純粋関数型言語Ruby

Rubyで型なしラムダ計算

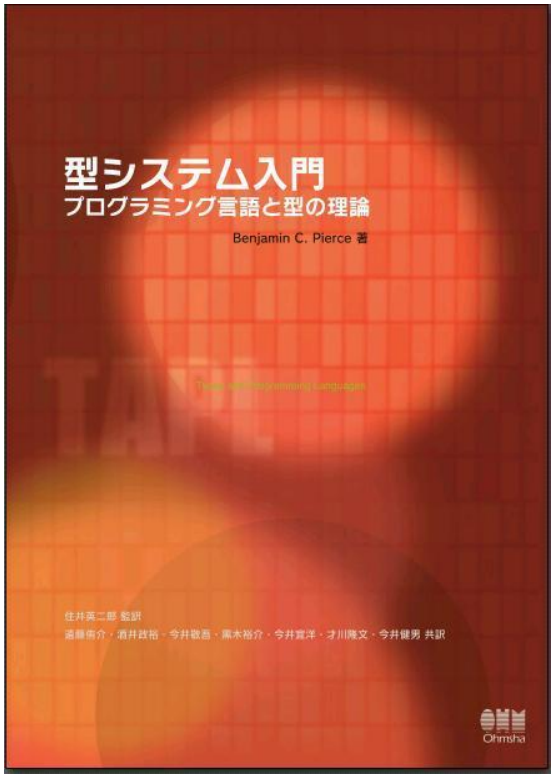
郡司啓(@gunjisatoshi)

宣伝

- Rubyist Magazine 0042 号リリースしました
  - http://magazine.rubyist.net/?0042
- 今号からURLが変わっています
  - Rubyist Magazine移行後記
  - http://magazine.rubyist.net/?0042-RubimaMigrationToRuby2.0
- 編集者募集中！
  - お近くのRuby編集者までお声掛けください
  - もしくはmagazine@ruby-no-kai.orgまでメールください

今日のねらい

- 「型システム入門」の魅力を知ってもらう
- 第5章をRubyで実装してみました（対応するページをページ番号で表示）



（型なし）ラムダ計算とは

- 「すべてのオブジェクトが関数」
  「型がない」と言うより「型が一つ（関数）しかない」
- 値すらも関数で表現する
- という縛りで「計算」を表現したもの
- 「型システム入門」P39あたりを参照

関数とは

- 入力と出力の対応規則のこと
- 入力して良いものの集合を「定義域」と呼ぶ
- 出力される可能性のあるものの集合を「値域」と呼ぶ
- RubyならProcオブジェクトが（ほぼ）関数に相当する
- 「型システム入門」P11あたりを参照

Rubyで（型なし）ラムダ計算

- 使っていいのは引数（入力）を一つだけ取るProcオブジェクトのみ
- そのProcオブジェクトの定義域は「引数を一つだけ取るProcオブジェクト」全体、値域も「引数を一つだけ取るProcオブジェクト」全体
  - Proc.call(x)のxはProcオブジェクトが前提、戻り値は必ずProcオブジェクト
- 関数をオブジェクトとして扱えればRuby以外の言語でも同様のこと（型なしラムダ計算）ができる

カリー化と自由変数

- f1=Proc.new {|a, b| ... }; f1.call(foo, bar)

と、

- f2=Proc.new {|a| Proc.new {|b| ... } };
f2.call(foo).call(bar)

は同じ

- 引数を複数取る関数を定義したい時に利用する
- f2の内側のProc内の変数aを自由変数と呼ぶ
- 「型システム入門」P44あたりを参照

Proc#callとProcのリテラル表現

- Proc.new {|a| ... }と->(a){ … }は同じ
- Proc#call(a)とProc#[a]は同じ
- 以後は後者の書き方で統一する

Churchブール値

- Trueの定義
  二つの引数を取り、必ず一つ目の引数の値をそのまま返す関数をTrueと定義する
  ```
  my_t = ->(t) {->(f) { t } }
  ```
- Falseの定義
  二つの引数を取り、必ず二つ目の引数の値をそのまま返す関数をFalseと定義する
  ```
  my_f = ->(t) {->(f) { f } }
  ```
- 「型システム入門」P44あたりを参照

- Churchブール値をRubyのtrue、falseに変換するには
  ```
  def my_to_bool(b)
    b[true][false]
  end
  ```
- 「型システム入門」P48あたりを参照

