Haskellで Lazy K 処理系を作る

67回生 @_yingtai

1. あいさつ

はじめまして、yingtaiです。今回はHaskellでLazy Kインタプリタを作ったことについて書きたいと思います。

Lazy Kは、いわゆる「難解プログラミング言語（Esolang）」と呼ばれる言語のひとつです。去年の部誌の表紙を飾ったbrainfuckとかが有名ですね。

brainfuckは > < + - [ ] , . の8つの文字だけで構成される言語ですが、Lazy Kはさらにシンプルで、 S, K, I, そして括弧 () のみで構成されます。

brainfuckがチューリングマシンを直接シミュレートする言語[1]なら、Lazy Kはラムダ計算（コンビネータ論理）を直接シミュレートする言語です。

…と言ってもなんのこっちゃという人が多いと思うので、ラムダ計算の概念から説明したいと思います。

この記事の前半は主に、関数型言語についてよく知らないというプログラマー向けです。後半では実際にHaskellでの実装を見ていきます。

2. 導入、関数型言語とは何か

さて、HaskellやLazy Kは「純粋関数型言語」にカテゴライズされるプログラミング言語です。 これらの言語は、CやJavaなどの手続き型言語[2]とは大きく異なるものです。

---

[1] brainfuckはWeb上にごまんと解説があるのでここでは詳説しません

[2] Javaなども「純粋関数型言語と比較して」手続き的であると言えます

関数型言語とは、すべての計算を関数の評価によって行う言語です。これらにはScala, Lisp, OCaml なども含まれます。

純粋関数型言語は「コンピューターの状態」という概念を持たない言語のことを指します。「タイミング」という概念がないと言ってもいいかもしれません。

純粋関数型言語においては、すべての変数は変更不可能です。そのため、式の値はそれを構成する関数や変数のみによって定まります。式を評価するタイミング（つまりコンピューターの状態）によって式の値が変わることはありません。式の中の関数や変数の値が別のタイミングで変更されることがないからです。

要するに、純粋関数型言語において、関数や変数は数学におけるそれと同じように振る舞います。数学で x = y + z とおいた後に同じ文脈で x = y * z と「おき直す」ことがないのと同じです。このような純粋関数型言語の性質を参照透過性と呼びます。

また、式をいつ評価してもよいので、手続き型プログラミング言語と違った順序で評価することができます。いわゆる遅延評価 (lazy evaluation) です。これによって一部の不必要な操作をせずにすみ、計算量が最適化されます。Haskellは遅延評価を原則としており、Lazy Kでは全ての式が遅延評価されます。

3. Lazy K とは?

Lazy Kを理解するためにはいくつかの予備知識が必要です。この章ではそれについて書きます。

例えばラムダ計算は、最も重要な概念です。冒頭で書いたように、Lazy Kはラムダ計算を直接プログラミング言語に落とし込んだものです。

3-1. ラムダ計算の基礎

少しプログラミングを離れて抽象的な話になります。

ラムダ計算とは、関数という概念を抽象化した計算体系です。関数はギリシャ文字のラムダ(λ)を使った書き方で表されます。たとえば、

$$f(x) = x+2$$

この関数は、ラムダ記法では次のように表現します:

$$\lambda x.\ x+2$$

このような式をラムダ式と呼び、式を作る構文のことをラムダ抽象(λ-abstraction) と呼びます。

xに2を代入したいときは、次のように書きます。

$$((\lambda x.\ x+2)\ 2)$$

この操作を、「式に2を適用(apply)する」といいます。これを実行することで、（当然ながら）以下のような結果になります。

$$((\lambda x.\ x+2)\ 2) \to 4$$

これを実行することをβ簡約 (β-reduction)といいます。

正確にはこの例には語弊があります。というのは、ラムダ計算には自然数や加算という概念はないからです。ラムダ計算における変数はすべて関数なのです。（が、チャーチ数という概念を使って自然数を表すことは可能です。これを定義することでLazy Kで数値を扱うことができます。後で詳しく説明します。）これはつまり、ラムダ計算においては関数の引数や戻り値がすべて関数であることを意味します。クロージャなどを思い浮かべるとよいでしょう。

$$\lambda f.\ (\lambda x.\ fx)$$

たとえば、上のラムダ式は「変数 f を引数にとって、「変数 x を引数にとって f(x) の結果を返す関数」を返す関数」を表します。右辺の ”f x”はfにxを適用するという意味です。

