MetaOCaml を使った自己反映言語のコンパイル

浅井 健一

お茶の水女子大学

asai@is.ocha.ac.jp

概 要　自己反映言語とは、言語の意味（セマンティクス）がプログラムに公開されており、プログラム内部から言語の意味にアクセスしたり意味を変更したりすることが許されているような言語のことである。一般に自己反映言語は柔軟性が高く、言語によってはいかなる言語仕様の変更・拡張も可能だったりするが、その効率的な実行は極めて難しい。基本的には部分評価のような技術を使って現在のインタプリタ定義に従ってコンパイルするしかないが、コンパイル結果をその場で使用しようと思うと、コンパイル結果をソースコードの形で出力する従来の部分評価器を使うのには難がある。ここでは、代わりにステージ化言語 MetaOCaml を使い、部分評価と同様の最適化を行いつつ、MetaOCaml の持つ「その場で生成されたプログラムを実行する機能」を使って自己反映言語のコンパイルを試みる。

1　はじめに

自己反映言語とは、言語の意味（セマンティクス）がプログラムに公開されており、プログラム内部から言語の意味にアクセスしたり意味を変更したりすることが許されているような言語のことである。自己反映言語は、その柔軟性と強力な表現力から多くの研究がなされている。オブジェクト指向言語 CLOS [9] では、オブジェクトの内部情報がメタオブジェクトプロトコルとして公開され、プログラム中から使用可能であった。また、このアプローチを特化したアスペクト指向言語 [10] なども提案され、その効率的な実装法 [11] も研究されている。Java にはリフレクション API が備わっており、言語の内部情報にアクセスすることを許している。これらの仕事は、実用的に便利な機能に特化してそれを効率的に実装できるような枠組みを提供しようというのが大まかな方向性だったが、あらかじめ必要な機能を列挙するのは難しい。そのため、言語のインタプリタをユーザに示すという一般的な方向性の研究 [14] も出てきている。

一方、自己反映言語の一般的な設計手法や実装法についての理論的な研究もなされている。3-LISP [12] に始まり、Brown [6, 15] や Blond [4] といった言語では、ユーザプログラムがメタサーキュラインタプリタの無限のタワーによって解釈実行されているモデルが示され、その内部データにアクセスできるようになっている。さらに、著者らは Black [3] という言語を提案し、そこでは言語の意味を与えるインタプリタをユーザプログラムが変更できるようになっている。インタプリタは言語の操作的意味を与えていると思えば、まさに言語の意味を動的に変更可能な言語ということができる。

しかし、自己反映言語は強力である反面、効率的な実装は簡単ではない。インタプリタを任意に変えられるという状態では、基本的に部分評価器 [8] を使ってインタプリタを特化するしかないが、従来の部分評価器は部分評価結果がソースコードとして得られてしまう。これでは、部分評価結果をそのまま現在の実行環境の中で使うのは難しい。著者はインタプリタ中に部分評価器を組み込んで、部分評価結果をインタプリタ環境に統合したような処理系を作成した [1] が、自己反映言語のコンパイルには至っていない。これは、部分評価器の制御が簡単ではないためである。

本論文では、部分評価の細かな制御と部分評価結果の利用を同時に満たす方法として MetaOCaml [13] を使った自己反映言語のコンパイルを試みる。MetaOCaml は、部分評価器のような自動の特

化はできないが、ユーザがきちんとアノーテーションを付ければ部分評価と同様の効果を実現できる。さらに MetaOCaml には run と呼ばれる機能があり、これを使うと生成したコードをその場の環境で使用することができる。これらの機能を使って自己反映言語をコンパイルするとどのようになるか、その模様を報告するとともに、その問題点についても考察する。

以下、2 節では本論文で扱う自己反映言語 Black を紹介し、3 節で MetaOCaml を使ったコンパイル手法を示す。4 節では、いくつかのコンパイルの例を示す。5 節で、紹介したコンパイル手法について考察し、6 節でまとめる。

2 自己反映言語 Black

この節では、自己反映言語 Black とその実装を概観する。Black には継続渡し形式 (CPS) のもの [3]、直接形式のもの [2] などいくつかのバリエーションが存在するが、ここで扱う Black は最初の CPS のものに基づいている。Black の設計と実装の詳細については [3] を参照されたい。

2.1 ユーザから見た Black

Black は Scheme の拡張で、自己反映機能を使わない限りは普通の Scheme と同じである。Black では、ユーザプログラムはインタプリタによって実行されているように見えている。このユーザプログラムが実行されているレベルをベースレベルと呼ぶ。ベースレベルのユーザプログラムは、Scheme で書かれた（ように見えている）インタプリタによって実行される。このインタプリタが実行されているレベルをメタレベルと呼ぶ。（さらに、このインタプリタ自身も、もうひとつ上のレベルで動いているインタプリタによって実行されているように見えており、これが無限に続く格好になっている。しかし、本論文では 2 レベルを超えるインタプリタは考えない。）

