MetaOCamlを使った部分評価器の実装

岩井亜里紗　(指導教員：浅井健一)

1　はじめに

MetaOCaml は、Multi-stage programming に対応した OCaml の拡張であり、MetaOCaml を使うことにより、コードを出力するようなプログラムが比較的容易に書けるようになる。本研究では、let-insertion をする部分評価器の作成を MetaOCaml を用いて行った。今回、部分評価器本体を継続渡し形式 (CPS) に書くことで let-insertion を実現するとともに、コードを残す部分では K 正規形を返すことで、結果を直接形式で得るようにしている。更に部分評価器の入力言語に shift/reset を加える方法についても議論する。

2　部分評価

部分評価とは、プログラム中で評価できることが明らかな部分を評価したプログラムを作成する処理である。これにより、計算時間の短縮などが期待される。

ここで扱う部分評価は、プログラムを事前に評価可能 (Static) な部分と評価不可能 (Dynamic) な部分に分け、それから評価していくというものである。

例えば、`(fun n → n * (1 + 2))` という式の場合、`(1 + 2)` は評価できるので Static、`n` の値がわからず、`n * (1 + 2)` は評価できないため、これは Dynamic となる。よって、この式を部分評価すると、`(fun n → n * 3)` のようになる。

3　shift/reset

継続とは、計算のある時点における残りの計算のことであり、shift は継続を取ってくるための命令、reset は継続の範囲を指定する命令である。例えば，`1 + reset(2 * shift(fun k → k (k 3)))` という式の場合、shift で取ってくる継続は、reset の中から shift を除いた `2 * []` という部分である。この継続は `(fun n → 2 * n)` と書けるので、`k (k 3)` の部分は、`(fun n → 2 * n) ((fun n → 2 * n) 3)` となり、結果は 12 である。これで reset の中身は切り取られたため、残りの `1 + []` の `[]` に 12 を渡して、式全体の結果は 13 となる。

4　継続渡し形式

継続渡し形式 (CPS：Continuation Passing Style) とは、継続を引数として渡すことで全ての関数呼び出しを末尾呼び出しの形にしたプログラム形式のことである。また、末尾呼び出しとは、関数を呼び出した後に覚えておかなくてはならない作業がないような関数呼び出しのことを言う。

5　MetaOCaml

MetaOCaml は、Multi-stage programming(MSP) に対応した OCaml の拡張である。また、MSP とは、Static な式と Dynamic な式を扱うプログラミングのことである。

MSP のための構文は以下の通りである。

Brackets (.⟨···⟩.)
　Dynamic な式をコードとして出力する

Escape (.~(···))
　コードの一部分を Static な式として実行する

Run (.!(···))
　コード全体を実行する

これらを用いた簡単なインタプリタを以下に記す。

```
let rec eval e env =
  match e with
    Int(i) → .⟨VInt(i)⟩.
  | Var(s) → env s
  | App(e1, e2) →
    .⟨((take_fun .~(eval e1 env))
                .~(eval e2 env)⟩.
  | Add(e1, e2) →
    .⟨VInt((f_int.~(eval e1 env))+
           (f_int.~(eval e2 env)))⟩.
    ⋮
  | Fun(s, e2) →
    .⟨VFun(fun v →
         .~(eval e2 (ext env s .⟨v⟩.)))⟩.
```

このインタプリタは、Int(i) のような数字は、.⟨VInt(i)⟩. のように数字のコードに、Add(e1, e2) のような足し算は、e1、e2 の二つの式をそれぞれコードに変換し、出てきた二つのコードを足すようなコードを返している。例えば `(fun x → x + 1) (2 + 3)` という式を上のインタプリタにかけると、

```
# eval App(Fun("x", Add(Var "x", Int 1)),
Add(Int 2, Int 3)) env0;;
- : ('a, 'b value) code =
.⟨(match
   (VFun (fun v_2 →
     (f_int v_2) + (f_int (VInt 1)))) with
  | VFun(f_1) →
     (f_1 (VInt((f_int (VInt 2)) + (f_int
(VInt 3)))))
  | _ → (raise Notmatch))⟩.
```

のようになる。このように、この段階ではすべて Dynamic となり、全てコードとして出力される。そして、これを

```
# .!(eval App(Fun("x", Add(Var "x", Int
1)), Add(Int 2, Int 3)) env0);;
- : ('a value) code = VInt 6
```

