ラムダ式コンパイラの実装

2017SE057 新美伊織 2017SE058 野橋祐人 2017SE080 諏訪貴大

1 ラムダ計算

ラムダ計算は計算の実行を関数への引数の評価と適用を用いて行う計算体系である。ラムダ計算では、関数をラムダ抽象を用いて記述する。例えば $f(x) = x$ の場合、$\lambda x.x$ となる。この関数に整数の 2 を適用する際は $(\lambda x.x)2$ と記述し、結果は 2 となる。

1.1 ラムダ式の構文

今回翻訳するラムダ計算の式は BNF を用いて以下のように定義される。

$\mathrm{E} ::= id \mid int \mid (fn\ id => E) \mid (E\ E) \mid (fix\ E)$
$\mid (E + E) \mid (E - E) \mid (E * E) \mid (E/E)$
$\mid (\mathrm{E} = \mathrm{E}) \mid (pair\ E\ E) \mid (fst\ E) \mid (snd\ E\ )$
$\mid (fst\ E) \mid (snd\ E\ ) \mid (inl\ E)$
$\mid (case\ E\ of\ 1(id) => E, 2(id) => E)$

ラムダ抽象に対応する式は $(fn\ id => E)$ であり、関数適用に対応する式は $(E\ E)$ である。

2 コンビネータ

コンビネータとは、実行環境に依存せずに完全にそのふるまいが定義された関数のことである。各コンビネータの簡約規則に従い簡約を行うことにより、計算を実行する。また、ラムダ式からコンビネータ式への変換が可能である。

2.1 コンビネータの式

コンビネータにおける式は以下のように与えられる。

$$C ::= x \mid c \mid A \mid C\ C$$

$$A ::= \mid S \mid K \mid P \mid F \mid N \mid L \mid R \mid A \mid X$$

x は変数、 c は定数である。

次に各コンビネータのラムダ式との対応を以下に示す。

$S \rightarrow \lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$

$K \rightarrow \lambda x.\lambda y.x$

$P \rightarrow \lambda x.\lambda y.(x, y)$

$F \rightarrow \lambda x.x[1]$

$N \rightarrow \lambda x.x[2]$

$L \rightarrow \lambda x.1(x)$

$R \rightarrow \lambda x.2(x)$

$A \rightarrow \lambda x.\lambda y.\lambda z.(case\ x\ of\ 1(x_1)$
$\Rightarrow y\ x_1, 2(x_2) \Longrightarrow z\ x_2$

$X \rightarrow \lambda x.fix(x)$

2.2 コンビネータの簡約規則

2.1 章の各コンビネータに対する簡約規則を以下に示す。

$$S\ M_1\ M_2\ M_3 \implies (M_1\ M_3)\ (M_2\ M_3)$$
$$K\ M_1\ M_2 \implies M_1$$
$$F\ (P\ M_1\ M_2) \implies M_1$$
$$N\ (P\ M_1\ M_2) \implies M_2$$
$$A\ (L\ M_1\ )\ M_2\ M_3 \implies M_2\ \mathrm{M}_1$$
$$A\ (R\ M_1\ )\ M_2\ M_3 \implies M_3\ \mathrm{M}_1$$
$$X\ M \implies M\ (X\ M)$$

2.3 ラムダ式のコンビネータ式への翻訳規則

まずコンビネータ式 C に対して、$\lambda^* x.C$ を以下のように定義する。

$\lambda^* x.C = KC$ （Cにxが自由に表れない場合）

$\lambda^* x.x = \mathrm{SKK}$

$\lambda^* x.(\ C_1 C_2\ ) = \mathrm{S}(\lambda^* x.C_1)(\lambda^* x.C_2)$

次に、ラムダ式 M のコンビネータ式への翻訳 $\overline{M}$ は以下のように与えられる。

$$\overline{x} = x$$
$$\overline{c} = c$$
$$\overline{\lambda x.M} = \lambda^* x.\overline{M}$$
$$\overline{M_1\ M_2} = \overline{M_1}\ \overline{M_2}$$
$$\overline{(M_1, M_2)} = P\ \overline{M_1}\ \overline{M_2}$$
$$\overline{M[1]} = F\ \overline{M}$$
$$\overline{M[2]} = N\ \overline{M}$$
$$\overline{1(M)} = L\ \overline{M}$$
$$\overline{2(M)} = R\ \overline{M}$$
$$\overline{(case\ M_1\ of\ 1(x) \Rightarrow M_2, 2(y) \Rightarrow M_3} = A\ \overline{M_1}(\lambda^* x.\overline{M_2})(\lambda^* x.\overline{M_3})$$
$$\overline{fix[1]} = X\ \overline{M}$$