この関数のように、ラムダ計算では1つの関数は引数を1つしか取りません。2つ以上の引数を取る場合、1つの引数を取るラムダ式を組み合わせます。

また、このラムダ式は、

$$\lambda fx.\ fx$$

という形に略記されることがあります。これはあくまで略記であって、直接2引数をとるという意味ではないことに注意してください。

β簡約が登場したので、α変換 (α-conversion) の話もしておきます。α変換とは、変数の名前を付けなおすことです。例えば、次の2つのラムダ式は本質的に同じです:

$\lambda \quad xy.\ x$

$\lambda ab.\ a$

基本的にはこれだけです。が、全ての変数の名前を自由に置換できるわけではありません。例えば、このラムダ式のxをyに替えると、

$\lambda \quad yy.\ y$

となりますが、これは意味合いが全く異なってしまいます。まあ、直感的に理解できると思います。要するにα変換が示しているのは、変数の名前は本質的ではない、ということです。

ラムダ式の同値関係は、α変換とβ簡約によって定義されます。同値なラムダ式は、一方にα変換かβ簡約を実行することで同じラムダ式にできます。また、β簡約の余地がないラムダ式を正規形であると言います。

また、ラムダ式は、その中に自由変数を含むことができます。自由変数とは、λによって値が束縛されていない変数のことです。例えば、

$\lambda x.\ y$

における y は自由変数です。

少し駆け足になってしまいましたが、ラムダ計算の文法に関してはこれくらいでいいでしょう。次の節では応用を見ていきます。

3-2. チャーチ数

では、先にちらっと紹介したチャーチ数について詳しく説明します。チャーチ数とは、ラムダ式を使って自然数を表すものです。

一般に、自然数は次のように定義できます。[3]

ペアノの公理：自然数は次の5条件を満たす。

---

[3] http://ja.wikipedia.org/wiki/ペアノの公理

- 自然数 0 が存在する。
- 任意の自然数 a にはその後者 (successor) suc(a) が存在する。(= a + 1)
- 0 より前の自然数は存在しない。
- 異なる自然数は異なる後者を持つ: $a \neq b$ のとき $suc(a) \neq suc(b)$ である。
- 0 がある性質を満たし、a がある性質を満たせば suc(a)もその性質を満たすとき、すべての自然数はその性質を満たす。（数学的帰納法）

ペアノの公理による自然数の構成は集合論によるものがもっとも有名ですが、ラムダ計算によって同様に構成することも可能です。これによって定義される自然数をチャーチ数と呼びます。

チャーチ数は具体的には以下のように定義されます:

$$0 := \lambda fx.\ x$$

$$1 := \lambda fx.\ fx$$

$$2 := \lambda fx.\ f(fx)$$

$$3 := \lambda fx.\ f(f(fx))$$

この定義のもとで、チャーチ数 n の後者を求める関数は次のようになります:

$$succ := \lambda nfx.\ f(nfx)$$

たとえば、succ(2) の場合、計算は次のような流れになります。

$$\begin{aligned} succ\ 2 &= \lambda fx.\ f((\lambda fx.\ f(fx))fx) \\ &= \lambda fx.\ f(f(fx)) \\ &= 3 \end{aligned}$$

（おまけ1）同じようにして加算も定義できます。

$$plus := \lambda mnfx.\ mf(nfx)$$

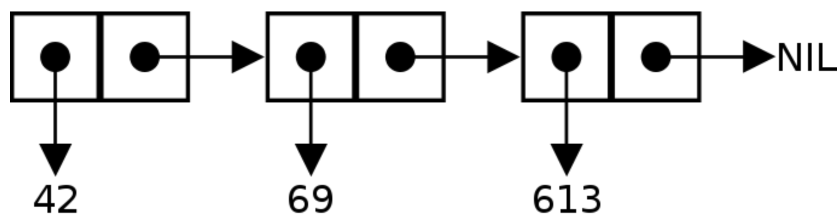
（おまけ2）前者 (predecessor) を求める関数は少し複雑で、次のようになります。

$$\mathrm{pred} := \lambda nfx.\ n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$$

さて、このようにして自然数が定義できました。これによって文字コードを表すことが可能になりました。Lazy Kを実装するためにはもう一つ重要なものを定義する必要があります。入出力を表すリストです。

