Interpreter without Interpretive Overheads

住井

1998 年 11 月 19 日

要旨

static binding にしたがう変数を持った言語に対し、 naive なインタプリタから efficient な「コンパイラ」を得る systematic な方法について述べる。同様の目的を達成する方法としては二村 projection が有名だが、今回の方法は一般的な partial evaluator を必要としないだけでなく、二村 projection では除去できない、環境を操作するオーバーヘッドも除去できる。

具体的な例として、 untyped λ-calculus のインタプリタを ML で記述し、そこから「untyped λ-calculus から ML へのコンパイラ」を導出する。それだけではつまらないので、もう一つの例として、 π-calculus についてもまったく同様のことを行ってみる。

1 はじめに

Type-Directed Partial Evaluation のほうでいろいろと考えていて思いついた話です。当初は何も知らなかったのですが、いろいろと調べてみたら、テクニカルには higher-order abstract syntax [3] と shortcut deforestation [4] の応用に当たるようです。ただ、それらを組み合わせてインタプリタに適用する、という話は見当たらなかったので、 novelty はあるかもしれません。[1]

2 naive なインタプリタ

まず、「普通に」インタプリタを書いて、どういう点で効率が悪いかを考えてみる。

一般にインタプリタを書くときは、 target language の abstract syntax tree を、 meta language において何らかの形で表現する。たとえば、 ML で λ-calculus のインタプリタを書くならば、 naive には

```
type var = string
```

のように λ 変数を ML の文字列で表し、

```
infix @@
infix ++

datatype exp =
    % of int              (* i *)
  | $ of var              (* x *)
  | \ of var * exp        (* lambda x . e *)
  | @@ of exp * exp       (* e1 e2 *)
  | ++ of exp * exp       (* e1 + e2 *)
  | IF of exp * exp * exp (* if e1 > 0 then e2 else e3 *)
```

のように、 λ 式を ML の datatype で表す。この表現だと、たとえば $(\lambda x.\lambda y.x+y)$ 3 7 という λ 式は、 ML では

[1] 他の人が同じことをすでに考えている / これから思いつく可能性は大いにあると思いますが。

```
- \("x", \("y", $"x" ++ $"y")) @@ %3 @@ %7;
val it = \ ("x",\ ("y",$ "x" ++ $ "y")) @@ % 3 @@ % 7 : exp
```

のように書ける。

式の表現を決めたら、次はそれを評価する関数を書く。α 変換や β 簡約といった規則をストレートに実装すると効率が悪過ぎるので、普通は環境や closure の概念を導入して、「λ 式と環境を受け取ると、その式をその環境で評価した結果の値を返す」ようにする。ここでは、値は整数または closure で、環境は変数から値への finite map で、 closure は関数と環境の組なので、 ML では

```
datatype value =
    Num of int
  | Closure of closure
withtype env = (var * value) list
and closure = var * exp * env
```

のように表現できる。また、環境に対する操作として、変数の値を環境に加える “update” と、変数の値を環境から引く “lookup” を以下のように定義する。

```
fun update (x, e) env = (x, e) :: env

fun lookup x ((n, v) :: rest) =
    if x = n
        then v
    else lookup x rest
```

これらを用いると、評価を行う関数は次のように書ける。

```
fun eval (%n) env = Num(n)
  | eval ($x) env = lookup x env
  | eval (\(x, e1)) env = Closure(x, e1, env)
  | eval (e1 @@ e2) env =
    let
        val Closure(x, e3, env0) = eval e1 env
        val v = eval e2 env
    in
        eval e3 (update (x, v) env0)
    end
  | eval (e1 ++ e2) env =
    let
        val Num(v1) = eval e1 env
        val Num(v2) = eval e2 env
    in
        Num(v1 + v2)
    end
  | eval (IF(e1, e2, e3)) env =
    (case eval e1 env of
         Num(0) => eval e3 env
       | Num(_) => eval e2 env)
```

たとえば、先の $(\lambda x.\lambda y.x+y)\ 3\ 7$ は

```
- eval (\("x", \("y", $"x" ++ $"y")) @@ %3 @@ %7) [];
val it = Num 10 : value
```

のように評価できる。

以上のようにインタプリタを書くと、（当然だが）プログラムを実行するときの速度が、コンパイラに比べて非常に遅い。たとえば、harp の sml/nj-110.0.3 で、このインタプリタを使って $ack(3,7)$ を計算すると、約 18 秒もかかる。ちなみに、まったく同じ λ 式を chez で評価すると、約 1 秒しかかからない。

