ML 勉強会

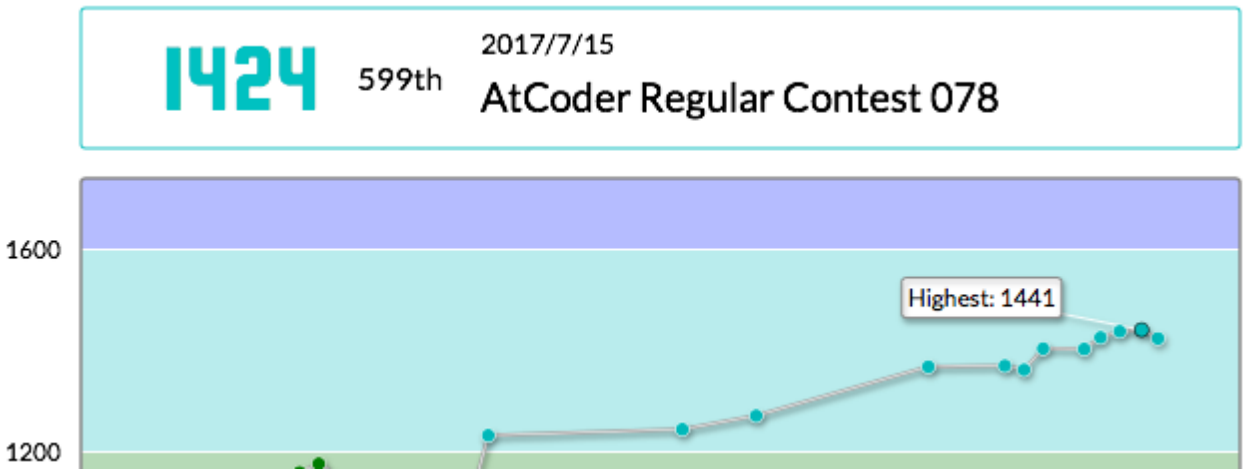
ML で AtCoder に参加した話

@fetburner

2017 年 7 月 22 日

自己紹介

- **水野雅之**
- Twitter:@fetburner
- Coq **でメタ定理の形式化やってる** M2
  - **日常生活で** Coq **しか書かなさすぎてヤバいので競プロで** ML **のリハビリ**



水色コーダーですが，宜しければしばしお付き合いを

Q. なぜ AtCoder?

- A. 初期の頃から OCaml をサポートしていたから
  - ABC 001 (2013 年 10 月) の頃には既にしてた
  - 現在処理系は OCaml 4.02.3 と比較的新しい
    - 昔は OCaml 3.12.1 でつらかった
  - ocamlopt でコンパイルするので，定数倍で TLE しない
    - 昔は ocamlc なので (ry
  - int が 63bit なのでオーバーフロー知らず
- 現在では SML (MLton) もサポート
  - これも定数倍で TLE しにくい
  - int が 31bit なので最悪

アウトライン

1. 簡単な問題と入出力テンプレ
2. 便利な標準ライブラリ関数の紹介
3. To DP or Not To DP
4. ライブラリ整備について
5. まとめ

アウトライン

(早速ですが) 問題例 1: 単純な入出力

> “高橋君はデータの加工が行いたいです。整数 a, b, c と、文字列 s が与えられます。整数 $a + b + c$ と、文字列 s を並べて表示しなさい。”
>
> — practice contest A - はじめてのあっとこーだー

競プロではこの問題と同じように，標準入力から受け取った値を加工して標準出力に流すプログラムが求められる場合が多い

OCaml による解答例

```
let () =
  let a = Scanf.scanf "%d\n" (fun a -> a) in
  let b, c = Scanf.scanf "%d␣%d\n"
    (fun b c -> b, c) in
  let s = Scanf.scanf "%s\n" (fun s -> s) in
  Printf.printf "%d␣%s\n" (a + b + c) s
```

- **入力には** Scanf.scanf **を使おう**
  - OCaml **にもあります**
  - **空白や改行の読み飛ばしが容易**
- **出力には** Printf.printf **を使おう**
  - OCaml **にも** (ry
  - **フォーマットに沿った出力がしやすい**

SML による解答例

```
fun readInt () =
  TextIO.scanStream (Int.scan StringCvt.DEC)
    TextIO.stdIn
val SOME a = readInt ()
val SOME b = readInt ()
val SOME c = readInt ()
(* skip newline *)
val _ = TextIO.inputLine TextIO.stdIn
val SOME s = TextIO.inputLine TextIO.stdIn
val () = print
  (Int.toString (a + b + c) ^ "␣" ^ s)
```

- **数値の入力には** StringCvt **を用いる**
  - Basis Library **には** printf **も** scanf **もない (絶望)**

問題例2: 任意個の入出力

“長さ n の数列 $a_1, \cdots, a_n$ が与えられます。空の数列 b に対して、以下の操作を n 回行うことを考えます。

i 回目には

1. **数列の i 番目の要素 a_i を b の末尾に追加する。**
2. **b を逆向きに並び替える。**

この操作をしてできる数列 b を求めて下さい。”

— ARC 077 C - pushpush

OCaml による解答例

```
let () =
  let n = Scanf.scanf "%d\n" (fun n -> n) in
  let as_ = Array.init n (fun _ ->
    Scanf.scanf "%d␣" (fun a -> a)) in
  Array.fold_left (fun (xs, ys) a ->
    (ys, a :: xs)) ([], []) as_
  |> (fun (xs, ys) -> ys @ List.rev xs)
  |> List.iter (Printf.printf "%d␣")
```