3-2. リスト

導入で紹介した関数型言語のLispでは、リストはペアというデータの連なりとして表現されます。下図のようなイメージです。



図の長方形を構成している2つの正方形がペアで、左の部分をcar部、右の部分をcdr部と呼びます。[4]例えば一番左のペアのcar部は数値42 へのポインタ、cdr部は右のペアへのポインタです。

このリストは、右から、「「「空リストに613をつなげたリスト」に69をつなげたリスト」に42をつなげたリスト」と考えられます。Lispではcons関数（cons対を作る関数）を使ってこれを次のように書けます。

(**cons** 42 (**cons** 69 (**cons** 613 nil)))

これはlist関数を使って次のようにも書けます。

[4] car, cdr という妙な呼称は大昔のハードウェアに由来してるのだそう

(**list** 42 69 613)

これによって、42, 69, 613 のリストを実現します。

Lazy Kでも、同じようにリストを実装します。メモリの代わりにラムダ計算を使うだけです。cons, car, cdr, 真偽値を次のように定義します。

$$
\begin{aligned}
\mathrm{true} &:= \lambda xy.\ x \\
\mathrm{false} &:= \lambda xy.\ y \\
\mathrm{cons} &:= \lambda sbf.\ fsb \\
\mathrm{car} &:= \lambda p.\ p\ \mathrm{true} \\
\mathrm{cdr} &:= \lambda p.\ p\ \mathrm{false} \\
\mathrm{nil} &:= \mathrm{false}
\end{aligned}
$$

これらがそれぞれをきちんと表現できていることは、実際にラムダ式を書いてみるとすぐに理解できると思います。 たとえば (cons 2 nil) をこの定義に従って書いた後、carに適用すると2が、cdrに適用するとnil (false)が返ってくるのが確認できます。

$$
\begin{aligned}
&((\lambda sbf.\ fsb)\ 2\ \mathrm{nil}) \\
\rightarrow\ &\lambda f.\ f\ 2\ \mathrm{nil} \\
&(\mathrm{car}\ (\lambda f.\ f\ 2\ \mathrm{nil})) \\
\rightarrow\ &(\lambda f.\ f\ 2\ \mathrm{nil})\ (\lambda\ xy.\ x) \\
\rightarrow\ &(\lambda\ xy.\ x)\ 2\ \mathrm{nil}
\end{aligned}
$$

$\to 2$

#cdrの例は省略

3-3. SKIコンビネータ

ここまでくれば、Lazy Kを理解するまであと一歩です。ここでサンプルコードを見てみましょう。[5]

```
K(SII(S(K(S(S(K(SII(S(S(KS)(S(K(S(KS)))(S(K(S(S(KS)(SS(S(S(KS)K))(
KK)))))(S(S(KS)(S(KK)(S(KS)(S(S(KS)(S(KK)(S(KS)(S(S(KS)(S(KK)(SII)
))

(K(SI(KK)))))))(K(S(K(S(S(KS)(S(K(SI))(S(KK)(S(K(S(S(KS)K)(S(S(KS)
K)I)
(S(SII)I(S(S(KS)K)I)(S(S(KS)K)))))(SI(K(KI)))))))))(S(KK)K)))))))(K(S(K
K)(S(SI(K(S(S(S(S(SSK(SI(K(KI))))(K(S(S(KS)K)I(S(S(KS)K)(S(S(KS)K)
I))
(S(K(S(SI(K(KI)))))K)(KK))))(KK))(S(S(KS)(S(K(SI))(S(KK)(S(K(S(S(KS
)K)))
 (SI(KK))))))(K(K(KI)))))(S(S(KS)(S(K(SI))(SS(SI)(KK))))(S(KK)
 (S(K(S(S(KS)K)))(SI(K(KI)))))))))(K(K(KI))))))))))(K(KI)))))(SI(KK)))))
 (S(K(S(K(S(K(S(SI(K(S(K(S(S(KS)K)I))(S(SII)I(S(S(KS)K)I)))))))K))))

(S(S(KS)(S(KK)(SII)))(K(SI(K(KI)))))))(SII(S(K(S(S(KS)(S(K(S(S(SI(KK))
 (KI))))(SS(S(S(KS)(S(KK)(S(KS)(S(K(SI))K)))))(KK))))))(S(S(KS)
(S(K(S(KS)))(S(K(S(KK)))(S(S(KS)(S(KK)(SII)))(K(S(S(KS)K)))))))(K(S(
S(KS)(S(K(S(S(SI(KK))(KI))))(S(KK)(S(K(SII(S(K(S(S(KS)(S(K(S(K(S(S(
KS)(S(KK)
 (S(KS)(S(K(SI))K))))(KK)))))(S(S(KS)(S(KK)(S(K(SI(KK)))(SI(KK)))))
```