遅さの主な原因は、いわゆる “interpretive overhead” にあると思われる。特に

- `lookup` が環境から変数の値を引くオーバーヘッド
- `eval` が λ 式をパターン・マッチングにより deconstruct するオーバーヘッド

の二つが大きいと思われる。

λ 式の evaluator に限らず、static binding にしたがう変数を持つ言語のインタプリタを naive に書くと、常に同様のオーバーヘッドが生じる。

3 higher-order abstract syntax

そこで、まずは一番目のオーバーヘッドを取り除く。環境をリストではなく、ハッシュなどの別の構造で表すことも考えられるが、実はそれよりももっと簡単で速い方法がある。

λ-calculus 変数が、ML の変数と同様に static binding にしたがうことに注目して、λ-calculus の変数・環境・closure として、ML の変数・環境・closure を利用する。

たとえば、先の $(\lambda x.\lambda y.x+y)$ 3 7 という λ 式の例だと、今までは

```
\("x", \("y", $"x" ++ $"y")) @@ %3 @@ %7
```

と表していたのを

```
\(fn x => \(fn y => $x ++ $y)) @@ %3 @@ %7
```

のように表すようにする。一般には、今まで `\("x", ...)` と表していた λ 式を `\(fn x => ...)` と表し、今まで `$"x"` と表していた λ 式を `$x` と表すようにする。

abstraction は ML の abstraction で表現するので、`exp` の定義は以下のようになる。

```
datatype exp =
    % of int
  | $ of exp
  | @@ of exp * exp
  | ++ of exp * exp
  | IF of exp * exp * exp
  | \ of exp -> exp
```

`\` の型が `var * exp` から `exp -> exp` になった点が以前と異なる（変数に ML の変数を使っているので、実は `$` は不要なのだが）。

また、closure も ML の closure で表現するので、`value` の定義は

```
datatype value =
    Num of int
  | Closure of value -> value
```

のようになる。

一般に、このような abstract syntax の表現を higher-order abstract syntax [3] というらしい。論文 [3] では、項の操作につきものの α 変換や side condition が不要になる、という利点しか述べていないのだが、eval において closure や環境を操作しなくて良い（meta language がやってくれる）ので楽で速い、という利点のほうがここでは大きい。

higher-abstract syntax を用いると、 abstraction の評価にちょっと工夫がいる。`exp -> exp` という型の与えられた値から、`value -> value` という型の値を 「自然に」作らないといけない。そこで、 exp に`Embed : value -> exp` という constructor を付け加えて、`value` を `exp` に「埋め込む」。そうすると、`f : exp -> exp` を与えられれば、`eval o f o Embed : value -> value` が得られる（下図）。ただし、`Embed(v)` の評価は、単に `eval(Embed(v)) = v` とする。

```
          f
  exp    -->    exp
  ^              |
  |Embed         |eval
  |              v
value   -->  value
  eval o f o Embed
```

同様の trick は、「関数の引数の位置」に recursive な出現を持つ data type に対する、一般的な catamorphism に適用できる。 [1]

以上をまとめると、インタプリタの全体は図 1 のようになる。使い方は

```
- eval (\(fn x => \(fn y => $x ++ $y)) @@ %3 @@ %7);
val it = Num 10 : value
```

のようになる。 $ack(3,7)$ にかかる時間は、約 18 秒から約 8 秒に改善する。

4 shortcut deforestation

さらに、二番目のオーバーヘッドを取り除くことを試みる。はじめの

```
- eval (\("x", \("y", $"x" ++ $"y")) @@ %3 @@ %7) [];
val it = Num 10 : value
```

という評価や、上の

```
- eval (\(fn x => \(fn y => $x ++ $y)) @@ %3 @@ %7);
val it = Num 10 : value
```

という評価は、

1. 値を求めたい λ 式の abstract syntax tree を construct する
2. その abstract syntax tree を destruct しながら値を求めていく

という計算を行っている。

ところが、 λ-calculus では「expression の全体の意味 (denotation) は、その subexpression の意味から決定される」という原理が成り立っているおり、`eval` もそういう形で書かれている。 formal にいうと、 λ-calculus の denotational interpreter である`eval` という関数は、 λ-calculus のプログラム（λ 式）を表現する `exp` という datatype に対する catamorphism になっている。同様のことは、 λ-calculus に限らず、大抵の言語についていえる。 [2]