Churchブール値で論理演算

- And関数の定義

```
my_and = ->(b) { ->(c) { b[c][my_f] } }
```

- Or関数の定義

```
my_or = ->(b) { ->(c) { b[my_t][c] } }
```

- Not関数の定義

```
my_not = ->(b) { b[my_f][my_t] }
```

- 「型システム入門」P45あたりを参照

デモ1

- Churchブール値を定義してみる
- Churchブール値で論理演算してみる
- 演算結果を出力する
- https://gist.github.com/gunjisatoshi/5283459

データ構造の表現

- 二つ組（いわゆるドット対）

```
my_pair = ->(f) { ->(s) { ->(b) { b[f][s] } } }
```

- 一つ目の取り出し（いわゆるcar）

```
my_first = ->(p) { p[my_t] }
```

- 二つ目の取り出し（いわゆるcdr）

```
my_second = ->(p) { p[my_f] }
```

- ドット対、car、cdrがあれば、ツリー構造やリストなど様々なデータ構造が表現できる
- 「型システム入門」P45あたりを参照

Church数

- 自然数の定義
  二つの引数を取り、1つ目の引数の関数を2つ目の引数の値にn回適用した関数で自然数を表現する
  my_zero = ->(s) {->(z) { z } } (実はFalseと同じ)
  my_one = ->(s) {->(z) { s[z] } }
  my_two = ->(s) {->(z) { s[s[z]] } }
  my_three = ->(s) {->(z) { s[s[s[z]]] } }
- 自然数の「次の数」を求める関数の定義
  一つの引数（二つの引数を取り、1つ目の引数の関数を2つ目の引数の値にn回適用した関数）を取り、「二つの引数を取り、1つ目の引数の関数を2つ目の引数の値にn+1回適用する関数」を返す
  my_succ = ->(n) { ->(s) { ->(z) { s[n[s][z]] } } }
- 「型システム入門」P46あたりを参照
- Church数をRubyのIntegerに変換するには

```
def my_to_i(n)
  n[:succ.to_proc][0]
end
```

- 「型システム入門」P48あたりを参照

Church数同士で計算

- 足し算の定義

```
my_plus = ->(m) { ->(n) { ->(s) { ->(z) {
  m[s][n[s][z]] } } } }
```

- 掛け算の定義

```
my_times = ->(m) { ->(n) { ->(s) { m[n[s]] } } }
```

- 「型システム入門」P46あたりを参照

「前の数」と「引き算」

- my_zz = my_pair[my_zero][my_zero]
- my_ss = ->(p) { my_pair[my_second[p]][my_plus[my_one][my_second[p]]] }
- my_pred = ->(m) { my_first[m[my_ss][my_zz]] }
- my_minus = ->(m) { ->(n) { n[my_pred][m] } }
- 「型システム入門」P47あたりを参照

デモ2

- Church数を定義してみる
- Church数同士で演算してみる
- 演算結果を出力する
- https://gist.github.com/gunjisatoshi/5284331

制御構造の表現

- ifの定義

```
my_if = -> (l) { -> (m) { -> (n) { l[m][n] } } }
```

- 「型システム入門」P44あたりを参照

- 繰り返し（再帰）生成器

```
y_combinator = -> (f) {
  (-> (q) { -> (m) { f[q[q]][m] } })[
    -> (q) { -> (m) { f[q[q]][m] } } ] }
```

- 「型システム入門」P49あたりを参照

「等しさ」の判定

- my_iszero = ->(m) { m[->(x) { my_f }][my_t] }
- 「型システム入門」P46あたりを参照

- my_equal = ->(m) { ->(n) { my_and[my_iszero[m[my_pred][n]]][my_iszero[n[my_pred][m]]] } }
- 「型システム入門」P47あたりを参照

応用:フィボナッチ数を求めてみる

- my_fib_ = ->(f) { ->(n) {
  my_if[my_iszero[n]][
  -> (x) { my_zero }][
  my_if[my_equal[n][my_one]][
  -> (x) { my_one }][
  -> (x) { my_plus[f[my_pred[n]]][
  f[my_pred[my_pred[n]]]]}]][my_zero] } }
- my_fib = y_combinator[my_fib_]
- 戻り値を遅延評価させないと無限ループになるので注意
- 「型システム入門」P50あたりを参照

デモ3

- フィボナッチ数を出力してみる
- https://gist.github.com/gunjisatoshi/5291931

まとめ

- Rubyでラムダ計算を実装してみた
- 手を動かすと理解が早いので実装してみよう
- 「型システム入門」読書会やりませんか