[5] http://homepages.cwi.nl/~tromp/cl/lazy-k.html

```
(K(SI(KK))))))))(S(S(KS)(S(K(S(KS)))(S(K(S(KK)))(S(S(KS)(S(KK)(SII)))
 (K(SI(K(KI))))))))(K(K(SI(K(KI))))))))(S(K(SII))(S(K(S(K(SI(K(KI))))))
 (S(S(KS)(S(KK)(SI(K(S(K(S(SI(K(KI)))))K)))))(K(S(K(S(SI(KK))))
 (S(KK)(SII)))))))))))(K(SI(K(KI))))))))(S(S(KS)K)I)
 (SII(S(K(S(K(S(SI(K(KI)))))K))(SII)))))
```

　これはエラトステネスの篩を使って実装された、素数を出力するコードです。コピペです。
見て分かるように、S, K, I, () から構成されています。
　ここで S, K, I は、それぞれ次のラムダ式と等価です:

$$S := \lambda xyz.\ (xz)(yz)$$

$$K := \lambda xy.\ x$$

$$I := \lambda x.\ x$$

　Lazy Kではソースコード自体が一つのラムダ式になっています。
コード中のS, K, I をそれぞれこれらのラムダ式と置き換える、と考えてもよいでしょう。
　ちなみにコンビネータとは、自由変数を含まないラムダ式のことです。これまでラムダ計算として扱ってきたもののほとんどはコンビネータを扱うコンビネータ論理で、Lazy Kで扱うラムダ計算も同様です。
　では、Lazy Kの具体的な処理について検討しましょう。
Lazy Kを実行するとき、次のような処理が必要になるでしょう:

1. 入力 V を文字列からチャーチ数のリストに変換する。
2. ソースコードをラムダ式 E として解釈する。
3. V を E に適用して新たなラムダ式 E’ を得る。
4. E’ をチャーチ数のリストと見なし、文字列に変換し、出力する。

4. 実装

では実際に処理系を実装してみましょう。以下がソースコードです。

```
import Data.Char (isSpace)
import Data.Functor ((<$>))
import System.Environment
import Text.Parsec
import Text.Parsec.String

data Expr = V Int       -- Value (Index)
          | L Expr      -- Lambda
          | A Expr Expr -- Application
          deriving Eq

-- evaluate
step :: Expr -> Expr
step (A f e)    = step' (A (step f) e)
  where step' (A (L v) e) = step $ subst 0 v (step e)
        step' (A g e)     = A g (step e)
step (L e)      = L $ step e
step e          = e

subst :: Int -> Expr -> Expr -> Expr -- substitute
subst v (L w) e     = L (subst (v + 1) w e)
subst v (A m n) e   = A (subst v m e) (subst v n e)
subst v (V n) e | n < v  = V n
                | n == v = subst' 0 v e
                | n > v  = V (n - 1)
  where subst' t v (L w)     = L (subst' (t + 1) v w)
        subst' t v (A m n)   = A (subst' t v m) (subst' t v n)
        subst' t v (V n) | n < t  = V n
                         | n >= t = V (n + v)

-- combinators
s, k, i, u, b, zero, suc, cons, car, cdr, nil :: Expr
s = L (L (L (A (A (V 2) (V 0)) (A (V 1) (V 0)))))
```

```
k = L (L (V 1))
i = L (V 0)
u = L (A (A (V 0) s) k)
b = L (L (L (A (A (V 2) (V 1)) (V 0))))
zero = (L (L (V 0)))
suc  = (L (L (L (A (V 1) (A (A (V 2) (V 1)) (V 0))))))
cons = (L (L (L (A (A (V 0) (V 2)) (V 1)))))
car  = (L (A (V 0) (L (L (V 1)))))
cdr  = (L (A (V 0) (L (L (V 0)))))
nil  = (L (L (V 0)))

-- encode / decode
s2e :: String -> Expr   -- input
s2e []     = nil
s2e (c:cs) = A (A cons $ i2e $ fromEnum c) $ s2e cs
   where i2e 0 = zero
         i2e n = A suc $ i2e $ n - 1

e2s :: Expr -> String   -- output
e2s e | e == nil  = ""
      | otherwise = (toEnum $ e2s' $ step $ (A car e)) : (e2s $ step $ A cdr e)
   where e2s' (L (L (V _))) = 0
         e2s' (L (L e))     = f e 0
         e2s' _             = 0
         f (A _ n) x        = f n (x + 1)
         f _ x              = x

-- parsers
pExpr, pTerm, pS, pK, pI, pU, pB, pCb, pAp :: Parser Expr
pExpr = foldl1 A <$> many1 pTerm
pTerm = pS <|> pK <|> pI <|> pU <|> pB <|> pCb <|> pAp

pS  = (char 'S' <|> char 's') >> return s
pK  = (char 'K' <|> char 'k') >> return k
pI  = char 'I' >> return i
pU = (char 'i' <|> char '0') >> return u
```

```
pB = (char '1') >> return b
pCb = between (char '(') (char ')') pExpr
pAp = (char '`' <|> char '*') >> pTerm >> pTerm

eval :: String -> String -> String
eval code input = case parse pExpr "" code of
   Left err  -> show err
   Right val -> e2s $ step $ A val (s2e input)

main :: IO()
main = do
   [path] <- getArgs
   code <- filter (not.isSpace) <$> readFile path
   input <- getContents
   putStr $ eval code input
```