ここまでだと Black はインタプリタ実行されている Scheme と同じだが、Black には普通の Scheme に加えて `EM` という命令が追加されている。これは `exec-at-metalevel` の頭文字をとった命令で、引数をメタレベルで実行する命令である。

メタレベルには、ベースレベルのプログラムを実行するインタプリタが定義され実行されている。`EM` を使うと、このインタプリタが定義されているレベルに入ることができ、さらにその中身を見たり、変更したり、拡張したりすることができるようになっている。メタレベルのインタプリタは言語の意味を定めているととらえることができるので、Black は実質的に言語の意味を動的に変更することができるような言語ととらえることができる。

メタレベルのインタプリタを変更・拡張するためには、そもそもメタレベルのインタプリタがどのように定義されているのかを知らなくてはならない。CPS の Black においては、メタレベルインタプリタは CPS で書かれている。その代表的な部分を図 1 に示す。これは標準的な CPS インタプリタである。このインタプリタが、メタレベルインタプリタを変更・拡張する際の方法、すなわち API を定めている。

2.2 Black の実装

Black のプログラムは、あたかも図 1 に示されるインタプリタの無限のタワーによって実行されているように見えているが、実際にはそうではない。実際の Black のインタプリタは、ふたつの方法を使って著しく効率を落とすことなくインタプリタの可変性を保つように作られている。

解釈実行されるコードと直接実行されるコードの区別　Black では、解釈実行されるコードと直接実行されるコードを区別している。解釈実行されるコードというのは、ベースレベルのユーザプログラムのように「ひとつ上のレベルのインタプリタによって解釈実行されているようなコード」のことである。完全に自己反映的な言語なら、すべてのコードが解釈実行されるべきところだが、そ

```
(define (base-eval exp env cont)
  (cond ((number? exp)           (cont exp))
        ((symbol? exp)           (eval-var exp env cont))
        ((eq? (car exp) 'quote)  (eval-quote exp env cont))
        ((eq? (car exp) 'if)     (eval-if exp env cont))
        ...
        ((eq? (car exp) 'lambda) (eval-lambda exp env cont))
        (else (eval-application exp env cont))))
(define (eval-var exp env cont)
  (let ((pair (get exp env)))
    (if (pair? pair)
        (cont (cdr pair))
        (my-error (list 'eval-var: 'unbound 'variable: exp) env cont))))
(define (eval-quote exp env cont) (cont (car (cdr exp))))
(define lambda-tag (cons 'lambda 'tag))
(define (eval-lambda exp env cont)
  (let ((lambda-body (car (cdr (cdr exp))))
        (lambda-params (car (cdr exp))))
     (cont (list lambda-tag lambda-params lambda-body env))))
(define (eval-list exp env cont)
  (if (null? exp)
      (cont '())
      (base-eval (car exp) env (lambda (val1)
        (eval-list (cdr exp) env (lambda (val2)
          (cont (cons val1 val2))))))))
(define (eval-application exp env cont)
  (eval-list exp env
             (lambda (l) (base-apply (car l) (cdr l) env cont))))
(define (base-apply operator operand env cont)
  (cond ((procedure? operator)
         (cont (scheme-apply operator operand)))
        ((and (pair? operator)
              (eq? (car operator) lambda-tag))
         (let ((lambda-params        (car (cdr operator)))
               (lambda-body     (car (cdr (cdr operator))))
               (lambda-env (car (cdr (cdr (cdr operator))))))
           (base-eval lambda-body
                      (extend lambda-env lambda-params operand)
                      cont)))
        (else (my-error (list 'Not 'a 'function: operator) env cont))))
```

図 1. ユーザから見たメタレベルインタプリタ（抜粋）

うするとメタレベルインタプリタを解釈実行するためにメタメタレベルのインタプリタが必要になり、無限個のインタプリタが必要になってしまう。そうではなく、Black では直接実行されるコードを導入することで、有限の資源で実行を可能にしている。具体的には、デフォルトのメタレベルインタプリタは全て直接実行されているとしている。これは、次のことを意味する。

- メタレベルインタプリタが直接実行されているので、有限の資源で実行可能である。
- メタレベルインタプリタは Scheme で実装される必要はない。実際、本論文で示す Black はメタレベルインタプリタが OCaml で実装されている。(つまり、メタレベルインタプリタは OCaml で書かれた Scheme インタプリタになっている。)
- ベースレベルプログラムは、直接実行されるインタプリタによって実行されている。従って、最初の段階では 2 段以上のインタプリタによって実行されていることはなく、極端に効率が悪いわけではない。
- メタレベルインタプリタは直接実行されているので、メタメタレベルインタプリタを変更しても、メタレベルインタプリタの挙動を変化させることはできない。

フックの使用　直接実行されるコードを導入することで、有限の資源で Black を実装できるようになるが、メタレベルインタプリタを機械語で全て完全に直接実行してしまうとメタレベルインタプリタを変更できなくなってしまう。そこで、関数の再定義を可能にするため、直接実行されているメタレベルインタプリタでは、関数呼び出しを行う部分にフックが入っており、そこで呼び出される関数が変更されているかのチェックを行うようになっている。

