストに加える
               (base-apply (car l) (cdr l) env (pe (lambda (m) ; <============== pe
                 (set-func! (cdr (see-func))) ; 関数から出るとリストから外す
```

	コンパイル時間			実行時間	倍率
	eval-application	thisjoinpoint	fac		
コンパイルなし	-	-	-	646.6	1.0
メタレベルをコンパイル	19.9	669.8	-	100.4	6.4
ベースレベルもコンパイル	19.9	669.8	62.7	66.7	9.7

表 4: ThisJoinPoint 付きの (fac 10) にかかる実行時間（秒）

```
            (cont m))))))))))
        (cont1 'thisjoinpoint-changed)))
    (set! eval-application (pe (lambda (exp env cont) ; <=================== pe
          (if (eq? (car exp) 'thisjoinpoint) (eval-thisjoinpoint (cdr exp) env cont)
              (original-application exp env cont)))))))
0-1: eval-application
```

コンパイル時間、及び (fac 10) の実行時間を表 4 に示す。ここでの速度向上はこれまでの例と比べて小さめだが、それでも 10 倍程度の向上になっている。ThisJoinPoint は、部分評価を使ってコンパイルができる、と言われているが、この結果はそれを間接的にサポートするものとなっている。

7　関連研究

自己反映言語の部分評価を使ったコンパイルは増原ら [10] が並列オブジェクト指向の自己反映言語 ABCL/1 に対して行なっている。本研究での仕事は、それを関数型の自己反映言語に対して行なったものととらえることができる。関数型言語では副作用が（あまり）ないため扱いやすく、オブジェクト指向言語の部分評価よりも比較的、容易である。そのため、増原らは、ABCL/1 のうちメソッド実行など扱いやすい部分を取り出してコンパイルしていたが、本研究では Black の実装をそのまま部分評価することができている。

アスペクトの部分評価を使ったコンパイルも増原ら [9] によってなされている。ここでの研究は、そのアプローチをベースレベル言語もメタレベル言語も関数型であるような世界で追実験した格好になっている。本稿では、部分評価（コンパイル）の起動自体もインタプリタに組み込んでいるため、コンパイルも含めた意味論を議論する土台になると期待される。

Sereni ら [11] は静的な解析を行なうことで、実行時のオーバーヘッドを削減する研究を行なっている。本研究では、静的な解析は一切、行なっていないので、部分評価によって彼らが得たのと同じ結果を得ることはできない。しかし、彼らの解析を組み込み、そこで得られた知識を使って、より賢く部分評価するという方向性は考えられる。

8　現状と今後の課題

本稿では、アスペクト指向言語のコンパイル手法を探る目的で、自己反映言語上にいくつかのアスペクトを実装し、部分評価によるコンパイルを行なった。現段階ではまだ基本的なアスペクトしか実装できていないが、ユーザ定義のインタプリタのもとで部分評価を行なうという非常に一般的な枠組の中でのコンパイルができるようになってきており、今後の発展によっては AOP のコンパイル方法のひとつになりうることを示している。

部分評価を使った自己反映言語のコンパイルという観点では、副作用の問題は依然として残っている。現在は、巧妙に set!などの命令が部分評価と衝突しないようにコンパイルしているが、これを一般のユーザに強いるのは難しい。変更可能な場所を何らかの方法で特定した上で、副作用命令も扱えるような枠組を構築する必要があるだろう。

副作用の問題が解決できれば、部分評価を使ったコンパイルに対する問題は解決しつつあるように感じられる。部分評価の無限ループを回避する問題があるが、こちらは比較的、単純な方法である程度の自動化ができるものと考えている。

今後は、織り込み方法が確立されていないアスペクトなども実装し、部分評価の観点からコンパイル法を探って

いきたいと考えている。また、査読者の方から指摘して頂いたが、アスペクト指向言語では難しいとされている分割コンパイルなどが、この枠組でどう扱えるかについても検討していきたい。

謝辞　東京大学の増原英彦氏、及び査読者の方々から多くの有益なコメントを頂きました。

参考文献

[1] Asai, K. "Integrating Partial Evaluators into Interpreters," In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation (LNCS 2196)*, pp. 126–145 (September 2001).

[2] Asai, K., S. Matsuoka, and A. Yonezawa "Duplication and Partial Evaluation — For a Better Understanding of Reflective Languages —," *Lisp and Symbolic Computation*, Vol. 9, Nos. 2/3, pp. 203–241, Kluwer Academic Publishers (May/June 1996).

[3] Consel, C. "A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages," *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pp. 145–154 (June 1993).

[4] Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971), reprinted in *Higher-Order and Symbolic Computation*, Vol. 12, No. 4, pp. 381–391, Kluwer Academic Publishers (December 1999).

[5] Hilsdale, E., and J. Hugunin "Advice Weaving in AspectJ," *International Conference on Aspect-Oriented Software Development (AOSD 2004)*, to appear (March 2004).

[6] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[7] Kelsey, R., W. Clinger, and J. Rees (editors) "Revised^5 Report on the Algorithmic Language Scheme," *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, pp. 7–105, Kluwer Academic Publishers (August 1998).

[8] Kiczales, G. J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J-M. Loingtier, and J. Irwin "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP97)*, pp. 220–242 (1997).

[9] Masuhara, H., G. Kiczales, and C. Dutchyn "A Compilation and Optimization Model for Aspect-Oriented Programs," *Proceedings of the 12th International Conference on Compiler Construction (CC2003), LNCS 2622*, pp. 46–60 (April 2003).

[10] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 300–315, (October 1995).

[11] Sereni, D., S., and O. de Moor "Static Analysis of Aspects," *International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp. 30–39 (March 2003).

[12] Smith, B. C. "Reflection and Semantics in Lisp," *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 23–35 (January 1984).

[13] Sullivan, G. T. "Aspect-Oriented Programming Using Reflection and Metaobject Protocols," *Communications of the ACM*, Vol. 44, No. 10, pp. 95–97 (October 2001).