- 任意個のデータを入力する際には Array.init が便利
- 陽な再帰関数を書く前に fold で書けないか検討しよう

SML による解答例

```
val SOME n = readInt ()
val as_=List.tabulate(n,fn _=>valOf(readInt()))
val () =
 let val (xs, ys) =
  foldl(fn(a,(xs,ys))=>(ys,a::xs))([],[])as_ in
  app(fn x=>print(Int.toString x^"␣"))(ys@rev xs)
  print "\n"
 end
```

- **任意個のデータを入力する際には** List.tabulate **が便利**
- **陽な再帰関数を書く前に** fold (ry

構文が重い（半ギレ）

アウトライン

List.fold_left 及び foldl

- ライブラリを整備して良し，問題を解いて良し
- ABC/ARC の C や AGC の A ぐらいまでは fold だけで解ける場合が多い
- まずはこれを使いこなしましょう

fold の例 1: 総和

- ある時は総和を求め

```
List.fold_left ( + ) 0 xs
```

- またある時は総乗を求め

```
List.fold_left ( * ) 1 xs
```

- 剰余類もなんのその

```
List.fold_left (fun x y -> x * y mod n) 1
  List.map (fun x -> x mod n) xs
```

モノイドだしタイトルは総和で良いかと

fold の例 2: アキュムレータを増やしてみる

- 隣り合った要素を取り出したり

```
let hd :: tl = xs in
List.fold_left (fun (ps, x) y ->
  ((x, y) :: ps, y)) ([], hd) tl
```

- 尺取もいける

```
List.fold_left (fun (m, s) a -> let rec
drop s=let a'=Queue.pop q in if a=a' then
s else drop(IntSet.remove a' s)in
Queue.push a q;if IntSet.mem a s then
(max m (IntSet.cardinal s), drop s)
else(m,IntSet.add a s))(0,IntSet.empty)as_
```

map と concat (リストモナド)

- 組み合わせの列挙に便利
- 再帰で DFS を書かなくても総当たりを書けることもある

冪集合

```
List.fold_left (fun xss x ->
  List.concat @@ List.map (fun xs ->
    [xs; x :: xs]) xss) [[]] xs
```

abc **からなる長さ** n **の文字列を列挙**

```
Array.init n (fun _ -> ['a'; 'b'; 'c'])
|> Array.fold_left (fun ss cs ->
  List.concat @@ List.map (fun s ->
    List.map (fun c -> c :: s) cs) ss) [[]]
```

有限集合 (Set) と有限写像 (Map)

- 純粋関数型に書きたい時便利
- OCaml の Set と Map は最大値/最小値の取り出しをサポートしている:
  簡易的な優先度付きキューになる！
  - 重複を許さないので気を付けよう
  - 優先度を減らす操作が無いのでダイクストラ法はキツい
  - プリム法ぐらいならいける
- SML だと NJ Library にあります
  - こちらは最大値/最小値の取り出しをサポートしない

アウトライン

That is the question.

競プロでよく用いられるDP，しかし…

```
let cmb = Array.make_matrix n n 0 in
for i = 0 to n - 1 do
 cmb.(i).(0) <- 1; cmb.(i).(i) <- 1 done;
for i = 1 to n - 1 do for j = 1 to i do
 cmb.(i).(j) <- cmb.(i - 1).(j - 1)
 + cmb.(i - 1).(j) done done
```

- 評価順にシビア
- 値域によってデータ構造を使い分ける必要がある
  - コードの再利用性に乏しい

メモ化再帰の導入

- 評価順は計算機が考えれば良い

```
let dp = Array.make_matrix n n None in
let rec body n r =
  if r = 0 || n = r then 1
  else cmb (n - 1) r + cmb (n - 1) (r - 1)
and cmb n r =
  match dp.(n).(r) with
  | Some x -> x
  | None ->
      let x = body n r in
      dp.(n).(r) <- Some x; x
```

ハッシュの導入・高階関数に

- OCamlのHashtblはハッシュ関数を面倒見てくれる
- メモ化部分を高階関数で括り出そう

```
let memoize n f =
  let dp = Hashtbl.create n in
  let rec get x =
    try Hashtbl.find dp x with Not_found ->
      let r = f get x in
      Hashtbl.add dp x r; r in get
let cmb = memoize 100 (fun cmb (n, r) ->
  if r = 0 || n = r then 1
  else cmb (n - 1, r) + cmb (n - 1, r - 1))
```

アウトライン

二分探索にまつわる様々な言い伝え

- 「無限ループに陥りやすい」
- 「添え字が1ずれる」
- 「半開区間でやれば誤りにくい」

⇑

毎回手書きするからバグるのでは？

高階関数による抽象化

再利用できるライブラリを用意すればよい

```
let rec upper_bound p l r =
  if r <= l + 1 then l
  else
    let m = (l + r) / 2 in
    if p m then upper_bound p m r
    else upper_bound p l m
```

しかし，肝心のライブラリが正しいとは限らない…

まさかの時の定理証明支援系

Coq で証明を付ければ解決！

```
Require Import Arith Omega Recdef.
Variable P : nat -> Prop.
Hypothesis P_dec : forall n,
  { P n } + { ~P n }.
Function lower_bound_aux b n { wf lt n } :=
(* (中略) *)
Theorem lower_bound_spec l r :
  l <= threshold <= r ->
  lower_bound l r = threshold.
```

- Coq には与えられたプログラムを OCaml に変換する機能がある
  - SML の人は Isabelle/HOL 使って

アウトライン

まとめ

- 高階関数の表現力を武器に，関数プログラミングでも結構闘える
- 抽象化機構を利用して，競プロであってもコードの再利用性を高めよう
  - コードを書かなければバグもない
- 時には定理証明支援系の手を借りるのも良い

MLerにはMLerのやり方がある