例えば、メタレベルインタプリタのメイン関数 `base-eval` に対応する OCaml の直接実行されている関数は以下のようになる。

```
(* base_eval : value_t -> mcont_t -> unit *)
let base_eval args =
  let (exp, env, cont) = decode_args3 args in
  match exp with
    Number (n) -> apply_cont cont (Number (n))
  | Symbol (x) -> meta_apply "eval-var" args
  | Pair ({contents=e1}, {contents=e2}) ->
      match e1 with
          Symbol ("quote") -> meta_apply "eval-quote" args
        | Symbol ("if") -> meta_apply "eval-if" args
        | ...
        | Symbol ("lambda") -> meta_apply "eval-lambda" args
        | _ -> meta_apply "eval-application" (encode_args4 (e1, e2, env, cont))
```

ここで `decode_args`n, `encode_args`n は長さ n の Black のリストと OCaml の n 個組を対応させる関数である。また、出てくる構成子は以下のように定義されている。

```
type value_t =
    Number of int                                  (* 数字 *)
  | Symbol of string                               (* シンボル *)
  | Nil                                            (* 空リスト *)
  | Pair of value_t ref * value_t ref              (*(書き換え可能な)ペア *)
  | Primitive of string                            (* + などの primitive 関数 *)
  | Lambda of value_t * value_t * value_t          (* ユーザ定義関数 *)
```

```
  | Evalfun of (value_t -> mcont_t -> unit)      (* 直接実行コード *)
and mcont_t = MCont of (value_t * value_t) * mcont_t Lazy.t (* メタ継続 *)
```

このデータ定義はほぼ自明であろう。Lambda の 3 つの引数はそれぞれパラメタ、本体、環境を value_t 型で表現したものである。Evalfun, mcont_t については後述する。

上記の base_eval を見ると、インタプリタの他の関数を呼ぶ際、直接、呼び出すのではなく meta_apply というフックを通して呼んでいるのがわかる。例えば、入力 exp がシンボル（変数）だった場合、シンボルを解釈実行する関数 eval-var を直接、呼び出すのではなく、meta_apply を通して呼び出している。meta_apply は以下のように定義されている。

```
(* meta_apply : string -> value_t -> mcont_t -> unit *)
let rec meta_apply proc_name operand =
  shift_up (fun (meta_env, meta_cont) ->
    let operator = cdr (get (Symbol proc_name) meta_env) in
    match operator with
        Evalfun (f) -> shift_down (f operand) meta_env meta_cont
      | _ -> meta_apply "base-apply"
                        (encode_args4 (operator, operand, meta_env, meta_cont)))
```

関数 meta_apply は、shift_up でレベルをひとつ上がってメタレベルの環境 meta_env を得て、そこから呼び出される関数 proc_name の値をとってくる。もし、呼び出される値が直接実行可能な関数 Evalfun であれば、元のレベルに戻ってそのまま直接、呼び出す。一方、直接実行可能な関数でなければ、それはユーザによってインタプリタが変更されたことを意味しているので、（メタレベルの）関数呼び出しを行う関数 base-apply を呼び出して、変更されたインタプリタの解釈実行を行う。この場合、ベースレベルプログラムは、メタメタレベルインタプリタによって解釈実行される（ユーザ定義の）メタレベルインタプリタのもとで実行されることになる。（同様に apply_cont も、引数の cont が直接実行可能かどうかをチェックしてから呼び出す関数である。）

このように、他の関数を呼び出す際、常にメタレベルの環境を見て、変更されているかどうかをチェックしながらメタレベルインタプリタは実行される。これによってインタプリタ関数の変更が実現されることになる。一方、これはデフォルトのメタレベルインタプリタは通常のインタプリタに比べてフックが入っている分、遅いことを意味している。

レベルの上下は、次のふたつの関数を使って行われる。

```
(* shift_up : (value_t * value_t -> mcont_t -> unit) -> mcont_t -> unit *)
let shift_up code = fun (MCont ((meta_env, meta_cont), meta_mcont)) ->
  code (meta_env, meta_cont) (Lazy.force meta_mcont)

(* shift_down : (mcont_t -> unit) -> value_t -> value_t -> mcont_t -> unit *)
let shift_down code env cont = fun mcont ->
  code (MCont ((env, cont), lazy mcont))
```

レベルはメタ継続を使って管理されている。メタ継続は、各レベルの環境と継続を持った無限の長さの lazy リストである。レベル上下は、単にメタ継続をひとつずらしているだけである。

3 MetaOCaml を使ったコンパイル

インタプリタをユーザプログラムが既知のもとで部分評価すると、コンパイルしたのと同じことになることが知られている [7]。MetaOCaml を使っても同様のことが可能である。本節では、それを自己反映言語に対して行う。

