

# プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 8b: 末尾再帰と継続渡し方式

# 目次

① 末尾再帰

② 繼続渡し方式

# 再帰関数の実行効率

再帰関数を素朴に書くと、「繰返し」によるプログラムより実行性能やメモリ使用量で劣る。

```
let sum1 x =
  let result = ref 0 in
  for i = 1 to x do
    result := !result + i
  done;
  !result ;;
sum1 300000 ;; (* 問題ない *)
```

```
let rec sum2 x =
  if x = 0 then 0
  else x + (sum2 (x-1));
sum2 300000 ;; (* メモリ不足で強制終了 *)
```

再帰関数は、 $\text{sum2 } (x-1)$  を計算した後、 $\text{sum2 } x$  に戻って加算をしないといけない。

# 末尾再帰 (1/2)

末尾呼び出し (tail call): 関数呼出しが「計算の最後」であること。

```
(foo 3) ;;      (* OK *)
(foo 3) + 5 ;; (* Not OK *)
foo (goo 3) ;; (* goo: not OK, foo: OK *)
if x then (foo 5) else (goo 8) ;; (* OK *)
```

末尾再帰関数=再帰関数のすべての呼び出しが、末尾呼び出しである関数

- ▶ 末尾再帰関数の計算では、呼び出し元に戻る必要がない。
- ▶ 内部の処理は、繰返し計算と同様 (速度、メモリとも)。

## 末尾再帰 (2/2)

前の例 (末尾再帰でない) :

```
let rec sum2 x =
  if x = 0 then 0
  else x + (sum2 (x-1));;
```

末尾再帰 :

```
let rec sum3 x res =
  if x = 0 then res
  else sum3 (x-1) (x + res);;
```

## 末尾再帰 (2/2)

前の例 (末尾再帰でない) :

```
let rec sum2 x =
  if x = 0 then 0
  else x + (sum2 (x-1));;
```

末尾再帰 :

```
let rec sum3 x res =
  if x = 0 then res
  else sum3 (x-1) (x + res);;
```

```
sum3 5 0
==> sum3 4 5
==> sum3 3 9
==> sum3 2 12
==> sum3 1 14
==> sum3 0 15
==> 15
```

# 末尾再帰に関する最適化

OCaml 言語の標準的な処理系は、末尾再帰を自動的に判定し、(内部的に) 繰返し計算と同等のコードに変換してくれる。

```
sum2 300000 ;; (* メモリ不足で強制終了 *)
sum3 300000 0 ;; (* 問題ない *)
```

上記の処理(末尾再帰最適化)をするかどうかは、言語処理系ごとに違う。

- ▶ OCaml のほか、Scheme(Racket) や Haskell の標準的処理系は、末尾再帰最適化をする。
- ▶ C や JavaScript の多くの処理系は、末尾再帰最適化をしない。

# 末尾再帰関数への変換 (1/3)

末尾再帰でない関数を、実質的に同等の末尾再帰関数にしよう。

```
(* not tail recursion *)
let rec reverse lst =
  match lst with
  | [] → []
  | h::t → List.append (reverse t) [h] ;;
reverse [1;2;3;...;300000] ;; (* エラー *)
```

```
(* tail recursion *)
let rec revappend lst a =
  match lst with
  | [] → a
  | h::t → revappend t (h::a) ;;
revappend [1;2;3;...;300000] [] ;; (* 問題なく動く *)
```

## 末尾再帰関数への変換 (2/3)

power(べき乗) 関数 (末尾再帰でない):

```
let rec power x n =
  if n = 0 then 1
  else x * (power x (n - 1)) ;;
power 1 300000 ;; (* エラー *)
```

末尾再帰:

```
let rec power2 x n res =
  if n = 0 then res
  else power2 x (n - 1) (x * res) ;;
power2 1 300000 1 ;; (* 問題ない *)
```

## 末尾再帰関数への変換 (3/3)

質問: 以下の再帰関数は、同等の末尾再帰関数に変換できるか？

```
let rec append lst1 lst2 =
  match lst1 with
  | [] → lst2
  | h::t → h :: (append t lst2) ;;
append [1;2;3] [4;5] ;; (* [1;2;3;4;5] *)
append [1;2;....300000] [1;2;3] ;; (* エラー *)
```

## 末尾再帰関数への変換 (3/3)

質問: 以下の再帰関数は、同等の末尾再帰関数に変換できるか？

```
let rec append lst1 lst2 =
  match lst1 with
  | [] → lst2
  | h::t → h::(append t lst2) ;;
append [1;2;3] [4;5] ;; (* [1;2;3;4;5] *)
append [1;2;....300000] [1;2;3] ;; (* エラー *)
```