そこで、「exp を construct してから、 destruct しながら値を求めていく」かわりに、最初から値を求めていく。つまり、 exp の constructor と eval を「合成」してしまう。たとえば、 application については、`@@` という constructor を

```
datatype value =
    Num of int
  | Closure of value -> value

infix @@
infix ++

datatype exp =
    % of int
  | $ of exp
  | @@ of exp * exp
  | ++ of exp * exp
  | IF of exp * exp * exp
  | \ of exp -> exp
  | Embed of value

fun eval (%n) = Num(n)
  | eval ($x) = eval x
  | eval (e1 @@ e2) =
    let
        val Closure(f) = eval e1
        val v = eval e2
    in
        f v
    end
  | eval (e1 ++ e2) =
    let
        val (Num(v1)) = eval e1
        val (Num(v2)) = eval e2
    in
        Num(v1 + v2)
    end
  | eval (IF(e1, e2, e3)) =
    (case eval e1 of
         Num(0) => eval e3
       | Num(_) => eval e2)
  | eval (\f) =
    Closure(eval o f o Embed)
  | eval (Embed(v)) = v
```

図 1: higher-order abstract syntax を用いたインタプリタ

```
fun (e1 @@ e2) env =
    let
        val Closure(x, e3, env0) = e1 env
        val v = e2 env
    in
        e3 (update (x, v) env0)
    end
```

という関数で置き換える。

一般に、何らかの datatype を construct してから catamorphism で destruct するというプログラムに対して、その datatype の constructor と catamorphism を「合成」してしまう、という変換を shortcut deforestation [4] というらしい。

ただ、 ML だとすべての関数が strict になるので、 termination に関して問題が生じる。この場合だと、式を与えるだけで評価が行われてしまうので、何も工夫しないと conditional expression (`IF`) で、 then の部分と else の部分が両方とも評価されてしまう。そこで、`IF` のように評価が遅延されるべき式は、`IF(`*`condition`*`, fn () => `*`then-part`*`, fn () => `*`else-part`* のように表現する。ちなみに、 Haskell などの lazy な言語を使えば、この場合は大丈夫である。[2]

以上をまとめると、インタプリタの全体は図 2 のようになり、

```
- \(fn x => \(fn y => $x ++ $y)) @@ %3 @@ %7;
val it = Num 10 : value
```

のように動く。 $ack(3,7)$ にかかる時間は、約 4 秒に縮まる。

これを “runtime system” として利用すれば、普通の abstract syntax tree として表された λ 式を、

```
\(fn x => \(fn y => $x ++ $y)) @@ %3 @@ %7
```

のように pretty print するだけで、 λ-calculus から ML への「コンパイラ」になる。

ちなみに、理論的には higher-order abstract syntax と shortcut deforestation は orthogonal なのだが、実際には `eval` に対して higher-order abstract syntax を用いずに shortcut deforestation を行っても、約 18 秒なのが約 17 秒になるだけで、ほとんど速くならない。 shortcut deforestation だけを行うと、 λ 式は（たとえ整数であろうと）`env -> value` という型の関数で表されるので、 ML にとって効率が悪いからだと思われる。

5 π-calculus の場合

λ-calculus だけではつまらないので、同様のことを簡単な π-calculus のインタプリタ（図 3）に対しても行ってみた。書き換えのしかたは本当にまったく同様なので、詳しい過程は省略するが、結果を図 4 に示す。

$pfib(28)$ を実行するのにかかる時間をはかってみると、 naive に書いたインタプリタでは約 22 秒だったのが、higher-order abstract syntax を使うと約 13 秒になり、さらに shortcut deforestation を行うと約 5 秒になった。

5.1 他の応用

今回の方法は、別にインタプリタでないと使えないわけではなく、 static binding にしたがう変数を持った式に対する、一般の catamorphism を計算するのに利用できる（type inference や proof checking など）。

実は、事前に計算したい catamorphism が与えられていなくても、似たようなことができる。たとえば、 higher-order abstract syntax を使って表現した λ 式に対して、図 5 のようにのように catamorphism を計算する一般的な関数を定義すると、`cata` もまた catamorphism なので、 shortcut deforestation を行うことができて、

[2] 高野先生によると、lazy で pure な functional language でも、一般には微妙な問題があるそうですが。

```
datatype value =
    Num of int
  | Closure of value -> value

infix @@
infix ++

fun %n = Num(n)
fun $x = x
fun (e1 @@ e2) =
    let
        val Closure(f) = e1
        val v = e2
    in
        f v
    end
fun (e1 ++ e2) =
    let
        val Num(v1) = e1
        val Num(v2) = e2
    in
        Num(v1 + v2)
    end
fun IF(e1, e2_, e3_) =
    (case e1 of
         Num(0) => e3_ ()
       | Num(_) => e2_ ())
fun \f = Closure(f)
```