3.1 ユーザ定義関数の表現

デフォルトの状態では、ユーザ定義の関数を処理する eval_lambda は以下のように実装されている。(図 1 の対応する関数を完全に実装した形にはなっていないが、完全に対応させることは容易である。)

```
(* eval_lambda : value_t -> mcont_t -> unit *)
let eval_lambda args =
  let (exps, env, cont) = decode_args3 args in
  let (params, body) =
    match cdr exps with
       Pair (params, {contents=Pair (body, {contents=Nil})}) -> (!params, !body)
     | _ -> raise (Error "Bad arguments for lambda") in
  apply_cont cont (Lambda (params, body, env))
```

この状態では、ユーザ定義関数は Lambda という単なるデータ構造になっている。これでは、たとえユーザが定義した関数のパラメタや本体（eval_lambda 中の params や body）がわかっていても、ユーザ定義の関数に対応する直接実行されるコードを得ることはできない。そこで Lambda というデータ構造を関数化し、「コンパイルされるユーザ定義関数」を意味する clambda という special form を導入し、clambda を以下のように解釈実行することにする。

```
(* eval_clambda : value_t -> mcont_t -> unit *)
let eval_clambda args =
  let (exps, env, cont) = decode_args3 args in
  let (params, body) =
    match cdr exps with
       Pair (params, {contents=Pair (body, {contents=Nil})}) -> (!params, !body)
     | _ -> raise (Error "Bad arguments for clambda") in
  apply_cont cont
    (Evalfun (fun args -> shift_up (fun (meta_env, meta_cont) ->
                let new_env = extend env params args in
                meta_apply "base-eval"
                           (encode_args3 (body, new_env, meta_cont)))))
```

eval_clambda は、Lambda というデータ構造ではなく、直接実行可能な Evalfun を返している。直接実行可能とは言ってもそれは表向きだけで、その中身は単にメタレベルのインタプリタを呼び出し、ユーザ定義関数を解釈実行しているだけなので、このままでは実行速度は Lambda を使っているのとほぼ同じである。しかし、このようにすると、Evalfun の中に OCaml の関数が存在するので、それを意味の等しいより速いものに置き換えることができれば高速化することができる。

3.2 メタ継続の既知化

本論文では、MetaOCaml を使って部分評価を行う。[1] 具体的には、前節に示した eval_clambda の中の Evalfun の引数の関数 fun args -> ... を params と body が既知のもとで部分評価する。この関数では、引数 args はまだ受け取っていないので未知、また shift_up の定義を展開するとメタ継続もまだ受け取っていないので、meta_env, meta_cont も未知である。この状況で部分評価を行えれば良いが、このままでは meta_env が未知のため meta_apply "base-eval" で呼び

[1] 使用したのは BER MetaOCaml N101 である。

出されるインタプリタが未知になってしまう。インタプリタの形が未知のままでは何も最適化をすることができないので、この時点で未知のメタ継続（コンパイルされる関数が呼び出される時点でのメタ継続）を現在のメタ継続（コンパイル時点でのメタ継続）に置き換えて部分評価を行う。

このメタ継続の置き換えは「これ以後、たとえメタレベルのインタプリタが変更されたとしても、ここでコンパイルされた関数は影響を受けない」ことを意味する。つまり、コンパイルして直接実行されるコードに変換するというのは、その時点でのメタレベルインタプリタを固定し、それに従って最適化するので、以後、メタレベルインタプリタの変更の影響を受けなくすることである。

本論文では、さらにコンパイル時のメタレベルインタプリタはデフォルトのインタプリタでユーザによる変更は行われていないと仮定する。この仮定は満足のいくものではなく、将来的にはユーザ定義のインタプリタのもとでコンパイルしていきたいが、5.3 節でみるように現時点ではそれは難しい。

3.3 インタプリタのステージ化

以上の考察をふまえ、MetaOCaml の命令を使って当該の関数（eval_clambda の中の Evalfun の引数の関数）を以下のようにステージ化する。

```
!. .< fun args -> shift_up (fun (meta_env, meta_cont) ->
      .~(let new_env = extend2 env params (Code .<args>.) in
         base_eval2 (encode_args3 (body, new_env, eta_up .<meta_cont>.)))))>.
```

ここに示したステージ化した関数は、もとの関数と比べて以下の点で変更されている。

ステージ化アノーテーションの追加　ステージ化した関数には、MetaOCaml のステージ化アノーテーションが追加されている。.< と >. は MetaOCaml のコードを生成する命令で、これらで囲まれた部分は実行されずにコードがそのまま生成される。ただし、その中の .~ に続く部分はコード生成時に実行され、結果として得られたコードがそこに埋め込まれる。従って、ここに可能な計算を全て行うようなプログラムを入れておけば、部分評価をしたコードを得ることができる。先頭の !. は、このようにして得られたコードをコンパイルし、実行可能な値に変換する命令である。

