末尾再帰でない再帰プログラムの高速化のための最適化に関する一考察

小林 周太郎　川端 英之　弘中 哲夫

分割統治による問題解決を始めとして，プログラム記述に再帰呼び出しが用いられるものは多い．しかし，末尾呼び出しでない再帰プログラムは大規模問題の処理時にスタックオーバーフローを引き起こし得る．一方，末尾呼び出しを徹底したプログラミングは必ずしも容易ではない．これに対し我々は，末尾再帰でない再帰プログラムを，効率よく実行可能な表現に自動変換する実用的手法を検討中である．再帰プログラムの末尾呼び出し化は，CPS 変換や，スタックに保持されるデータ列をリストとして扱うなどの方法で機械的に行えるが，再帰呼び出し回数に比例するメモリ消費量を要するコードは効率的とは言い難い．また，小規模な問題の場合には素朴な記述の方が高速に処理され得る．我々は，これらを踏まえ，十分に高速で，処理の反復回数の制約無く実行可能なプログラムへの変換手法とその適用範囲について検討している．本稿では，これらの検討項目について考察する．

1 はじめに

関数プログラミングにおいては，ループ構文は嫌忌される傾向がある．関数型プログラミング言語（以下，関数型言語），とりわけ「純粋な」関数型言語にいわゆる `for` や `while` などのループ構文 †1 が強く求められない理由を探すと，

1. 関数の再帰的定義でより宣言的に書けて，
2. しかも十分高速で，
3. 高階関数を使えば一般的でモジュラーな記述ができるから，

ということのようである †2．分割統治による問題解決など，再帰呼び出しを含むプログラム記述が自然に（トップダウンに，宣言的に）行える場合も多いが，ともかく，関数型言語を使いこなすには，ループ構文を用いないプログラミングが自然にできる力が求められる．その習得は，簡単だろうか．

前述の 1. と 2. は常に両立するとは限らない．再帰呼び出しによる処理の反復は，関数呼び出しとリターンの処理にかかるコスト（スタックフレームの管理やデータ授受の処理，場合によっては命令キャッシュのヒット率低下など）に注意を払う必要がある．また，配慮のない再帰呼び出しではコールスタックが溢れ得る．

スタックオーバーフロー発生の可能性は，OCaml のような正格評価の言語を用いる場合，再帰呼び出しを末尾再帰の形式で記述することにより排除できる．しかしながら，末尾再帰を徹底したプログラミングは必ずしも容易ではない．末尾再帰による反復処理記述は goto によるループ記述に他ならず [16] [15]，明示的な再帰呼び出しが散らばるプログラムはむしろ読みにくく，ループ構文を取り入れた場合と比較してソフトウェア工学的視点から見て問題点が多いとの指摘もある [14] †3（それを踏まえてか，コンパクトな仕様が特徴的な言語である Scheme [13] にも do 構文は存在するし，ループ構文のマクロパッケージの開発

On optimization for recursive programs without tail-calls.

Shutaro Kobayashi, Hideyuki Kawabata, and Tetsuo Hironaka, 広島市立大学, Hiroshima City University.

†1 純粋な関数型言語では処理の反復（状態遷移）の制御は関数の再帰的定義に頼ってしか行えないが，ここでは構文糖衣等も含めて考えている．

†2 例えば，`https://www.quora.com/Why-dont-pure-functional-programming-languages-provide-a-loop-construct`.

†3 文献 [14] では，`map` や `fold` などの高階関数を用いても状況の改善されない場合の存在を指摘している．

も盛んになされている[14]．OCaml もループ構文を用意している†4)．再帰プログラムの末尾呼び出し化は，Continuation-Passing Style (CPS) への変換や，スタックに保持されるデータ列をリストとして扱うなどの方法で機械的に行えるが，反復回数に比例するメモリ消費量を要するコードは効率的とは言い難い．また，小規模な問題の場合には素朴な記述の方が高速に処理され得る．

関数型言語は，部品であるところの関数を合成してより大きなプログラム（関数）を構成し易いのが特徴で，前述の 1. と 2. はある意味で 3. を下支えする項目である．反復処理も高階関数の組み合わせでコンパクトに記述でき，代数的データ型のデータに対する累積計算（fold）で多くのことが記述できる．しかしながら，全てをそのように整えて記述することは現実的ではない（[9] のような研究が重要視される所以でもある）．高階関数との比較で，再帰関数での反復処理記述は「低レベル」と言われることがあるが，末尾呼び出しでの記述は更に低い水準の（機械語寄りの）プログラミングと言える†5．