図 2: higher-order abstract syntax と shortcut deforestation を適用したインタプリタ

```
type var = string

datatype exp =
    EInt of int
  | EAdd of exp * exp
  | EVar of var
and value =
    VInt of int
  | VChan of queue ref
and queue =
    QOut of value list list
  | QIn of receiver list
and proc =
    POut of (exp * exp list) (* e![f1, ..., fn] *)
  | PIn of (exp * var list * proc) (* e?[x1, ..., xn].P *)
  | PPar of proc * proc (* P | Q *)
  | PNew of (var * proc) (* new x in P *)
  | PRIn of (exp * var list * proc) (* e?*[x1, ..., xn].P *)
  | PIf of (exp * proc * proc) (* if e>0 then P else Q *)
withtype env = (var * value) list
and receiver = bool (* whether replicated or not *) * var list * proc * env

fun lookup x ((y, v) :: env) = if x = y then v else lookup x env
  | lookup x [] = (print ("undefined variable: " ^ x ^ "\n"); raise Match)

fun update (xs, vs) env = (ListPair.zip (xs, vs)) @ env

fun eval env (EInt(i)) = VInt(i)
  | eval env (EAdd(e, f)) =
    let
        val VInt(i) = eval env e
        val VInt(j) = eval env f
    in
        VInt(i + j)
    end
  | eval env (EVar(x)) = lookup x env

fun newchan () = VChan(ref (QOut([])))

fun send q vs =
    (case ! q of
         QOut(vss) => q := QOut(vss @ [vs])
       | QIn([]) => q := QOut([vs])
       | QIn((r as (rep, xs, p, env)) :: rs) =>
             (q := QIn(if rep then rs @ [r] else rs);
              exec (update (xs, vs) env) p))
and recv q (r as (rep, xs, p, env)) =
    (case ! q of
         QIn(rs) => q := QIn(rs @ [r])
       | QOut([]) => q := QIn([r])
       | QOut(vs :: vss) =>
             (q := QOut(vss);
              if rep
                  then recv q r
              else ();
              exec (update (xs, vs) env) p))
and exec env (POut(e, fs)) =
    let
        val VChan(q) = eval env e
        val vs = map (eval env) fs
    in
        send q vs
    end
  | exec env (PIn(e, xs, p)) =
    let
        val VChan(q) = eval env e
    in
        recv q (false, xs, p, env)
    end
  | exec env (PPar(p1, p2)) = (exec env p1; exec env p2)
  | exec env (PNew(x, p)) = exec ((x, newchan ()) :: env) p
  | exec env (PRIn(e, xs, p)) =
    let
        val VChan(q) = eval env e
    in
        recv q (true, xs, p, env)
    end
  | exec env (PIf(e, p1, p2)) =
    let
        val VInt(i) = eval env e
    in
        exec env (if i > 0 then p1 else p2)
    end
```

図 3: π-calculus の naive なインタプリタ

```
datatype value =
    VInt of int
  | VChan of queue ref
and queue =
    QOut of value list list
  | QIn of receiver list
withtype receiver = bool * (value list -> unit)

fun EInt(i) = VInt(i)
fun EAdd(e, f) =
    let
        val VInt(i) = e
        val VInt(j) = f
    in
        VInt(i + j)
    end

fun newchan () = VChan(ref (QOut([])))

fun send q vs =
    (case ! q of
         QOut(vss) => q := QOut(vss @ [vs])
       | QIn([]) => q := QOut([vs])
       | QIn((r as (rep, func)) :: rs) =>
             (q := QIn(if rep then rs @ [r] else rs);
              func vs))
fun recv q (r as (rep, func)) =
    (case ! q of
         QIn(rs) => q := QIn(rs @ [r])
       | QOut([]) => q := QIn([r])
       | QOut(vs :: vss) =>
             (q := QOut(vss);
              if rep
                  then recv q r
              else ();
              func vs))
fun POut(VChan(q), vs) = send q vs
fun PIn(VChan(q), func) = recv q (false, func)
fun PPar(p1, p2) = (p1; p2)
fun PNew(func) = func (newchan ())
fun PRIn(VChan(q), func) = recv q (true, func)
fun PIf(e, p1_, p2_) =
    let
        val VInt(i) = e
    in
        if i > 0 then p1_ () else p2_ ()
    end
```