全体として、既知の body を実行する部分をできる限り実行し、その結果を埋め込んだ最適化したコードを生成し、それを実行可能な値に変換している。

meta_apply の使用中止　メタレベルインタプリタの形が未知のままでは部分評価できないので、meta_apply の使用をやめて、代わりに直接、インタプリタ関数を呼び出すような形になっている。同様の効果が、既知のメタ継続を meta_apply に渡すことでも実現できると思われるが、簡単のためにここではそうしていない。

値へのコードの追加　上のプログラム自体からは観察しにくいが、ステージ化するために値 value_t の定義を拡張して、以下のようなコードを追加している。これは、ステージ化したコードや、コードを扱う関数（継続）を（encode_args 等の引数に渡すなど）他の通常の値と同列で扱う必要があるためである。

```
type value_t =
    ...
  | EvalfunC of (value_t code -> (mcont_t -> unit) code)  (* 部分評価時の継続 *)
  | Code of value_t code                                  (* コード *)
```

EvalfunC は部分評価時の（static な）継続、Code はコードである。ステージ化した関数にでてくる eta_up は、未知の継続 meta_cont を部分評価時の継続に持ち上げる関数で、以下のように定義される。

```
(* eta_up : value_t code -> value_t *)
let eta_up cont_code =
  (EvalfunC (fun v -> .<apply_cont .~cont_code .~v>.))
```

これは部分評価時に未知の値をあたかも既知の値であるかのように扱うための 2 レベルの η 拡張と呼ばれるトリック [5] で、通常の部分評価を行う際にも使われる。

ステージ化されたインタプリタの使用　ステージ化した関数では、もとの base_eval ではなくステージ化した関数である base_eval2 を使っている。これがステージ化の要点である。base_eval2 は以下のように定義される。

```
(* base_eval2 : value_t -> (mcont_t -> unit) code *)
let rec base_eval2 args =
  let (exp, env, cont) = decode_args3 args in
  match exp with
    Number (n) -> apply_cont2 cont .<exp>.
  | Symbol (x) -> eval_var2 args
  | Pair ({contents=e1}, {contents=e2}) ->
      match e1 with
          Symbol ("quote") -> eval_quote2 args
        | Symbol ("if") -> eval_if2 args
        | ...
        | Symbol ("lambda") -> eval_clambda2 args
        | Symbol ("clambda") -> eval_clambda2 args
        | _ -> eval_application2 (encode_args4 (e1, e2, env, cont))
```

式 exp の内容に従って場合分けをしている部分は全て部分評価時に実行されるようになっていることがわかる。例えば、exp が数字かどうかの判定は部分評価時に行われ、その結果、数字だった場合には数字のコードが返される。ここで apply_cont2 は value_t 型に埋め込まれている継続 cont にコードを渡す関数で、以下のように定義されている。

```
(* apply_cont2 : value_t -> value_t code -> (mcont_t -> unit) code *)
let apply_cont2 (EvalfunC (f)) operand = f operand
```

exp の中にさらにユーザ定義の関数が書かれているときには、それが lambda であるか clambda であるかに関わらず eval_clambda2 を呼び出してコンパイルするようになっている。

exp が関数呼び出しだった場合には、以下の関数が呼ばれる。[2]

```
let eval_application2 args =
  let (f, exps, env, cont) = decode_args4 args in
  base_eval2 (encode_args3 (f, env, (EvalfunC (fun v ->
  eval_list2 (encode_args3 (exps, env, (EvalfunC (fun vs ->
  .<meta_apply "base-apply"
               (encode_args4 (.~v, .~vs, env, .~(eta_down cont)))>.))))))))
```

[2] let 文で定義しているが、実際には base_eval2 と相互再帰で定義されている。

eval_list2 は式の列を評価して各評価結果のリストを返す関数（をステージ化したもの）である。base_eval2 や eval_list2 が返すものはコードで、部分評価時には値がわからないので、関数呼び出しをすることはできない。そのため、各部分式の評価が終了したら「実行時に関数呼び出しを行うようなコード」を出力している。（最後の 2 行は、ステージ化していないデフォルトの base-apply を呼び出すようなコードである。）

値呼びの言語では、関数呼び出しのコードを残す際には、その順序に気をつけなくてはならない。上の eval_application2 では、最後の 2 行で関数呼び出しのコードを残しているが、全体が CPS で書かれておりこの関数呼び出しを行った後に cont を行う形になっているため実行順序は保たれている。ここで、eta_down は eta_up の逆で、部分評価時の継続をコードに変換する関数で、以下のように定義されている。

```
(* eta_down : value_t -> value_t code *)
let eta_down cont = .<Evalfun (fun x -> .~(apply_cont2 cont .<x>.))>.
```

これは手動で let-insertion を行っていることに相当する。ここで新たに作られる変数 x は、MetaOCaml が自動で他とは重ならない新しい変数名をとってくれるので、同じ変数名がぶつかってしまうことを心配する必要はない。

