Haskell で Brainf*ck Interpreter

2021/8/21 kaage(@ageprocpp)
@JOI夏季セミナー

自己紹介

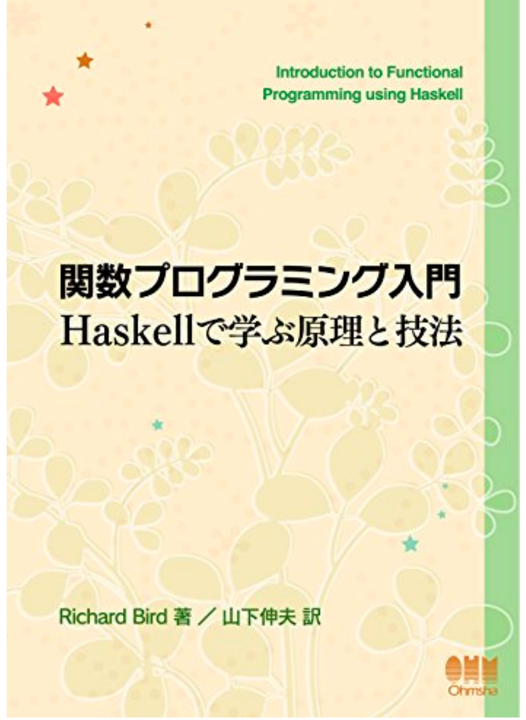
筑波大学附属駒場高等学校 1 年

AtCoder 黄

夏季セミナー参加は 2 回目

最近の実績(?):

- JOI 春合宿
- DDCC 本選
- SuperCon 本選
- JPhO 2次 ← 情報科学と関係ない
- STAGE:0 競プロ・チーム部門 5 位 (with define)



俺ガイルの聖地
千葉市立稲毛高校 →

今日のお話

今日のお話

Haskell について

Haskell is 何

純粋関数型言語のひとつ（cf. Scala）
C/C++/Java/Python などのように、手続きの記述はしない
すべて関数の式（と、リスト内包表記）でプログラムを記述する
保守性は高く、コンパイラ・インタプリタなどの記述に優れている
逆に、アルゴリズムの記述は苦手→競プロに向かない
関数は遅延評価されるため、無限に続くリストなどが使える特徴もある

変数の再代入や変更は不可、変更には関数を通す必要がある
配列の要素を変更したければ、「配列の要素を変更して返す関数」を使う
関数は必ず 1 つしか引数をとらない

Haskell の構文（カリー化）

```
add :: Int -> Int -> Int
add x y = x + y
```

add x は「引数 y に対して x + y を返す**関数**」を返す

```
add :: Int -> Int -> Int -> Int
add x y z = x + y + z
```

add x は「引数 y に対して『引数 z に対して x + y + z を返す関数』を返す関数」を返す

このようにして、実質複数の引数をとる関数を定義できる

Haskell の構文（ポイントフリースタイル）

```
add :: Int -> Int -> Int
add x = (+ x)
```

add x は、(+ x) という関数（Int -> Int）を返すので y を使わずに書ける

well-definedness にさえ気をつけていればかなり書きやすい

```
smaller :: Int -> Int -> Int
smaller x y = if x <= y then x else y
```

if 文ももちろんある

```
average :: [Int] -> Int
average l = s `div` (length l)
    where s = sum l
```

