ソフトウェア技法: No. 8 (末尾再帰その他)

亀山幸義 (筑波大学情報科学類)

1 末尾再帰

この授業では、プログラムを正しく、理解しやすい形で書くことに焦点を当てており、実行効率の良し悪しについてはあまり考慮してこなかった。もう少し正確に言い直すと、アルゴリズムレベルで実現できる効率化を気にしたことはあったが、実装レベルで実現できる効率化についてはまったく気にしてこなかった。本章では、再帰的に関数定義をする場合の興味深い効率化である末尾再帰について説明する。

1.1 末尾再帰とは何か

再帰関数の定義 (本体) において、再帰している関数を「末尾位置 (tail position)」で呼び出しているものに着目しよう。末尾位置における関数呼び出しとは、直観的には、「その関数呼び出しをした後に、もう計算すべきものがない」ということであり、関数呼び出しをした結果が、そのまま、現在の関数の答えになる場合である。

具体的に見てみよう。以下の関数 foo の定義において：

```
let goo x = x * 2 ;;

let rec foo x =
   match x with
   | 1 →   foo 2
   | 2 →   (foo 5) + 3
   | 3 →   foo (goo 8)
   | 4 →   goo (foo 9)
   | _ →   if x>1 then foo 4 else foo 5 ;;
```

match 式の中の 3 行のうち最初の `foo 2` における `foo` と、3 番目の `foo (goo 8)` における `foo` の呼び出し，最後の if 式の中の 2 回の `foo` の呼び出しは、末尾位置である。これらは，`foo 2` や `foo (goo 8)` などが返す答えが、そのまま、いま定義している関数 `foo` の返り値となる。つまり、これらの関数呼び出しの「後」にやる計算がない。

一方、2 番目の `(foo 5) + 3` における `foo` と、4 番目の `goo (foo 9)` における `foo` は、末尾位置でない。2 番目については、`foo 5` の計算のあとに「その結果に 3 を加える」という計算が残っている。また、4 番目については、`foo 9` の計算のあとに「その結果に `goo` を適用する」という計算が残っている。

このように、`foo` の呼び出しごとに、末尾位置にあるかどうかが判定できる。末尾位置における関数呼び出しを「末尾呼び出し」(tail call) と言う。末尾呼び出しは「残りの計算」がないので、計算終了のあとに、呼び出し元の関数に「戻ってくる」必要がない。従って、単なるジャンプ命令で関数呼び出しを実現できるため、処理速度が高速になる。また、「戻ってくる」ための情報を記憶しなくて済むのでメモリ (正確にはスタックのためのメモリ) の使用量が少なくて済む。再帰関数の定義において、その再帰関数を呼び出す部分がすべて末尾呼び出しになっているものを「末尾再帰」と言う。

OCaml, Haskell, Scheme など多くの関数型言語の処理系 (コンパイラ) は、再帰関数が末尾再帰であるかどうかを判定し、その判定結果が YES のものについては効率的なコードを生成する。よって、これらの言語のプログラマとしては、再帰関数を、なるべく、末尾再帰とすると実行効率の点で有利である。こうすることにより、再帰関数の実行効率が、単純なループと同程度になることが期待でき、「何でもかんでも再帰関数で表す」というプログラミングスタイルで済む (処理性能をそれほど犠牲にせずに済む)。

言語処理系による末尾再帰の効率化は、比較的単純であるにもかかわわらず、効果が非常に大きく、それなしには、再帰関数の実行効率が非常に悪い。したがって、、再帰関数を用いてプログラムを組み上げていく関数型プログラミングのスタイルでプログ

ラムを書くためには、処理系 (コンパイラ) が、末尾再帰の効率化をすることが必須条件となる[*1]。

1.2 末尾再帰による関数定義

では、再帰関数を末尾再帰のみで書くことはできるだろうか？もっと正確にいうと、自然に書いたときに末尾再帰にならない関数を、末尾再帰だけを使った定義に書き換える方法はあるだろうか？