環境の扱い　最後に環境の扱いについて触れておこう。環境を大きくする関数 extend についてもステージ化した関数である extend2 が使われている。環境は、フレーム（連想リスト）のリストで表されており、extend2 は以下のように定義されている。

```
(* make_pairs2 : value_t -> value_t -> value_t *)
let rec make_pairs2 params args = match (params, args) with
    (Nil, _) -> Nil
  | (Symbol (s), _) -> cons (cons params args) Nil
  | (Pair ({ contents = param }, { contents = params }),
     Pair ({ contents = arg }, { contents = args})) ->
      cons (cons param arg)
           (make_pairs params args)
  | (Pair ({ contents = param }, { contents = params }), Code (c)) ->
      cons (cons param (Code .<car .~c>.))
           (make_pairs2 params (Code .<cdr .~c>.))
  | _ -> raise (Error "Bad arguments to make_pairs2")

(* extend2 : value_t -> value_t -> value_t -> value_t *)
let extend2 env params args =
  cons (make_pairs2 params args) env
```

ステージ化されるにあたって付け加わっているのは make_pairs2 の 4 つ目のケースのみである。引数 args が未知の場合、その中から必要な場所をとってくるコードを環境に入れている。

4　例

本節では、コンパイルの例として、ベースレベルのプログラムをコンパイルした結果と、ユーザによって変更されたメタレベルのインタプリタをコンパイルしたときのコンパイルの様子と実行速度について述べる。本節の例では、これまでに説明して来なかった define や begin などいろいろな special form を使うが、これらをサポートするのは容易である。

4.1 ベースレベルプログラムのコンパイル

(clambda (x) 3) のコンパイル結果は、

```
fun args_1 -> shift_up (fun (meta_env_2, meta_cont_3) ->
  apply_cont meta_cont_3 exp)
```

となる。ここで shift_up, apply_cont, exp は MetaOCaml のステージ間永続機能 (cross-stage persistence; CSP) により部分評価時の値がコードに埋め込まれているものである。特に、exp は部分評価時の base_eval2 の中に出てくる exp のことで、上の場合は Number (3) が入っている。base_eval2 による場合分けは部分評価時に実行され、3 を直接返すコードになっていることがわかる。

同様に (clambda (x) x) のコンパイル結果は

```
fun args_4 -> shift_up (fun (meta_env_5, meta_cont_6) ->
  apply_cont meta_cont_6 (car args_4))
```

となる。引数のひとつ目をとってくるコードを生成できている。

(clambda (x) (+ x 1)) をコンパイルすると以下が得られる。

```
fun args_7 -> shift_up (fun (meta_env_8, meta_cont_9) ->
  meta_apply "base-apply"
    (encode_args4 (v, (cons (car args_7) (cons exp nil)), env,
                   (Evalfun (fun x_10 -> apply_cont meta_cont_9 x_10)))))
```

ここで v は + を環境から引いてきた結果が入っている変数、exp は 1 が入っている変数である。CSP により結果のコードは見にくいが、+ を [x, 1] という引数に適用するような CPS のコードとなっている。

これ以上、複雑な関数のコンパイル結果は CSP のため読むのは困難である。代わりに、以下のように定義したフィボナッチ関数の実行速度をコンパイル前後で比べてみた。

```
(define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
```

コンパイル前は (fib 16) の実行に約 2.5 秒かかったが、これを

```
(set! fib (clambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
```

のようにコンパイルしてから実行すると (fib 16) の実行が約 1.4 秒となり、速度向上が見られた。コンパイル結果は、関数呼び出しのたびに meta_apply "base-apply" が実行されるので高速とは言いがたいが、それでも単純な解釈部分がコンパイルされ多少の速度向上を得られている。

4.2 メタレベルインタプリタのコンパイル

前節の例は、自己反映言語でなくても可能なコンパイルである。この節では、自己反映言語特有のコンパイルとして、メタレベルのインタプリタを変更する例を示す。

自己反映機能を使ったひとつの応用として変数のトレースをあげられる。メタレベルインタプリタでは、変数アクセスがインタプリタ内の 1 ヶ所で記述されているので変数のトレースをとりやすい。ここでは関数 fib 内の変数 n が何回アクセスされたかを数えるように、メタレベルインタプリタを変更してみる。具体的には、以下のようなプログラムを実行する。

```
(EM (begin (define counter 0)
           (define old-eval-var eval-var)
           (set! eval-var (lambda (e r k)
                            (if (eq? e 'n)
                              (set! counter (+ counter 1))
                              0)
                            (old-eval-var e r k)))))
```

まず、EM でメタレベルに上がり、そこに counter を用意する。次に、インタプリタ中の変数を処理する関数 eval-var を保存しておいた上で、eval-var を変更する。新しい eval-var では、アクセスされた変数名が n であるかを調べ、n であったら counter の値を 1 増やしている。

このような変更をインタプリタに対して行った上で、コンパイルしていない fib を使って (fib 16) を実行すると 18.1 秒とかなりの時間がかかるようになった。そこで、eval-var を次のようにコンパイルした。

```
(EM (set! eval-var (clambda (e r k)
                     (if (eq? e 'n)
                       (set! counter (+ (lift counter) 1))
                       0)
                     (old-eval-var e r k))))
```