リスト（ランダムアクセス O(N)）がデフォルトで定義され、

where 節を使えば、関数内でも引数に対して const な変数を使える

Haskell の構文

where 節を使いまくれば、FFT も書ける

```
dft :: [Complex Double] -> Int -> [Complex Double]
dft x inverse = if length x == 1 then x else x_transformed
    where
        n = length x
        fh = map snd . filter fst . zip (cycle [True, False]) $ x
        sh = map snd . filter fst . zip (cycle [False, True]) $ x
        fh_transformed = dft fh inverse
        sh_transformed = dft sh inverse
        fh_cur = take (2 * length fh_transformed) $ cycle fh_transformed
        sh_cur = take (2 * length sh_transformed) $ cycle sh_transformed
        zeta = mkPolar 1.0 $ fromIntegral(inverse * 2) * pi / fromIntegral(n)
        sh_zeta = zipWith (*) sh_cur $ iterate (* zeta) 1
        x_transformed = zipWith (+) fh_cur sh_zeta

fft :: [Complex Double] -> [Complex Double] -> [Complex Double]
fft x y = zipWith (*) (dft (zipWith (*) (dft x_resized 1) $ dft y_resized 1) $ -1) $ repeat $ 1 / fromIntegral(n)
    where
        n = head [k | k <- iterate (*2) 1, k >= length x + length y]
        x_resized = x ++ replicate (n - length x) (0 :+ 0)
        y_resized = y ++ replicate (n - length y) (0 :+ 0)
```

Haskellとデータ構造

さて、Haskell では関数の戻り値としてしか値操作ができないのだった

Haskellとデータ構造

さて、Haskell では関数の戻り値としてしか値操作ができないのだった

↓

データ構造は、変更に対しては「変更されたデータ構造」を返すはず

Haskellとデータ構造

さて、Haskell では関数の戻り値としてしか値操作ができないのだった

↓

データ構造は、変更に対しては「変更されたデータ構造」を返すはず

↓

データ構造は永続的でなければならない！

Haskellとデータ構造

永続 stack の例

MkStack (Stack t) t は Stack t 型のなにか
実際は、MkStack a b は a に b を push した状態
top は MkStack s v を剥がして v を返す
pop は MkStack s v を剥がして s を返す

```
data Stack t = Empty | MkStack (Stack t) t

push :: Stack t -> t -> Stack t
push s v = MkStack s v

top :: Stack t -> t
top (MkStack s v) = v

pop :: Stack t -> Stack t
pop (MkStack s v) = s
```

Haskellとデータ構造

永続 stack の例

MkStack (Stack t) t は Stack t 型のなにか
実際は、MkStack a b は a に b を push した状態
top は MkStack s v を剥がして v を返す
pop は MkStack s v を剥がして s を返す

```
data Stack t = Empty | MkStack (Stack t) t

push :: Stack t -> t -> Stack t
push s v = MkStack s v

top :: Stack t -> t
top (MkStack s v) = v

pop :: Stack t -> Stack t
pop (MkStack s v) = s
```

メモリの確保・解放を考えなければならない C++ より大幅に楽に書ける

Haskell で Brainf*ck Interpreter

Haskell の関数を使うと、再帰的な構文解析がきれいに記述できる
身近な (?) 言語である Brainf*ck の Interpreter を書いてみた
（ここでは全部ひとつの関数で適当に済ませているが、もっとちゃんとしたものを書くにはモナドを有効に使ったりする必要がある）

Haskell で Brainf*ck Interpreter

Brainf*ck の構文

‘<‘ : ポインタを左に移動

‘>’ : ポインタを右に移動

‘+’ : メモリの値をインクリメント

‘-‘ : メモリの値をデクリメント

‘.’ : 出力

‘,’ : 入力

‘[‘ : 開き括弧、ポインタの値が 0 でなければループする

‘]’ : 閉じ括弧

Haskell で Brainf*ck Interpreter

たとえば、次のコードで “Hello World!” が出力できる

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>—.+++++++..+++.>>.<-.<.+++.——.———.>>+.>++.
```

このようなコードを入力として、実行できるプログラムを書くのが目標

Haskell で Brainf*ck Interpreter

戦略

「入力文字列を読むカーソル」「ポインタ」「メモリの状態」を引数として、再帰して読んでいく

こうすると、入力を読み進めながらポインタとメモリを変更していける

Haskell で Brainf*ck Interpreter

戦略

「入力文字列を読むストリーム」「ポインタとメモリの状態」を引数として、再帰して読んでいく

こうすると、入力を読み進めるたびにポインタとメモリを変更していける

Haskell で Brainf*ck Interpreter

戦略

「入力文字列を読むカーソル」「ポインタ」「メモリの状態」を引数として、再帰して読んでいく

こうすると、入力を読み進めながらポインタとメモリを変更していける

‘[‘, ‘]’ によるループの処理が必要

さっき作った永続 stack を使う！

Haskell で Brainf*ck Interpreter

正しい戦略

「入力文字列を読むカーソル」「ポインタ」「メモリの状態」

「今までに通った ‘[‘ の位置と Bool のペアを入れた stack」を引数として、再帰して読んでいく

‘[‘ に来たら、ループに入る場合は（position, True）を stack に入れて、ループに入らなかったり、無効なループの中にいるときは（position, False）を入れる

stack が空か、top が True のときのみ処理を行う

（実際には番兵（-1, True）を最初に入れておく）

Haskell で Brainf*ck Interpreter

実装