3 実装

今回はプログラミング言語 OCaml を用いて 1.1 章で与えたラムダ式をコンビネータ式に変換するコンパイラを実装する。
また、コンビネータ式を計算するプログラムも実装する。

3.1 データ構造の定義

OCaml 上で 1.1 章の構文と 2.1 章のコンビネータ式のデータ構造を定義する。
それらを以下に示す。

```
type id = string
type exp =
| EId of id
| EInt of int              | ELam of id * exp
| EApp of exp * exp        | EFix of exp
| EAdd of exp * exp        | EMin of exp * exp
| EMul of exp * exp        | EDiv of exp * exp
| EEq of exp * exp         | EPair of exp * exp
| EFst of exp              | ESnd of exp
| Inl of exp               | EInr of exp
| ECase of exp * string * exp * string * exp
| ELet of string * exp * exp
type con =
   | INT of int
   | ID of string
   | S | K | P | F | N | L | R | A | X
   | EQ | ADD | SUB | MUL | DIV
   | CC of con * con
```

1.1 章のラムダ式が exp と対応し、2.1 章のコンビネータ式が con と対応する。

3.2 コンビネータ式に翻訳する関数の定義

ラムダ式データ型をコンビネータ式データ型に翻訳する関数を定義する。関数を以下に示す。

```
let rec cc c =
   let rec lambda x m=
      match m with
| ID(e) -> if e = x    then CC(CC(S,K),K)
                       else CC(K,ID(e))
| CC(e1,e2) -> CC(CC(S,lambda x    e1),lambda
x e2)
| P -> CC(K,P) | F -> CC(K,F) | N -> CC(K,N)
| L -> CC(K,L) | R -> CC(K,R) | A -> CC(K,A)
| X -> CC(K,X)
      | _ -> failwith "unknown expression"
   in
   match c with
   | EInt(n) -> INT(n)
   | EId(n) -> ID(n)
   | ELam(x,e) -> (lambda x (cc e))
   | EApp(x1,x2) -> CC(cc x1,cc x2)
   | EPair(x1,x2) ->CC(CC(P,cc x1),cc x2)
   | EFst(x) -> CC(F,cc x)
   | ESnd(x) -> CC(N,cc x)
   | EInl(x) -> CC(L,cc x)
   | EInr(x) -> CC(R,cc x)
   | ECase(e1,x,e2,y,e3)
    -> CC(CC(CC(A, cc e1),
    (lambda x (cc e2))),(lambda y (cc e3)))
   | EFix(x) -> CC(X,cc x)
   | _ -> failwith "unknown expression"
```

補助関数 lambda は 2.2 章の $\lambda^* x.C$ に対応しており、関数 cc は 2.2 章の翻訳規則に対応する。関数 cc は exp 型の式を受け取り、con 型の式を返す。
実行の例を以下に示す。

```
# cc(Elam("x",EId "x"));;
- : con = CC (CC (S, K), K)
# cc(Elam("x",EPair(EId "x",EId "x")));;
- : con =CC (CC (S, CC (CC (S, CC (K, P)),
  CC (CC (S, K), K))), CC ((CC (S, K), K))
```

3.3 Pretty-print 関数の定義

コンビネータ式データ型をプリントする Pretty-print 関数を定義する。以下にその関数を示す。

```
let rec print_conv c =
   match c with
   | INT(n) -> string_of_int n
   | ID(n) -> n
   | CC(c1,c2) ->
        begin
          match c1 with
| CC(e1,e2) ->if e1 = S
  then (print_conv e1) ^"("^(print_conv e2)
  ^")(" ^(print_conv c2) ^")"
  else (print_conv c1)    (print_conv c2)
| _ -> (print_conv c1) ^ (print_conv c2)
        end
   | S -> "S"
   | K -> "K"
   | P -> "P"
   | F -> "F"
   | N -> "N"
   | L -> "L"
   | R -> "R"
   | A -> "A"
   | X -> "X"
   | _ -> failwith "unknown expression"
```