ここで 3 行目に出てくる lift というのは、変数を引数にとり、その変数の値を部分評価時にはとって来ずに、代わりに変数にアクセスするようなコードにコンパイルする、という意味である。本論文のコンパイルは、コンパイル時の環境の値を使ってコンパイルされる。これは、インタプリタ自身が（メタレベルの）環境に入っているため、環境の値を見ないことにはコンパイルできないからである。しかし、上の counter の例のように「コンパイル時の counter の値」ではなく実行時の値を使わないと意味がない場合もある。そのために lift という特殊な命令を用意して、変数アクセスを制御している。これについては 5.1 節で議論する。

上のように eval-var をコンパイルした上で（コンパイルしていない fib を使って）(fib 16) を実行すると 4.3 秒とそれなりの高速化ができた。参考までに、デフォルトのインタプリタのもとで

```
(define counter 0)
(define fib (lambda (n)
              (if (< (begin (set! counter (+ counter 1)) n) 2)
                (begin (set! counter (+ counter 1)) n)
                (+ (fib (- (begin (set! counter (+ counter 1)) n) 1))
                   (fib (- (begin (set! counter (+ counter 1)) n) 2))))))
```

というように手動で n にアクセスするたびに counter の値を更新するようにしたプログラムを実行してみると (fib 16) に 5.2 秒かかったので、eval-var のコンパイル結果は、そう悪くはないと思われる。

5 議論

この節では、本論文で紹介した MetaOCaml を使ったコンパイルについて考察する。

5.1 副作用の影響

本論文で示した Black のインタプリタは、`set!` のような命令を許し、それは内部では環境への破壊的代入によって実装されている。そして、コンパイル時には環境がこの先、変わらないという仮定をおいて、コンパイル時点での値を使用してしまっている。コンパイルをするということは、その時点での言語の意味（＝メタレベルインタプリタ）を固定して、それを使って最適化するということなので、コンパイル時の環境の値を使うのは自然のように見えるが、前節の `lift` の使用にみられるように場合によってはその時点での値を使って欲しくないこともあるなど、必ずしもこれが最適の方法ではなさそうである。

一方で、これ以外に何かできるかというとそれも難しい。最初に思いつくのは、インタプリタの関数を格納している環境は既知にする（したがって、インタプリタの形は部分評価時に既知となる）が、コンパイルされるプログラムの変数を格納している環境は未知にすることである。しかし、環境は通常の Black のペアを使って実装されているので、一概に「環境は未知にする」と言っても、あちこちで使われているペアのうちどの部分を未知にするのかは明らかではない。例えば、インタプリタ関数の引数は全て `encode` されるが、これらも皆ペアで実装されている。これらを未知にしてしまっては、ほとんど何も部分評価できなくなる。結局、部分評価する際に「この部分は環境として使っているので、その中の binding を表す部分のみは未知にする」など現在、使われている特定のインタプリタにおける環境の実装法についての知識が必要になる。これは使われているペアがコンパイルされる方法の違いで 2 種類あることを意味するが、そのような違いはメタレベルインタプリタ（つまり言語の意味）には現れておらず、言語の意味をユーザに見せるという自己反映言語の趣旨からはずれてしまう。

そもそもメタレベルインタプリタの変更を許可しないような pure な自己反映言語を設計すれば、このような問題は生じないが、それではできることが限られる。変更を許可するが、その実装には破壊的代入は使わずにストアを引き回す形にするという方法もあるが、そうするとメタレベルインタプリタもそのような形になり、従来の破壊的代入を使ったインタプリタを扱うことはできなくなる。このように上手にコンパイルしようとすると、いずれにしろ ad hoc になることは避けられそうにないので、現在の実装では `lift` という明示的に値を未知にするというオペレータを用意し、それ以外は既知とするという方法をとった。

5.2 効率的なコードと簡明なインタプリタのトレードオフ

現在のコンパイルでは、出てくるコードの質は良いとは言えない。しかし、これをより効率的なコードにしようとするのも一筋縄ではない。例えば、加算などの primitive 演算をコンパイルした場合は、`meta_apply "base-apply"` への呼び出しではなく、加算のコードそのものにコンパイルしてしまいたい。しかし、これをするためには `+` を実行した結果の値が必要である。（Black では（普通の Scheme と同様）primitive 演算子は再定義可能であることに注意されたい。`+` をインライン展開するためには、`+` の実行結果が加算演算子であることを知る必要がある。）MetaOCaml のアノーテーションでは、演算子の場合には値の中身を使い、そう出なければコードとして残すといったことを書くことはできないので、このような場合分けをすることはできない。（このようなことをしたければオンラインの部分評価器を使う必要がある。）

primitive 演算は変更不可能という仮定をおき、インタプリタをそのように作れば primitive 演算をインライン展開することはできるだろう。つまり、変更不可能な環境と変更可能な環境の両方を引き回す形でインタプリタを書き、変更不可能な環境については束縛された値まで使ってコード生成を行えば良い。しかし、それにはメタレベルインタプリタをそのような形で書き直さなければならない。