この章では、これについて、例にもとづいて考えてみよう。

```
let rec sum x =
  if x = 0 then 0
  else x + (sum (x − 1))
in
  sum 1000000 ;;
Stack overflow during evaluation (looping recursion?).
```

この関数 sum の定義は、漸化式に基づいていて、極めて自然な定義であるが、末尾再帰になっていない。したがって、計算に時間がかかるという以前の問題として 1 回の再帰呼び出しごとにメモリ (スタックのためのメモリ) を使ってしまい、1000000 回繰返すと Stack overflow (スタック用のメモリが不足してしまった) というエラーで止まってしまう。

sum の定義を末尾再帰に変更するためには、引数 x の値だけでなく、「そこまでで計算した結果 (計算の途中結果)」を渡す必要があるので、2 引数の補助関数を作るとよい。

```
let rec sum_aux x s =
  if x = 0 then s
  else sum_aux (x − 1) (s + x) ;;

let sum2 x =
  sum_aux x 0
in
  sum2 1000000 ;;
− : int = 500000500000
```

今度は、エラーにならずに、答えが返ってきた。1000000 回も再帰呼び出しをしたのに、stack overlow にならない、ということは、再帰呼び出しごとにスタックを 1 バイトも使わなかったということであり、末尾再帰の効率化がきちんとなされたことがわかる。

さて、関数 sum_aux の中身を見てみると、これは、もとの sum 関数にあった引数 x のほか、「計算の途中結果」をあらわす s という引数をとっている。ここにどんどん値を足しこんでいき、x=0 になったら、その s を返して計算を終了する。このようにして、「関数の呼び出し元に返る」必要のないプログラムとなっている。

ここで使った引数 s のように、「途中結果を蓄積していく」引数のことを accumulating parameter と言い、関数プログラミングに頻繁に出てくる技法である。

上記と同様の技法で、多くの関数が末尾再帰で定義できる。たとえば、既に出たリストの revserse 関数は同じパターンであった。

[*1] たとえば、JavaScript などの言語では、再帰関数の定義や高階関数を使うこともできるのであるが、末尾再帰への対応がされない処理系が多いため、「関数型言語らしいプログラム」を書くことは推奨されていない。

```
(* append は末尾再帰でないが、これは準備である *)
let rec append lst1 lst2 =
  match lst1 with
  | [] → lst2
  | h::t → h::(append t lst2) ;;

(* 末尾再帰でない reverse の定義 *)
let rec reverse lst =
  match lst with
  | []   → []
  | h::t → append (reverse t) [h] ;;

(* 末尾再帰による reverse の定義 *)
let rec reverse_tailrec lst acc =
  match lst with
  | []   → acc
  | h::t → reverse_tailrec t (h :: acc) ;;

let reverse2 lst =
  reverse_tailrec lst [] ;;
```

末尾再帰バージョン reverse_tailrec では、acc という引数を追加して、そこに途中結果を蓄積することにより、「reverse_tailrec を呼んだ後の処理 (残りの計算)」をなくしている。

整数リストの最大値を求める関数も同様である。

```
(* 末尾再帰でない max の定義 *)
let rec max lst =
  match lst with
  | []   → failwith "empty list"
  | [h]  → h
  | h::t → let m = max t in
           if h > m then h else m ;;

(* 末尾再帰による max の定義 *)
let rec max_tailrec lst m =
  match lst with
  | []   → m
  | h::t → if h > m then max_tailrec t h
           else max_tailrec t m ;;
let max2 lst =
   max_tailrec lst min_int ;;
```