関数 print_conv は con 型の式を受け取り文字列を返す関数である。再帰呼び出しと文字列を連結する演算子 ^ を用いて実装した。実行の例を以下に示す。

print_conv(CC(CC(S,K),K));;

- : string = “S(K)(K)”

#print_conv(CC(CC(CC(S,K),K),CC(CC(S,K),K)));;

- : string = “S(K)(K)S(K)(K)”

しかし、この関数では、上から 1 番目の結果のように無駄な括弧がプリントされたり、上から 2 番目の結果のように適用の式の際に括弧がプリントされるようになってしまった。これは改善できると考えられるが実装できなかった。

3.4 最外最左簡約を行う関数の定義

まず、コンビネータが変換可能か判定する関数と最外最左簡約を 1 回行う関数を定義する。以下にその関数を示す。

```
let rec conj c =
   match c with
   | CC(c1,c2) ->
       begin
          match c1 with
          | X -> true
          | F ->
              begin
                 match c2 with
                 | CC(e1,e2) ->
                      begin
                         match e1 with
                         | CC(f1,f2) -> if f1 = P then
                                              true
                                              else false
                         | _ -> false
                      end
                 | _ -> false
              end
          | N ->
              begin
                 match c2 with
                 | CC(e1,e2) ->
                      begin
                         match e1 with
                         | CC(f1,f2) -> if f1 = P then
                                              true
                                              else false
                         | _ -> false
                      end
                 | _ -> false
              end
          | CC(c3,c4) ->
              begin
                 match c3 with
                 | K -> true
                 | CC(c5,c6) ->
                      begin
                      match c5 with
                      | S -> true
                      | CC(c7,c8) ->
                           begin
                              match c7 with
                         | A -> if (c8 = L) || (c8 = R)
                                    then true
                                    else false
```

```
                    | _ -> false
                    end
                | _ -> false
                end
            | _ -> false
        end
    | _ -> false

let rec eval_conv c =
  match c with
  | CC(c1,c2) ->
      begin
        match c1 with
        | X -> CC(c2,CC(X,c2))
        | F ->
            begin
              match c2 with
              | CC(e1,e2) ->
                  begin
                    match e1 with
                    | CC(f1,f2) -> if f1 = P
                                   then f2
        else failwith "unknown expression"
                    | _ ->
             failwith "unknown expression"
                  end
              | _ ->
              failwith "unknown expression"
            end
        | N->
            begin
              match c2 with
              | CC(e1,e2) ->
                  begin
                    match e1 with
                    | CC(f1,f2) -> if f1 = P
                                   then e2
        else failwith "unknown expression"
                    | _ ->
             failwith "unknown expression"
                  end
              | _ ->
              failwith "unknown expression"
            end
        | CC(c3,c4) ->
            begin
              match c3 with
              | K -> c4
              | CC(c5,c6) ->
                  begin
                    match c5 with
          | S -> CC(CC(c6,c2),CC(c4,c2))
                    | CC(c7,c8) ->
                        begin
                          match c7 with
                        | A -> if c8 = L
                               then CC(c4,c6)
                               else if c8 = R
                               then CC(c2,c6)
      else failwith "unknown expression"
                    | _ ->
            failwith "unknown expression"
                    end
                | _ ->
            failwith "unknown expression"
                end
            | _ -> failwith "unknown expression"
          end
      | _ -> failwith "unknown expression"
```

判定する関数 conj では、受け取った式のパターンマッチを繰り返し、簡約を行うことができる場所にコンビネータが現れたら true を返しそれ以外なら false を返すようにした。