まず、Has-

kellでラムダ計算を表現するためにExpr型を定義しています。

```
data Expr = V Int       -- Value (Index)
        | L Expr      -- Lambda
        | A Expr Expr -- Application
        deriving Eq
```

このようにして、ラムダ式を代数的データ型で定義し、抽象構文木として扱います。

またここで、ラムダ式をde Brujin index[6]化しています。de Brujin index はラムダ式の記法の一種で、変数を文字列ではなく数字で表現するものです。

数字 (index) は、変数が数字のある位置から何番目のλで束縛されているかを指します。

例えば、

$$\lambda x.\lambda y.\ x$$

6 読みは「デュブランインデックス」

このラムダ式をde Brujin index化すると

λλ2

となります。

これを先ほどのデータ型で表現すると、

```
L (L (V 1))
```

となります。上のラムダ式ではindexを1から数えていますが、コード中では簡単のためindexを0から数えています。

なぜde Brujin index化するかというと、それによってラムダ計算の実装の際に変数名を気にする必要がなくなるからです。変数名にStringを使ってα変換して、というのはいかにも煩雑です。

de Brujin indexを使うとα変換の必要もなくなります。「簡約と同時にα変換が行われる」と考えることもできるでしょう。

それでは簡約を定義しましょう。この部分です。

```
-- evaluate
step :: Expr -> Expr
step (A f e)    = step' (A (step f) e)
   where step' (A (L v) e) = step $ subst 0 v (step e)
         step' (A g e)     = A g (step e)
step (L e)      = L $ step e
step e          = e

subst :: Int -> Expr -> Expr -> Expr -- substitute
subst v (L w) e     = L (subst (v + 1) w e)
subst v (A m n) e   = A (subst v m e) (subst v n e)
subst v (V n) e | n < v  = V n
                | n == v = subst' 0 v e
                | n > v  = V (n - 1)
          where subst' t v (L w)     = L (subst' (t + 1) v w)
                subst' t v (A m n)   = A (subst' t v m) (subst' t v n)
                subst' t v (V n) | n < t  = V n
                                 | n >= t = V (n + v)
```

de Brujin index化したラムダ式のβ簡約は次のようなルールで実行されます:[7]

- 適用する関数からのλの数と同じ変数を置換する
- 置換対象である変数について、置換対象の変数のindexをnとすると、置き換わる側の式内の自由変数のindexをすべて +n して、置換対象変数と置き換える
- λをはずし、ラムダ式の本体の自由変数のindexをすべて -1 する

