輪講：Type Systems and Programming Languages

Mizukami Tatsuo

平成 13 年 4 月 23 日

6 Nameless Representation of Terms

変数をどのように表すかにはいくつかの方法があり、それぞれにはそれなりの利点がある。

1. 代入時の名前つけ変え規則を形式化し、文字列で変数を表現する。
2. 名前つけ変えを必要としない方法を新たに作成する。

バグが早期に見つかりやすい等の実装上の都合として、この本においては、作者は後者を選択している。

6.1 Terms and Contexts

ここで紹介する方法は De Bruijin さんが考えだした方法であり、文字列の変数の代わりに数字を使用する。その方法とは、もとの文字列変数とその定義されている変数の間の λ の数が k 個の場合、その変数を k に置き換えるという方法である。

つまり、例をあげると、

$$\lambda x.\lambda y.x(y\ x) \Longrightarrow \lambda.\lambda.1(0\ 1)$$

$$\lambda f.(\lambda x.f(\lambda y.(x\ x)y))(\lambda x.f(\lambda y.(x\ x)y))$$
$$\Longrightarrow \lambda.(\lambda.1(\lambda.(1\ 1)0))(\lambda.1(\lambda.(1\ 1)0))$$

のように、文字列の変数から新たに定義した表現方法に変換されるわけである。名前無しの term(λ.1 等) を de Bruijin term といい、実際の数字の部分を de Bruijin index という。

次に de Bruijin term の syntax を定義するために、0 から n-1 までの数のうちで高々n 個の自由変数しかない term の集合を T_n とすると、T_n は次のようにして定義できる。

6.1.2 DEFINITION[TERMS]

1. $k \in T_n$ whenever $0 \leq k < n$
2. if $t_1 \in T_n$ and $n > 0$, then $\lambda.t_1 \in T_{n-1}$
3. if $t_1 \in T_n$ and $t_2 \in T_n$, then $(t_1\ t_2) \in T_n$

補足として、T を $\{T_0, T_1, \ldots\}$ として定義し、T_n の要素を n-term という。

ここまでは、文字列の変数から数字の変数に名前のつけ変えの原理を説明し、term の集合を定義したのだが、変換の際に λ 式の中に自由変数がある場合がまだ説明されていない。ここではその場合に対応するために、まず、文字列の自由変数と数字との対応が naming context F に定義されているとする。例えば、

$$F = \begin{array}{l} x \to 4 \\ y \to 3 \\ z \to 2 \\ a \to 1 \\ b \to 0 \end{array}$$

これを略記すると $F = x, y, z, a, b$ として表され、$dom(F)$ を $\{x, y, z, a, b\}$ と定義する。(定義 6.1.3) 輪講用の本には、元の文字列バージョンの λ 式と naming context F を使用しての名前を数字に変換するためのアルゴリズムが書かれていないと思うが、自分なりに考えたところによると、

- もし、元の変数が束縛変数の時、(囲まれている λ の数) を使用。
- もし、元の変数が自由変数の時、(囲まれている λ の数)+(F で対応づけられた数) を使用。

そうすると、上記の例の F のもとでは、

$$\begin{array}{rcl} x(y\ z) & \to & 4(3\ 2) \\ \lambda x.yx & \to & \lambda.4\ 0 \\ \lambda w.\lambda a.x & \to & \lambda.\lambda.6 \end{array}$$

実際の名前無しの場合の λ 式の代入を定義することは、単純な問題ではない。なぜなら、自由変数の数字の割り当て方法を上のように定義したために、代入する値も、代入される場所によってその割り当て番号が変わってしまう。そのための代入を行う場所によって、代入する値を変えるための関数 $\uparrow^d_c$ を次のように定義する。

6.2.1 DEFINITION[SHIFTING]

$$\uparrow_c^d (k) = \begin{cases} k & \text{if } k < c \\ k + d & \text{if } k \geq c \end{cases}$$

$$\uparrow_c^d (\lambda.t_1) = \lambda. \uparrow_{c+1}^d (t_1)$$

$$\uparrow_c^d (t_1\ t_2) = \uparrow_c^d (t_1) \uparrow_c^d (t_2)$$

d は λ 式を一つ評価すると割り当てる数を増やす値であり、c は naming context で束縛された変数の数として使用される。直感的には $\uparrow_c^d$ は適用される λ 式の自由変数の値を 1 だけ増やすことを行う。すると、上記の式を利用して代入を定義できる。

6.2.4 DEFINITION[SUBSTITUTION]

$$[j \to s]k = \begin{cases} s & \text{if } k = j \\ k & \text{otherwise} \end{cases}$$

$$[j \to s](\lambda.t_1) = \lambda.[j + 1 \to \uparrow^1 (s)]t_1$$

$$[j \to s](t_1\ t_2) = ([j \to s]t_1[j \to s]t_2)$$

実際の数字バージョンの λ 式の評価方法は次のようにして定義される。

6.2 Evaluation

$$(\lambda.t_{12})v_2 \to \uparrow^{-1} ([0 \to \uparrow^1 (v_2)]t_{12})$$

つまり、この章で行いたかったことは、文字列変数で表現された λ 式を数学的に美しく表すために、文字列変数と自然数との対応関係を定義を行い、実際に自然数変数に変換された λ 式における代入と評価を定義した。

7 An ML Implementation of the Lambda-Calculus

この章は 4 章で話されたインタープリターの実装を、6 章の変数の自然数による定義とそれに基づく代入等を使用して、拡張を行おうというお話です。

7.1 Terms and Contexts

前章の term の定義から ML の実装をすなおに表現すると、

```
type term =
    TmVar of int
  | TmAbs of term
  | TmApp of term * term
```

インタープリタの環境等の情報を保存するために、info という型を付け加える。

```
type term =
    TmVar of info * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

変数の定義時の naming context のサイズを保存するために、新たな型を付け加える。TmVar の 2 番目の int 値は変数を実際に表示するときにデバッグ目的に使用されるらしい。また、TmAbs の string 値は見易さのために、関数に名前をつけれるようにしている。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * term
  | TmApp of info * term * term
```

7.2 Shifting and Substitution

この章も前章で定義したシフト、代入の関数を OCaml で再定義しただけです。

```
let termShift d t =
    let rec walk c t = match t with
      TmVar(fi, x, n) -> if x >= c then TmVar(fi, x+d, n+d)
                         else TmVar(fi, x, n+d)
    | TmAbs(fi, x, t1) -> TmAbs(fi, x, walk (c+1) t1)
    | TmApp(fi, t1, t2) -> TmApp(fi, walk c t1, walk c t2)
    in walk 0 t

let termSubst j s t =
    let rec walk c t = match t with
      TmVar(fi, x, n) -> if x = j+c then (termShift c s)
                         else TmVar(fi, x, n)
```

```
| TmAbs(fi, x, t1) -> TmAbs(fi, x, walk (c+1) t1)
| TmApp(fi, t1, t2) -> TmApp(fi, walk c t1, walk c t2)
in walk 0 t
```

7.3 Evaluation

一回評価する関数 eval1 を定義して、その関数評価を停止するまで評価する関数 eval を定義しています。それだけです。