```
interpret :: BString -> IO ()
interpret s = exec 0 0 (VU.replicate 5000 0) $ MkStack Empty (-1, True)
    where
        exec :: Int -> Int -> VU.Vector Int -> Stack (Int, Bool) -> IO ()
        exec c m_c m st
            | c == BS.length s           = return ()
            | (BS.index s c) == '['      = exec (c + 1) m_c m $ push st (c, (m VU.! m_c /= 0) && (snd $ top st))
            | (BS.index s c) == ']'      = exec (if snd $ top st then prev_c else c + 1) m_c m prev_st
            | not . snd $ top st         = exec (c + 1) m_c m $ st
            | (BS.index s c) == '>'      = exec (c + 1) (m_c + 1) m st
            | (BS.index s c) == '<'      = exec (c + 1) (m_c - 1) m st
            | (BS.index s c) == '+'      = exec (c + 1) m_c (m VU.// [(m_c, m VU.! m_c + 1)]) st
            | (BS.index s c) == '-'      = exec (c + 1) m_c (m VU.// [(m_c, m VU.! m_c - 1)]) st
            | (BS.index s c) == '.'      = do
                putChar . chr $ m VU.! m_c
                exec (c + 1) m_c m st
            | (BS.index s c) == ','      = do
                in_c <- getChar
                exec (c + 1) m_c (m VU.// [(m_c, ord in_c)]) st
            | otherwise                  = exec (c + 1) m_c m st
                where prev_st = pop st; prev_c = fst $ top st
```

Haskell で Brainf*ck Interpreter

課題

メモリ配列を毎回引数として渡しているので、メモリの大きさぶんのコピーコストが毎回かかっている

Haskell で Brainf*ck Interpreter

課題

メモリ配列を毎回引数として渡しているので、メモリの大きさぶんのコピーコストが毎回かかっている

↓

永続配列を使うことで、コピーコストが $\Theta(\log N)$ まで削減できる

Haskell で Brainf*ck Interpreter

課題

メモリ配列を毎回引数として渡しているので、メモリの大きさぶんのコピーコストが毎回かかっている

↓

永続配列を使うことで、コピーコストが $\Theta(\log N)$ まで削減できる

↓

soraie くんの書いた永続セグ木を借りて、使う！（え？）

Haskell で Brainf*ck Interpreter

永続配列のおかげでたくさんの AtCoder の問題が通った

ABC205-C POW (Uniden さんのコード)

ABC212-A Alloy (RedSpica さんのコード)

ABC213-A Bitwise Exclusive Or (Uniden さんのコード)

ARC118-C Coprime Set (Uniden さんのコード)

Haskell で Brainf*ck Interpreter

さらなる課題

計算量が、最初に確保するメモリの大きさに左右されてしまう

本家 Brainf*ck には 30000 個のメモリを持つことが規定されているが、これだけのメモリを使う例はそこまで多くない

メモリの大きさを事前に見積もってそれに合わせたり、メモリの大きさをあとから変更できるようにするとさらなる効率化が望める

まとめ

Haskell は純粋関数型言語

永続データ構造の実装やパーサーの実装などに向いていて、特に永続 stack は非常に簡単に実装できる

永続 stack, 永続配列, 再帰を使って Brainf*ck の interpreter を書いたら、ARC-C を含む AtCoder のいくつかの問題が通せた

ご静聴ありがとうございました