末尾再帰でない定義では、max t を計算した後に、処理がいくつか残っている。一方、末尾再帰バージョンでは、max_tailrec ... の計算結果がそのまま全体の計算結果になっていて、「残りの計算」がない。

max と同様、整数リストの 2 番目を求める関数も末尾再帰にできる。

```
(* 末尾再帰による top2 の定義 *)
let rec top2_tailrec lst top second =
  match lst with
  | []   → second
  | h::t → if h > top then top2_tailrec t h top
           else if h > second then top2_tailrec t top h
           else top2_tailrec t top second ;;
let top2 lst =
   match lst with
   | h1::(h2::t)  → if h1 > h2 then top2_tailrec t h1 h2
                    else top2_tailrec t h2 h1 ;;
   | _ → failwith "ist is too short"
```

このように考えてくると、すべての再帰関数が (accumulating parameter の技法により) 末尾再帰のものに変形できそうに思えてくるが、実際には、できないものもある。たとえば有名な Fibonacci 関数は、以下の定義のように、2 箇所で再帰呼び出しをしているので、簡単な手法で末尾再帰にすることはできない。

```
let rec fib n =
  if n <= 2 then 1
  else (fib (n − 2)) + (fib (n − 1)) ;;
```

この関数の場合、「途中結果」が何であるかわからないので、accumulcating parameter を使うというのはうまく行きそうもない。

このように、うまく末尾再帰に直せないプログラムもあるが、それはたいていの場合「本質的に難しい (計算に時間がかかる)」ケースであり、プログラムをどうこうして高速化するよりも、アルゴリズムそのものを考えなおした方がよい。

たとえば、fib の場合、高速に計算するためには、fib(n) と fib(n+1) を「1 つのセット」にして、計算をすればよいことがわかるので、以下のように書き直す。

```
(* fib n と fib (n+1)をペアにして計算する方法 *)
let rec fibpair n =
  if n = 1 then (1,1)
  else match fibpair (n − 1) with
       | (x1, x2) → (x2, x1 + x2) ;;

let fib2 n = fst (fibpair n) ;;
```

この fibpair 関数はアルゴリズムの意味での高速化は達成したが、末尾再帰になっていないので、コードの実装としては改善の余地がある。ここでも、accumulating parameter を使って、「そこまで計算した途中結果」をあらわすことにすると、以下のように末尾再帰にできる。

```
(* fib n と fib (n+1)をペアにして計算する方法の末尾再帰版 *)
let rec fibpair_tailrec i last p =
  if i = last then p
  else match p with
       | (x1, x2) →
          fibpair_tailrec (i + 1) last (x2, x1 + x2) ;;

let fib2 n = fst (fibpair_tailrec 1 n (1,1)) ;;
```

少し複雑になったので、各自でじっくり解析してほしい。

ところで、fibpair_tailrec は、もとの fibpair と同じ関数にしたかったので、引数をペア (2 ケ組) にした p という引数で受け取っているが、この 2 個は、関数の返り値ではなく関数の引数なので、ばらばらに扱ってもよい。(関数の返り値は 1 つだけなので、ペアにしてまとめるが、関数の引数は何個でも取ることができるため。) そこで、p の部分を 2 つの引数 x1,x2 にして、以下の関数定義にすることもできる。

```
(* fib n と fib (n+1) をペアにしないで計算する末尾再帰版 *)
let rec fib_tailrec i last x1 x2 =
  if i = last then (x1,x2)
  else fib_tailrec (i + 1) last x2 (x1 + x2) ;;

let fib3 n = fst (fib_tailrec 1 n 1 1) ;;
```

そして、fib_tailrec の方が fibpair_tailrec よりも、OCaml らしいプログラムである。つまり、もとのプログラムは、ペアのデータ構造を作るのに、メモリを無駄に少し消費しているが、改善後は、一番最後の瞬間に、1 回ペアを作っているだけである。

さらに、この fib_tailrec はペアを返しているが、呼び出し元の fib では、そのペアの fst(左側の要素) のみを必要としているので、以下のように、改善できる。

```
(* fib n と fib (n+1) をペアにしないで計算する末尾再帰版の改善版 *)
let rec fib_tailrec2 i last x1 x2 =
  if i = last then x1
  else fib_tailrec2 (i + 1) last x2 (x1 + x2) ;;

let fib4 n = fib_tailrec2 1 n 1 1 ;;
```

こうなると、ペアを 1 回も作っていないことが明確になるであろう。

最後のプログラムにおける引数 i と last は 1 つの引数で済むので、さらに簡単なプログラムにできるのであるが、それは各自で考えてほしい。

演習問題 1.