我々は，末尾再帰でない再帰プログラムを，効率よく実行可能な表現に自動変換する実用的手法を探りたいと考えている．等価な変換，等価でない変換を含め，素朴な書き方の再帰プログラムを，十分に高速で，処理の反復回数の制約無く実行可能なプログラムへの変換手法とその適用範囲について考察している．本稿は，現時点での考察内容をまとめたものである．

なお，5 節で述べるように，本稿で述べる内容に関連する既存研究は豊富にある．個々の研究との対応関係の詳細な調査は我々にとって喫緊の課題である．

2 動機：関数プログラミングで感じる疑問

ここでは，本研究の動機説明のために，反復処理を再帰関数で記述することを強いられた関数プログラミング初心者が疑問を抱くであろう場面を眺める．以降の説明では関数型言語として OCaml を用いる．

2.1 関数の再帰呼び出しを用いてループを実現

関数プログラミングの入門書を紐解くと，プログラムは宣言的に記述するものだと書かれている．例えば関数 $sum\ n = \sum_{i=0}^{n} i$ は次のように記述できる：

```
let rec sum n =
  if n = 0 then 0 else n + sum (n-1)
```

このプログラムを通してユーザは次のようにコンピュータと対話できる．

```
# sum 3 ;;
- : int = 6
# sum 300 ;;
- : int = 45150
# sum 300000 ;;
Stack overflow during evaluation
 (looping recursion?).
#
```

16GB のメモリを積み 3.3GHz のプロセッサを持つコンピュータで，(この方法では) 0 から 300000 までの総和を計算できない，ということを知る．

2.2 末尾再帰と累積を用いてループを実現

再帰プログラムで多数回の反復を行う場合，末尾再帰で記述すればスタックが溢れる問題を回避できる：

```
let rec sum' n a =
  if n = 0 then a else sum' (n-1) (n+a)
```

```
# sum' 300000000 ;;
- : int -> int = <fun>
# sum' 300000000 0 ;;
- : int = 45000000150000000
#
```

0 から 300000000 の和でも計算できる．ただし，関数に余分な引数が求められる．

2.3 関数の合成によるプログラミング

関数プログラミングは，性質のはっきりしている小さい部品の組み合わせで大きなプログラム（関数）を構成し易いことがその大きな特徴である[1]．例えば，n から m までの整数を昇順に並べたリストを生成する関数 `upto` と，リストの要素の総和を計算する関数

†4 OCaml のループ構文は，破壊的代入とともに用いない限りは，冗長なプログラムの短縮表現程度にしか使えないようである．

†5 末尾呼び出しでの記述はループ不変条件を見出すことと関連が深いと思われる．末尾呼び出しで記述できることがプログラミングスキルとして重要である点に異議を挟むつもりはない．

lsum があるならば，n から m までの整数の総和を計算する関数 sum は次のように構成できる：

```
let sum n m = lsum (upto n m)
```

あるいは高階関数を用いて次のようにも定義できる：

```
let sum n m =
  List.fold_right (+) (upto n m) 0
```

これらのプログラムを動かそうとしてみる．まず，関数 upto を作る．末尾再帰で記述した関数も作っておく（upto’）．末尾再帰で記述する理由は，リストを作る仕事を任される関数 upto’ が長いリストを構築する際に，リストを格納するために要するヒープ領域が足りなくなることが心配だから，ではなく，スタックオーバーフローを避けるためである．

ここで，upto’ を局所関数を用いて書いてみる — 局所関数は便利だが，末尾再帰にするために必要となるアキュムレータの初期値の設定をユーザから意識させないようにするためだけにあるのではないはず，と信じつつ．

```
let upto’ n m =
  let rec iter n m a =
    if n >= m then a
    else iter n (m-1) (m::a)
  in iter n m []
```

```
# let l = upto 0 300000 ;;
Stack overflow during evaluation
 (looping recursion?).
# let l = upto’ 0 300000 ;;
val l : int list =
  [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; ...]
#
```