結局、現在のコンパイル結果が十分でないのはメタレベルインタプリタがあまりに一般的すぎるからである。より効率的なコードを出したければ、コンパイル時により多くの情報を得られるよう

にメタレベルインタプリタを書き換えれば良い。しかし、そうするとメタレベルインタプリタはどんどん複雑になり、それが果たして求めているものなのかは検討が必要である。

5.3 多段のコンパイル

本論文で示したコンパイルは、いずれもデフォルトのインタプリタのもとでのコンパイルであった。しかし、自己反映言語をフルに使うには、まずユーザによって変更されたメタレベルインタプリタをコンパイルし、次にその変更されたインタプリタのもとでユーザプログラムをコンパイルしたい。前節では、前者はある程度、可能であることを示したが、後者は現状では残念ながら難しい。その理由は、コンパイルするためにはステージ化されたインタプリタが必要だからである。

現在のコンパイルは、人間が手動でインタプリタをステージ化したものを用意し、それを使っている。コンパイルされたユーザ定義のインタプリタのもとでコンパイルするためには、前者をステージ化したものが必要である。しかし、それをどのようにして得たら良いのかは全く明らかではない。多段のステージ化を用いるとできるかも知れないし、コンパイラ自身をステージ化するといったアイディアが使えるかも知れないが、いずれにしてもさらなる研究が必要である。

ユーザ定義のインタプリタをコンパイルせずに、ベースレベルのプログラムをコンパイルするというのは考えられる。メタレベルインタプリタが変更された状態なので、メタメタレベルインタプリタを使いながら、それをベースレベルプログラムについて部分評価するのである。今のところ試してはいないが、コンパイル時間がかかると予想されること以外には問題はなさそうである。これも今後の課題である。（コンパイル時間は、本論文で示した程度のコンパイルなら、いずれも無視できる程度である。）

5.4 束縛時解析

本論文ではステージ化アノーテーションを手で書いてコンパイルを行ったが、ここでやっていることは従来のオフライン部分評価器の束縛時解析と同じである。MetaOCaml のステージ化アノーテーションは柔軟でいろいろなことが書けるが、本論文の目的を考えるとむしろアノーテーションを手動で書きたくはなく、自動で行えるのが望ましい。これを自動で行えれば、多段のコンパイルのところで触れた問題も自然と解決する。従来の部分評価と比べた MetaOCaml の利点のひとつは生成したコードをその場で実行して使うことができることだが、そこに部分評価で培われた束縛時解析の技術を導入して自動でアノーテーションをつけるようなことができるようになると、自己反映言語のコンパイルにはインパクトがありそうである。

6 おわりに

本論文では、MetaOCaml を使って自己反映言語 Black のコンパイルを行った。MetaOCaml のステージ化アノーテーションを使うことで、部分評価を使ったコンパイルがある程度できている。さらに MetaOCaml の「その場でコードを実行可能な値にして使用する」機能を使うことでそのときの状況に従ったコンパイルができるようになった。一方で、一般的なメタレベルインタプリタを使ってそのまま効率的なコードを生成するのは簡単ではないこともわかった。メタレベルインタプリタのわかりやすさや一般性と効率的なコードの生成の間のトレードオフをどう扱うべきかは今後の課題である。

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. "Reflection in Direct Style," *Proceedings of the Tenth Conference on Generative Programming and Component Engineering (GPCE '11)*, pp. 97–106 (October 2011).

[3] 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).

[4] Danvy, O., and K. Malmkjær "Intensions and Extensions in a Reflective Tower," *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pp. 327–341 (July 1988).

[5] Danvy, O., K. Malmkjær, and J. Palsberg "The Essence of Eta-Expansion in Partial Evaluation," *Lisp and Symbolic Computation*, Vol. 8, No. 3, pp. 209–227, Kluwer Academic Publishers (1995).

[6] Friedman, D. P., and M. Wand "Reification: Reflection without Metaphysics," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 348–355 (August 1984).

[7] 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).

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

[9] Kiczales, G., J. des Rivières, and D. G. Bobrow *The Art of the Metaobject Protocol*, Cambridge: MIT Press (1991).

[10] 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 (ECOOP'97)*, pp. 220–242 (1997).

[11] 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).

[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] Taha, W. "A Gentle Introduction to Multi-stage Programming," In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation (LNCS 3016)*, pp. 30–50 (2004).

[14] Verwaest, T., C. Bruni, D. Gurtner, A. Lienhard, and O. Niestrasz "PINOCCHIO: Bringing Reflection to Life with First-Class Interpreters," *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '10)*, pp. 774–789, (October 2010).

[15] Wand, M., and D. P. Friedman "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, pp. 298–307 (August 1986).