答え: revappend を補助関数に使えばよい(若干効率が悪い)。

```
let rec append2 lst1 res lst2 =
  match lst1 with
  | [] → revappend res lst2
  | h::t → append2 t (h::res) lst2 ;;
append2 [1;2;....300000] [] [1;2;3] ;; (* 問題ない *)
```

# 末尾再帰関数まとめ

末尾再帰関数:

- ▶ 再帰関数は、通常、同等な繰返し関数より遅く、メモリも多く使う。
- ▶ 末尾再帰関数=すべての再帰呼び出しが「末尾」である再帰関数
- ▶ 末尾再帰関数は、(言語処理系が頑張れば) 繰返し関数と同様の効率(時間、メモリ)
- ▶ 再帰関数を、同等な末尾再帰関数に書き直すのは、知恵が必要なことが多い。

# 演習問題(その1)

以下の関数を末尾再帰関数として実現せよ。ただし、補助関数を使ってもよいが、使う場合はそれらもすべて末尾再帰関数とせよ。

- ▶  $f1 [7;-3;5;6] ==> 15$

関数  $f1$  は、整数のリストの要素の総和を計算するが、素朴に実装すると末尾再帰にならない。(ヒント: 補助関数を作るとよい。)

- ▶  $f2 [10;20;30] [4;5;6;7] ==> [7;36;25;14]$

関数  $f2$  は、2つのリストの同じ位置にある要素同士をそれぞれ足したリストの逆順を返す。

- ▶  $f3 (\text{fun } x \rightarrow x * 2) [1;2;3] ==> [2;4;6]$

関数  $f3$  は、`List.map` と同じ意味だが、これは末尾再帰でないので、工夫が必要である。

# 目次

① 未尾再帰

② 繼続渡し方式

# 疑問

再帰関数の中には、末尾再帰関数化が困難なものがある。

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

アルゴリズムを変更すれば、簡単に末尾再帰化できるが、、、

```
let rec fib2 n res1 res2 =
  if n = 0 then res1
  else if n = 1 then res2
  else fib2 (n - 1) res2 (res1 + res2);;
fib2 10 1 1;;
```

# 疑問

再帰関数の中には、末尾再帰関数化が困難なものがある。

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

アルゴリズムを変更すれば、簡単に末尾再帰化できるが、、、

```
let rec fib2 n res1 res2 =
  if n = 0 then res1
  else if n = 1 then res2
  else fib2 (n - 1) res2 (res1 + res2);;
fib2 10 1 1;;
```

これは、関数の意味を考えて変形してしまった。もとのアルゴリズムのままでの末尾再帰関数化は可能か？

# 継続渡し方式

発想を転換して、「途中結果」ではなく「残りの計算」を渡していく。

普通の再帰関数の例:

```
let rec fact n =
  if n = 0 then 1
  else n * (fact (n - 1)) ;;
let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n - 1) (fun x → k (n * x)) ;;
```

# 継続渡し方式

発想を転換して、「途中結果」ではなく「残りの計算」を渡していく。

普通の再帰関数の例:

```
let rec fact n =
  if n = 0 then 1
  else n * (fact (n - 1)) ;;

let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n - 1) (fun x → k (n * x)) ;;

fact_cps 3 (fun x → x)
==> fact_cps 2 (fun x → 3 * x)
==> fact_cps 1 (fun x → 3 * (2 * x))
==> fact_cps 0 (fun x → 3 * (2 * (1 * x)))
==> (fun x → 3 * (2 * (1 * x))) 1
==> 3 * (2 * (1 * 1))
==> 6
```

# 継続渡し方式関数は高階関数

```
fact_cps 3 (fun x → x)  
==> ...  
==> fact_cps 1 (fun x → 3 * (2 * x))
```

関数  $\text{fun } x \rightarrow 3 * (2 * x)$  は、残りの計算を表す。

- ▶ この時点で実行中の計算は  $\text{fact } 1$  (1 の階乗)。
- ▶ 「 $\text{fact } 1$  の計算」残りの計算は、「その結果に 3 と 2 を乗算する」。
- ▶ 「現在の計算結果に 3 と 2 を乗算する」を  $\text{fun } x \rightarrow 3 * (2 * x)$  で表す。

# 継続渡し方式関数

```
let rec power x n =
  if n = 0 then 1
  else x * (power x (n - 1)) ;;
let rec power_cps m n k =
  if n = 0 then k 1
  else power_cps m (n - 1) (fun x → k (x * m)) ;;

          power_cps 2 3 (fun x → x)
==> power_cps 2 2 (fun x → 2 * x)
==> power_cps 2 1 (fun x → 2 * (2 * x))
==> power_cps 2 0 (fun x → 2 * (2 * (2 * x)))
==> (fun x → 2 * (2 * (2 * x))) 1
==> 2 * (2 * (2 * 1))
```