整数のリストを受け取ってその要素の総和を計算するために用いる関数も，末尾再帰でなければ長いリストを扱えない（lsum’）．

```
# lsum (upto’ 0 300000) ;;
Stack overflow during evaluation
 (looping recursion?).
# lsum’ (upto’ 0 300000) ;;
- : int = 45000150000
#
```

最後に，標準ライブラリにある高階関数を使い，「低レベル」な問題に頭を悩まされるのとは一味違うエレガントなプログラミングの体験をしようとしてみる．

```
let rec loop arg =
  if is_final arg then r arg
  else f (d1 arg) (loop (d2 arg))
```

図 1　再帰関数による反復処理の一般形

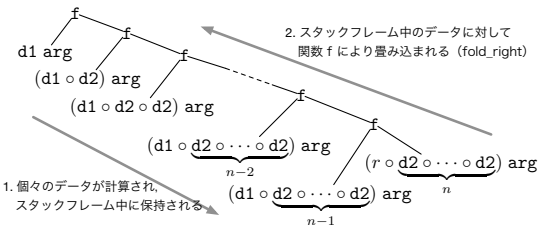

図 2　再帰関数による反復処理のデータフロー

```
# List.fold_right (+) (upto’ 0 300000) 0;;
Stack overflow during evaluation
  (looping recursion?).
# List.fold_left (+) 0 (upto’ 0 300000);;
- : int = 45000150000
#
```

標準ライブラリの関数を用いても，躓きに事欠かない．関数プログラミングではスタック溢れへの配慮が大変重要であるということが強く認識される †6．

実際のところ，現実的なプログラミングにおいてスタックの溢れで困る場面がどの程度あるのだろうか．我々の知る限りでは，この点に関する定量的な評価は存在しない．小規模な反復しかしないプログラムでは素朴な再帰関数記述の方がむしろ高速であるし，実際，OCaml の標準ライブラリにある「反復処理を行う関数」のいくつか（List.map や List.fold_right など）の実装が，スタックオーバーフローを引き起こし得るにもかかわらず末尾再帰になっていないのは，これらの関数の扱われ方を総合的に判断しての結果だと思われる．ともあれ，再帰関数での反復処理の記述にあたっては，スタックオーバーフローへの配慮は無視することはできない．

3　考察：再帰関数による反復処理について

本節では，再帰関数定義によって記述された反復処理の挙動を分析し考察する．3.1 節では，素朴な関数

†6　遅延評価の関数型言語では事情は異なる．また，foldr/build ルール（より一般的には hylo fusion）により中間的なデータ構造構築の必要性を排除する最適化手法が実現されている [8] [17].

の再帰的定義による反復処理の一般形を示し，様々な反復アルゴリズムを例に挙げてその挙動について述べる．続いて，末尾再帰への機械的な変換（3.2，3.3節），および，特殊な場合の変換（3.4，3.5節）について述べる．3.6節ではコンパイラによる最適化について言及する．

3.1 再帰関数で記述されるループとその挙動

反復処理を関数の再帰的定義で記述するときの一般的な表現として，図1に示すプログラムを考える．図1のプログラムは，引数 arg に基づいて得られる値の列に対して関数（二項演算）f を用いて畳み込み（fold）を行う．図1のプログラムの挙動は図2のように描ける．すなわち，i 回目（$i = 1, 2, \cdots$）の loop の呼び出し時にスタックフレーム中に

$$(\mathtt{d1} \circ \underbrace{\mathtt{d2} \circ \cdots \circ \mathtt{d2}}_{i-1})\ \mathtt{arg}$$

が格納され，i 回目の loop の実行時に

$$\mathtt{is\_final}\ (\underbrace{(\mathtt{d2} \circ \cdots \circ \mathtt{d2}}_{i-1})\ \mathtt{arg}) = \mathrm{true}$$

の成立が確認された時点で畳み込みが行われる．関数 d1 および d2 は，それぞれ，畳み込みに用いる値の調整をする関数，および，反復のたびに arg を「減らす」関数に対応する．関数 is_final，r，d1，d2 の引数，および，d2 の戻り値は，arg と同じ型を持つ．

以下，図1で表現される計算の例をいくつか挙げる．

総和計算 関数 is_final，r，d1，d2，および f を次のように定義すると，図1の関数 loop は2.1節の sum と同じものになる（int → int）．

```
is_final = fun x → x = 0
       r = fun x → x
      d1 = fun x → x
      d2 = fun x → x − 1
       f = fun x y → x + y
```