- $n \geq 0$ に対して、x の n 乗を計算する関数 power を末尾再帰で書いてみなさい。
- 以前定義したリストの append 関数は、末尾再帰でない。このことを確認せよ。
  一方、rev_append [1;2;3] [4;5;6] = [3; 2; 1; 4; 5; 6] となる関数 rev_append は末尾再帰で書ける。これを書いてみなさい。
- 末尾再帰の効果を、実際の実行時間を計測することにより、明確にせよ。つまり、同一の意味をもつ再帰関数を、末尾再帰とそうでない書き方の 2 通りで記述し、それらを走らせて実行時間を計測して比較せよ。例題は何でもよいが、たとえば、以下の関数 foo1 とその末尾再帰版を試してみよ。

```
(* 末尾再帰でない *)
let rec foo1 n1 n2 =
  if n1 = n2 then 0.0
  else  let r1 = float n1 in
          (sqrt r1) +. (sin r1) +. (cos r1) +. (foo1 (n1 + 1) n2) ;;
```

なお、末尾再帰でないバージョンは、再帰呼び出しが深くなるとすぐに Stack overflow してしまい時間が測定できないので、再帰呼び出しが深くならないよう注意すること。たとえば、同一の再帰関数を 100 万回呼びだすと Stack overflow になるので、1000 回の再帰呼び出しで計測するか、(それでは時間が短かすぎて計測できない場合) それを 2000 回繰返すなどして合計時間を計測せよ。この場合、外側の「2000 回繰返す」方は、末尾再帰版を使うのが良い。

OCaml 関数を実行したときに、時間計測の方法については OCaml の Sys モジュールの説明を読むか、TA のページを参考にしてほしい。

2 カリー化

これは、末尾再帰の関連の話ではなく、高階関数に関連した話である。関数型言語では、複数の引数を取る関数の書き方として 2 通りある。

```
(* 複数の引数を取る 普通の書き方 *)
let rec foo x y z =
   if x = 0 then y + z
   else foo (x−1) (y * 2) (z+3) ;;

(* 複数の引数を取る 別の書き方 *)
let rec goo (x, y, z) =
   if x = 0 then y + z
   else goo (x−1, y * 2, z+3) ;;
```

第一の書き方が、OCaml では推奨されているが、第二の書き方もできる。第二の書き方は、実は、以下のものと同じである。

```
(* 複数の引数を取る 別の書き方 *)
let rec goo p =
   match p with
   | (x, y, z) →
           if x = 0 then y + z
           else goo (x−1, y * 2, z+3) ;;
```

こでわかるように、goo は実際には 1 引数であり、その中身が、(x,y,z) という 3 つ組だというだけである。実際、上記の関数 goo の型は、`(int * int * int) -> int` であり、`int * int * int` 型の引数を 1 つだけ取っていることがわかる。人間は、このような 3 つ組の引数のことを「引数が 3 つある関数」と思っているだけであり、内部的にはあくまで 1 引数である。

実は、OCaml には 1 引数の関数しか存在しない。複数の引数をもつ関数を書いている気分になっているが、内部的には 1 引数であり、それを「あの手この手で」で、複数の引数がある関数に見せているだけである。