図 4: 図 3 のインタプリタに higher-order abstract syntax と shortcut deforestation を適用したもの

```
infix @@
infix ++

fun %i (fs as (fint, _, _, _, _)) = fint i
fun $x fs = x fs
fun (e1 @@ e2) (fs as (_, fapp, _, _, _)) =
    fapp (e1 fs, e2 fs)
fun (e1 ++ e2) (fs as (_, _, fadd, _, _)) =
    fadd (e1 fs, e2 fs)
fun IF(e1, e2, e3) (fs as (_, _, _, fif, _)) =
    fif (e1 fs, fn () => e2 fs, fn () => e3 fs)
fun \f (fs as (_, _, _, _, fabs)) =
    fabs(fn v => f (fn fs => v) fs)
```

のようになる。すると、たとえば eval は

```
datatype value =
    Num of int
  | Closure of value -> value

val eval =
    (fn n => Num(n),
     fn (Closure(f), v) => f v,
     fn (Num(v1), Num(v2)) => Num(v1 + v2),
     fn (Num(0), _, v) => v ()
      | (Num(_), v, _) => v (),
     Closure)
```

と定義でき、

```
- ((\(fn x => \(fn y => $x ++ $y))) @@ %3 @@ %7) eval;
val it = Num 10 : value
```

のように使える。しかし、これだと効率は非常に悪くて、 $ack(3,7)$ で約 38 秒もかかるので、実用になるかどうかは疑問である。

6 related work & future work

6.1 higher-order abstract syntax

先にも述べたが、元の論文 [3] では「binder を持つ項の操作が簡単になる」という点しか述べられておらず、インタプリタに使うという話は書かれていない。というか、 higher-order abtract syntax を用いて表現した式をうまく eval する方法は、わりと最近 [1] まで知られていなかったらしい。

6.2 shortcut deforestation

eval が exp に対する catamorphism であるということは、だれでも気が付くと思うのだが、意外と最近の論文 [2] で主張されている。この辺の話が主に list や tree に対する操作を対象としてきたためかもしれない。 shortcut deforestation も同様で、インタプリタに使うという話は (調べた限りでは) 見当たらなかった。

6.3 partial evaluation

冒頭にも書いたが、インタプリタからコンパイラを導出する方法としては、二村 projection が有名である。しかし、最初の naive なインタプリタに二村 projection を使えば、確かに pattern matching のオーバーヘッドは

```
infix @@
infix ++

datatype 'a exp =
    % of int
  | $ of 'a exp
  | @@ of 'a exp * 'a exp
  | ++ of 'a exp * 'a exp
  | IF of 'a exp * 'a exp * 'a exp
  | Abs of 'a exp -> 'a exp
  | Embed of 'a

fun cata (%i) (fs as (fint, _, _, _, _)) = fint i
  | cata ($x) fs = cata x fs
  | cata (e1 @@ e2) (fs as (_, fapp, _, _, _)) =
    fapp (cata e1 fs, cata e2 fs)
  | cata (e1 ++ e2) (fs as (_, _, fadd, _, _)) =
    fadd (cata e1 fs, cata e2 fs)
  | cata (IF(e1, e2, e3)) (fs as (_, _, _, fif, _)) =
    fif (cata e1 fs, fn () => cata e2 fs, fn () => cata e3 fs)
  | cata (Abs(f)) (fs as (_, _, _, _, fabs)) =
    let
fun c e = cata e fs
    in
fabs(c o f o Embed)
    end
  | cata (Embed(v)) _ = v
```

図 5: higher-order abstract syntax を用いて表現された λ 式に対して、一般の catamorphism を計算するプログラム

消えるが、`lookup` とか `update` とかのオーバーヘッドまでは消えない。 higher-order abstract syntax を用いたインタプリタに二村 projection を使うという手も考えられるが、 shortcut deforestation のみを行うのに比べて大きなメリットがあるかどうかは、まだよく考えていない。

6.4 attribute grammar

式に対して "compositional" に値を求めていくという点で、 attribute grammar と非常に関係が深そうなのだが、知識がなさすぎてまったくわからない。 attribute grammar の世界では今回の話が既知であるという可能性もあると思うので、ちゃんと調べる必要がある。

参考文献

[1] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 21st Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, 1994.

[2] Graham Hutton. Fold and unfold for program semantics. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 280–288, 1998.

[3] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[4] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 306–313, 1995.