# 継続渡し方式の関数は高階関数かつ多相型

継続渡し方式 (Continuation-Passing Style) の関数:

```
let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n - 1) (fun x → k (n * x)) ;;
```

- ▶ 通常の関数に，継続 (continuation, 残りの計算) を表す引数を追加．
- ▶ CPS の関数は，常に末尾再帰関数かつ高階関数である．
- ▶ CPS の関数の型は，多相型．

# 継続渡し方式の関数の型

継続は多相型，従って，継続渡し方式の関数も多相型：

```
let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n - 1) (fun x → k (n * x)) ;;

fact_cps 10 (fun x → x) ;; (* k : int → int *)
fact_cps 10 (fun x → sqrt (float x)) ;; (* k : int →
  float *)
fact_cps 10 (fun x → string_of_int x) ;; (* k : int →
  string *)
```

関数 fact\_cps の型は多相型：

$$\text{fact\_cps} : \forall \alpha. (\text{int} \rightarrow ((\text{int} \rightarrow \alpha) \rightarrow \alpha))$$

## 継続渡し方式のメリット: 制御構造 (1/2)

継続渡し方式は「ジャンプ」を表現できる。

```
let rec f lst =
  match lst with
  | [] → 0.0
  | h::t → (sqrt h) +. (f t) ;;

f [1.0; 3.0; 5.0] ;;      (* 4.968 *)
f [1.0; (-3.0); 5.0] ;;  (* おかしな値 (nan) *)
```

負の数が含まれていたら、その値を返したいが、以下はうまくいかない。

```
let rec f_bad lst =
  match lst with
  | [] → 0.0
  | h::t → if h >= 0.0 then (sqrt h) +. (f_bad t)
            else h ;;

f_bad [1.0; (-3.0); 5.0] ;;  (* -2.0 *)
```

## 継続渡し方式のメリット: 制御構造 (2/2)

継続渡し方式は「ジャンプ」を表現できる。

```
let rec f_cps lst k =
  match lst with
  | [] → k 0.0
  | h::t → if h >= 0.0 then
              f_cps t (fun x → k ((sqrt h) +. x))
            else h ;;
f_cps [1.0; 3.0; 5.0] (fun x → x) ;;          (* 4.968.. *)
f_cps [1.0; (-3.0); 5.0] (fun x → x) ;;      (* -3.0 *)
```

継続渡し方式では、ジャンプなどの制御を表せる。

- ▶ 継続  $k$  を使わずに捨てると、大域脱出 (global exit) を表現 [例外処理など]
- ▶ 継続  $k$  を 2 回以上使うと、非決定的選択 (choice) を表現 [探索問題など]

# 継続渡し方式のメリット: 中間言語として

継続渡し方式: 評価順序が一意的、かつ、中間結果に名前付け

最適化がやりやすい

(@を、なんらかの二項演算子とする)

$a @ b$  を CPS に変換:

(\* 第 1 引数から評価 \*)

```
fun k → [a] (fun x →  
    [b] (fun y →  
        k (x @ y)))
```

(\* 第 2 引数から評価 \*)

```
fun k → [b] (fun y →  
    [a] (fun x →  
        k (x @ y)))
```

ただし、[m] は式 m を CPS にしたもの。

# 継続渡し方式のまとめ

継続渡し方式:

- ▶ 継続=残りの計算
- ▶ 継続渡し方式=継続を表す「関数」を引数として渡していくスタイル
- ▶ 定理. どんな関数も、継続渡し方式に変換可能。
- ▶ 利点. 末尾再帰、制御(ジャンプなど)の表現、最適化などがやりやすい。
- ▶ 欠点. 継続を使わない末尾再帰化より、実行性能が劣る。

cf. CPS を中間言語とするコンパイラの構成:

A. Appel, Compiling with Continuations, Cambridge Univ. Press, 1991.

継続渡し方式は、関数型プログラム言語と相性が良い。

- ▶ 継続という「概念」は、どんなプログラム言語にもあるが、
- ▶ 継続をそのまま表現できるのは、高階関数と多相型がある言語のみ。
- ▶ C言語でCPSを表現するためには、継続を関数以外のデータで表現する必要がある。

## 演習問題(その2)

問題: 繰続渡し方式の関数 fact\_cps について、第1引数  $n$  を 3 とし、第2引数  $k$  を以下の関数として実行したとき、fact\_cps が呼びだされる毎、第2引数(継続を表す)とその型がどうなるか示しなさい。(n=3 なので fact\_cps は合計 4 回呼び出される。)

- ▶ `fun x -> x`
- ▶ `fun x -> string_of_int x`

また、 $k$  が上記のそれぞれの値のときに fact\_cps を OCaml で実行して、実行結果を示しなさい。

発展課題: fib 関数を継続渡し方式に変換せよ。