続いてSKIコンビネータ、後に必要となるU, Bコンビネータ、0とsucc関数、cons, car, cdr をそれぞれ次のように定義します。

```
-- combinators
s, k, i, u, b, zero, suc, cons, car, cdr, nil :: Expr
s = L (L (L (A (A (V 2) (V 0)) (A (V 1) (V 0)))))
k = L (L (V 1))
i = L (V 0)
u = L (A (A (V 0) s) k)
b = L (L (L (A (A (V 2) (V 1)) (V 0))))
zero = (L (L (V 0)))
suc  = (L (L (L (A (V 1) (A (A (V 2) (V 1)) (V 0))))))
cons = (L (L (L (A (A (V 0) (V 2)) (V 1)))))
car  = (L (A (V 0) (L (L (V 1)))))
cdr  = (L (A (V 0) (L (L (V 0)))))
nil  = (L (L (V 0)))
```

ソースコードが既にesotericな感じがしますが気のせいでしょう。ちゃんと定義できているか確かめるのはそう難しくないと思います。

s2e, e2s関数はチャーチ数のリストと文字列の変換を行います。

```
-- encode / decode
s2e :: String -> Expr   -- input
```

[7] http://en.wikipedia.org/wiki/De_Bruijn_index

```
s2e []     = nil
s2e (c:cs) = A (A cons $ i2e $ fromEnum c) $ s2e cs
   where i2e 0 = zero
         i2e n = A suc $ i2e $ n - 1

e2s :: Expr -> String   -- output
e2s e | e == nil  = ""
      | otherwise = (toEnum $ e2s' $ step $ (A car e)) : (e2s $ step $ A cdr e)
   where e2s' (L (L (V _))) = 0
         e2s' (L (L e))     = f e 0
         e2s' _             = 0
         f (A _ n) x        = f n (x + 1)
         f _ x              = x
```

そして、パーサ部分を実装します。実装にPar-secライブラリを利用します。

実は、Lazy Kには4つの記法があります：コンビネータ記法、Unlambda記法、Iota記法、Jot記法です。3-3.で説明した記法はコンビネータ記法です。

Lazy Kはこれらの記法を一つ以上使って書きます。いくつもの記法をごちゃ混ぜにして書くことも許されています…。

そんな複雑な言語のパースも、Has-kellならごく簡潔に書くことができます。

先ほど定義したU, Bコンビネータはここで使います。pExpr関数がパーサの全体です。

```
-- parsers
pExpr, pTerm, pS, pK, pI, pU, pB, pCb, pAp :: Parser Expr
pExpr = foldl1 A <$> many1 pTerm
pTerm = pS <|> pK <|> pI <|> pU <|> pB <|> pCb <|> pAp

pS  = (char 'S' <|> char 's') >> return s
pK  = (char 'K' <|> char 'k') >> return k
pI  = char 'I' >> return i
pU = (char 'i' <|> char '0') >> return u
```

```
pB = (char '1') >> return b
pCb = between (char '(') (char ')') pExpr
pAp = (char '`' <|> char '*') >> pTerm >> pTerm
```

最後にこれらの関数をまとめます。

```
eval :: String -> String -> String
eval code input = case parse pExpr "" code of
    Left err  -> show err
    Right val -> e2s $ step $ A val (s2e input)

main :: IO()
main = do
    [path] <- getArgs
    code <- filter (not.isSpace) <$> readFile path
    input <- getContents
    putStr $ eval code input
```

以上です。ちゃんと動いているか確認してみましょう。

```
% ghc lazy.hs
% cat id.lazy
SKK
% cat test.in
Hello, world!
% ./lazy id.lazy < test.in
Hello, world!
```

SKK == I なので、期待通りの動作をしていることが分かりますね。

5. 最後に

Lazy KはbrainfuckなどのEsolangに比べると無名な部類だと思いますが、それに負けないくらいに面白い言語です。そしてなにより、ラムダ計算、ひいては関数型言語へのよい入門になる言語でもあると思います。

Haskellの説明はほとんどできませんでしたが、これもとても魅力的な言語です。なんといってもHaskellはエレガントかつ強力です。触ったことのない人はこの機会にかじってみてはいかがでしょうか。

謝辞

この記事を書くにあたって参考にさせていただいたウェブサイト様、文献の筆者様、ならびに関数型言語の魅力を教えてくださった全ての方々に感謝します。
Special thanks to @wasabiz!