のように .! を用いて実行すると、確かに答えの 6 が返ってくることがわかる。

6 実装

本研究では、[1] を参考にして以下のような順で部分評価器の実装を行った。

6.1 Static/Dynamic の導入

入力言語に Static/Dynamic を導入したインタプリタを以下に記す。

```
let rec eval2 e env =
  match e with
    Int(i) →  (Dyn(.⟨VInt(i)⟩.))
  | IntS(i)  →  (VInt(i))
  | Var(s) →  (env s)
  | App(e1, e2) →
    Dyn(.⟨(((take_fun .~(eval2 e1 env))
                      .~(eval2 e2 env)⟩.)
  | AppS(e1, e2) →
       (((take_fun .~(eval2 e1 env))
                   .~(eval2 e2 env)
  | Add(e1, e2) →
    Dyn(.⟨VInt((f_int .~(eval2 e1 env))+
           (f_int .~(eval2 e2 env)))⟩.)
  | AddS(e1, e2) →
    VInt((f_int(eval2 e1 env))+
           (f_int(eval2 e2 env)))
    ⋮
  | Fun(s, e2) →
    Dyn(.⟨VFun(fun v →
         .~(eval2 e2 (ext env s .⟨v⟩.)))⟩.)
  | FunS(s, e2) →
    VFun(fun v → eval2 e2 (ext env s v))
  | Lift(e) →
    Dyn(.⟨.~(lift (eval2 e env))⟩.)
```

この場合、返ってくる式には Static のものと Dynamic のものがあるため、Dynamic なものは Dyn(...) とする。また、入力言語に関して IntS のように、S がついているものは Static なものである。

上の式の (2 + 3) を Static にした式をこのインタプリタにかけると、

```
# eval2 App(Fun("x", Add(Var "x", Int 1)),
Lift(AddS(IntS 2, IntS 3))) env0;;
- : ('a, 'b value) code =
.⟨(match
   (VFun (fun v_2 →
     (f_int v_2) + (f_int (VInt 1)))) with
  | VFun(f_1) → (f_1 (VInt 5))
  | _ → (raise Notmatch))⟩.
```

のようになり、Static な式のみが実行されていることがわかる。

6.2 let-insertion の導入

```
  .⟨fun f → fun x →
                .~((fun y → x) .⟨f x⟩.)⟩.
```

のような式 (ただし、.⟨ ··· ⟩. はコードとして残す Dynamic な部分、.~(···) は Dynamic な式の中に存在する Static な部分) において、関数 f がループしてしまう場合、実際には答えが出ないはずだが、この式を let-insertion を行わない部分評価器にかけると、結果は .⟨fun f → fun x → x⟩. となり、x という結果が返ってくる関数に変わってしまう。そのため、(f x) を式に残すことが必要である。

そこで Dynamic な関数適用が現れた場合、

```
.⟨fun f → fun x → let a = (f x) in x⟩.
```

のように let 文を用いてその関数適用を継続の前に残すようにする。これにより、上のように部分評価器にかけたために結果が変わってしまうことを防ぐことができる。

これを実装するため、まずインタプリタを CPS で書き換え、関数適用の部分を以下のようにした。

```
 | App(e1, e2) →
   eval3 e1 env (fun e1' →
     eval3 e2 env (fun e2' →
       Dyn(.⟨let a =
         (take_fun .~(d_to_code e1'))
                   .~(d_to_code e2') in
         .~(d_to_code(k (Dyn(.⟨a⟩.)))))⟩.)))
```

返すコードの初めに let 文を挿入して、関数適用を変数にいれ、その後 CPS による継続にその変数を渡すようになっている。

上の式をこのインタプリタにかけると、確かに、

```
# eval3 (Fun ("f", Fun ("x", AppS (FunS
("y", Var "x"), App (Var "f", Var "x")))))
env0 k0;;
- : ('a value) code =
Dyn.⟨(VFun (fun v_1 →
       (VFun (fun v_2 →
          let a_3 = (take_fun v_1) v_2 in
v_2)))))⟩.
```

のように返ってくる。

6.3 shift/reset の導入

インタプリタを CPS に変換したため、継続が使えるようになり、shift/reset の導入が可能となった。現在、[2] を参考に、Static な shift/reset の導入が終わっている。

7 まとめと今後の課題

本研究では、MetaOCaml を用いて、let-insertion をする部分評価器の実装を行った。これにより、ステージングを行うことでコンパイラや部分評価器を得られることがわかった。その一方で、出力されるコードには多くのタグが入ってしまうこともわかった。現在、Static な shift/reset の導入までは終わっているため、今後は Dynamic な shift/reset の導入も進めていきたい。

参考文献

[1] Walid Taha “A Gentle Introduction to Multi-stage Programming”, Domain-Specific Program Generation 2003, pp. 30–50, (2003).

[2] Kenichi Asai “Logical Relations for Call-by-value Delimited Continuations”, In A chapter of Trends in Functional Programming, Vol, 6, pp. 63–78, Intellect (2007).