上記の第一の書き方も、実際には以下のように 1 引数関数を組み合わせただけのものである。

```
(* 上記の foo と同じ *)
let rec foo x y z =
   if x = 0 then y + z
   else foo (x-1) (y * 2) (z+3) ;;

(* 上記の foo とまったく同じだが、引数 z を本体に移動させたもの *)
let rec foo2 x y =
   fun z →
     if x = 0 then y + z
     else foo2 (x-1) (y * 2) (z+3) ;;

(* 上記の foo とまったく同じだが、引数 x,y,z をすべて本体に移動させたもの *)
let rec foo3 =
   fun x → fun y → fun z →
     if x = 0 then y + z
     else foo3 (x-1) (y * 2) (z+3) ;;
```

最後の foo3 は、「x をもらうと fun y -> fun z -> ... という関数を返す」という高階関数である。この返ってきた関数は、y をもらうと、fun z -> ... という関数を返すという高階関数である。このように、「返す値が関数」になっている高階関数を 3 段階「入れ子」にしたのが、上記の foo,foo2,foo3 の「本当の姿」である。

foo (および foo2,foo3) の型は、int -> (int -> (int -> int)) であり、これを見ても、返り値が関数の型を持っている高階関数であることが明確にわかる。

なお、第一の書き方 (関数 foo で型が int -> (int -> (int -> int)) であるもの) と、第二の書き方 (関数 goo で型が (int * int * int) -> int であるもの) とは、相互に変換することが可能である。つまり、foo の形の定義を goo の形の定義に変換することができるし、逆もできる。この変換には、この分野の偉人である Curry の名前がつけられており、第二から第一への変換を curring (カリー化)、第一から第二への変換を uncurring (非カリー化) と呼ぶ。そして、第一の形式を「カリー化された関数」と呼ぶことがある。これは難しい呼び名に見えるが、何のことはない、OCaml における「複数引数を持つ普通の関数」の定義のことである。

この章の内容は、プログラミングというよりは、概念的なことだったので、少しわかりにくかったかもしれない。まとめると、以下の通り。

- OCaml には厳密な意味では 1 引数の関数しか存在しない。それを、いろいろな技法を使って、多引数関数 (2 個以上の引数を持つ関数) に見せている。(OCaml ユーザは、多引数関数であると思ってもよい。)
- 1 つ目のやりかたでは、「返り値が関数」となる高階関数を何段階か入れ子にしている。その型は、たとえば、int -> (int -> (int -> int)) である。これをカリー化された関数と呼ぶことがある。
- 2 つ目のやりかたでは、「引数が組」となる高階関数を使う。その型は、たとえば、(int * int * int) -> int である。
- 上記の 2 つの関数定義の方法は、相互に変換可能である。

OCaml の多くのプログラムは、1 つ目のやりかた (カリー化されたスタイル) で書かれている。

演習問題 2.

- (発展) OCaml の加算 (たし算) は、+ であり、これは 2 引数で a+b のように中置記法で書かれる。一方、この加算を前置記法で書く記号もあり、(+) a b のように、かっこをつけた + 記号で書く。この (+) という関数の型を調べ、どのような高階関数であるかを言葉で説明せよ。
- (発展) (+) は、OCaml の関数である以上、内部的に厳密にみれば、2 引数の関数ではなく、1 引数の関数であるはずである。そこで、(+) 3 などの式を評価して、どのようになるか (エラーになるか、型は何になるか、何が返ってくるか、等) を調べよ。さらに、(+) 3 の結果を x という変数に格納して、x 5 などの式を評価して、どのようになるか (エラーになるか、

型は何になるか、何が返ってくるか、等) を調べよ。

- (発展) OCaml で型を表記するとき、`->` は右結合である。その理由を考えなさい。(注。前のバージョンのこのテキストでは「`*` は左結合である」と書いていたが、実際には、`*`は左結合でも右結合でもなかったので、この部分を削除する。)
- (発展) 3 引数のカリー化された関数 f を非カリー化する関数 uncurry3 を定義しなさい。逆のことをする関数 curry3 も定義しなさい。

  例: `uncurry3 (fun x -> fun y -> fun z -> x+y+z)` = `fun p -> match p with (x,y,z) -> x+y+z`