簡約を 1 回行う関数 eval_conv では、関数 conj の true を返す部分を 2.2 章で示した簡約後のコンビネータ式を返すように変更することで実装した。実行例を以下に示す。

```
# conj(CC(CC(CC(S,K),K),CC(CC(S,K),K)));;
- : bool = true
#conj(CC(CC(S,K),K));;
- : bool = false
# eval_conv(CC(CC(CC(S,K),K),INT 1));;
- : con = CC (CC (K, INT 1), CC (K, INT 1))
# eval_conv(CC(CC(CC(S,CC(CC(S,CC(K,P)),
CC(CC(S,K),K)),CC(CC(S,K),K)),INT 1));;
- : con = CC (CC (CC (CC (S, CC(K, P)), CC
(CC (S,K), K)), INT 1), CC (CC (CC (S, K), K),
INT 1));;
```

3.5 可能な限り最外最左簡約を行う関数の定義

関数 conj,eval_conv を組み合わせて可能な限り簡約を行う関数を定義する。
次ページにその関数を示す。

```
let rec evalc c =
  if (conj c) = true then evalc(eval_conv c)
  else c
```

関数 evalc は、受け取った式が簡約可能なら１回簡約を行った結果を evalc に渡して再帰し、簡約不可能となったらその式を返す。
実行例を以下に示す。

```
# evalc(CC(CC(CC(S,K),K),INT 1));;
- : con = INT 1
#evalc(CC(CC(CC(S,K),K),CC(CC(S,K),K)));;
- : con = CC (CC (S, K), K)
```

3.6 特定の処理を行うトップレベルプログラムの定義

最後に、以下の処理を行うトップレベルプログラムを定義をする。
１．ラムダ式を入力し、入力したラムダ式データをプリントする。
２．ラムダ式をコンビネータに変換し、変換したコンビネータ式をプリントする。
３．コンビネータ式に対して最外最左簡約を実行し、結果をプリントする。
このプログラムを作成するには、1.1 章で定義したラムダ式の字句解析器と構文解析器が必要であるため、ocamllex と ocamlyacc で実装された模範解答を用いることにする。
以下には main.ml 内の関数のみを示す。

```
let lamcc e =
  match e with
  | Exp(e1) -> print_conv(cc e1)
let eval c =
  match c with
  | Exp(e1) -> print_conv(evalc(cc e1))
let rec read_eval_print () =
  print_string "->";
  flush stdout;
  let line = Parser.toplevel Lexer.tokenize
(Lexing.from_channel stdin)
  in
  let out =
    try
      Pretty.ppProgram (line)
    with
      Parsing.Parse_error -> "Syntax error."
  in
  print_string "Source expr:   ";
  print_endline out;
  print_string "Compiled to:   ";
  print_endline (lamcc line);
  print_string "Reduced to:   ";
  print_endline (eval line);
  read_eval_print ()
;;
read_eval_print ()
```

ここでは補助関数 lamcc と eval を定義した。関数 lamcc はラムダを受け取り、コンビネータ式に変換したものをプリントする関数である。関数 eval はラムダ式を受け取り、コンビネータに変換し、簡約を行った後の式をプリントする関数である。実行例を以下に示す。

```
->((fn x => x) (fn x=> x));
Source expr:  ((fn x => x) (fn x => x))
Compiled to:  S(K)(K)S(K)(K)
Reduced to:  S(K)(K)
->(fn x => (x,x));
Source expr: (fn x => (x, x))
Compiled to: S(S(KP)(S(K)(K)))(S(K)(K))
Reduced to: S(S(KP)(S(K)(K)))(S(K)(K))
```

4 まとめ

今回のプログラムでは、ラムダ式からコンビネータの翻訳はほぼうまくいった。しかし、コンビネータ EQ や ADD などの実装をしておらず、また、簡約を行う関数で一部のコンビネータにおいて簡約を行うこと

ができないこともあった。まだ改善できる
部分がたくさんあると考えられる。

5 参考文献

[1] 大堀 淳・ジャック ガリグ・西村 進
(1999)『コンピュータサイエンス入門
<アルゴリズムとプログラミング言語>』
p.199-212
[2]「コンピュータサイエンス入門<アルゴ
リズムとプログラミング言語>」
p.150-151 問 11.7 の解答例
<http://owari.se.nanzan-u.private/
seminar2019/interpreter/src_lex_yacc.zip>
2019 年 11 月アクセス
[3]「コンビネータ論理 – Wikipedia」
<https://ja.wikipedia.org/wiki/
コンビネータ論理>
2019 年 11 月アクセス