リストの長さ 次の定義により，関数 loop はリストの長さを計算する関数となる（α list → int）．

```
is_final = fun x → x = [ ]
       r = fun x → 0
      d1 = fun x → 1
      d2 = fun x → List.tl x
       f = fun x y → x + y
```

ベキ乗 次のように定義すれば，関数 loop は，整数のペア (n, m) を受け取り n の m 乗（n^m）を計算する関数となる（int * int → int）．

```
is_final = fun (n, m) → m = 0
       r = fun (n, m) → 1
      d1 = fun (n, m) → n
      d2 = fun (n, m) → (n, m − 1)
       f = fun x y → x * y
```

リストの結合 次のように定義すれば，関数 loop は，リストのペア (l_1, l_2) を受け取り結合されたリスト $l_1@l_2$ を計算する関数となる（α list * α list → α list）．

```
is_final = fun (l1, l2) → l1 = [ ]
       r = fun (l1, l2) → l2
      d1 = fun (l1, l2) → List.hd l1
      d2 = fun (l1, l2) → (List.tl l1, l2)
       f = fun x y → x::y
```

フィボナッチ数列 次のように定義すれば，関数 loop は，非負整数 $n > 0$ を受け取りフィボナッチ数列の n 番目の値と $n-1$ 番目の値のペアを計算する関数となる（int → int * int）．

```
is_final = fun n → n = 1
       r = fun n → (1, 1)
      d1 = fun n → (1, 1, 1, 0)
      d2 = fun n → n − 1
       f = fun (x, y, z, w) (p, q)
             → (x * p + y * q, z * p + w * q)
```

関数 f は2行2列の行列ベクトル積であるが，これを2行2列の行列積で表現するなども可能である（それに応じて r も変更が必要）．また，この定義には最適化の余地が多々ある．例えば，f の第1引数は常に $(1, 1, 1, 0)$ になるので，関数の本体は (p + q, p) にできる．

リストへの要素の挿入 整列されているリストに新たな要素を適切な位置に挿入する関数（挿入ソートで用いられる）が実現できる．関数 loop は，整数と（昇順に整列済みの）整数のリストのペアを受け取り，一つの整列済みのリストを返す

```
let rec loop arg k =
  if is_final arg then k (r arg)
  else loop (d2 arg)
            (fun x -> k (f (d1 arg) x))
```

図 3　CPS による末尾再帰での反復処理

$(\alpha * \alpha\,\mathtt{list} \to \alpha\,\mathtt{list})$.

```
is_final = fun (e,l) → l = []
                       || e < List.hd l
       r = fun (e,l) → if l = [] then [e]
                       else (e::l)
      d1 = fun (e,l) → List.hd l
      d2 = fun (e,l) → (e,List.tl l)
       f = fun h t → h::t
```

最大公約数　次のように定義すれば，関数 loop は，二つの非負整数 (n, m) を受け取り両者の最大公約数を返す関数となる（int * int → int）.

```
is_final = fun (n,m) → m = 0
       r = fun (n,m) → n
      d1 = fun (n,m) → 0
      d2 = fun (n,m) → (m, n mod m)
       f = fun x y → y
```

非斉次線形差分方程式　次のように定義すれば，関数 loop は，非負整数 n を受け取り，非斉次線形差分方程式 $a_{n+1} - 2a_n - 2n + 3 = 0, a_0 = 4$ の第 n 項 a_n を求める関数となる（int → int）.

```
is_final = fun n → n = 0
       r = fun n → 4
      d1 = fun n → 2 * n − 3
      d2 = fun n → n − 1
       f = fun x y → x + 2 * y
```

3.2　CPS による末尾再帰

図 1 の関数 loop は末尾再帰ではない †7．結果として，loop はスタックオーバーフローを生じさせ得る．

スタックオーバーフローの危険性を排除するナイーブな方法として，元の関数に対する CPS 変換が挙げられる．一般の再帰関数に対して CPS 変換を施すと全ての関数呼び出しは末尾呼びに変更される．例えば図 1 の関数 loop に対する CPS 版の定義は図 3 のようになる．3.1 節に挙げた図 1 の具体化例はいずれも，図 3 においても同じ振る舞いをする（初期の継続として関数 fun x → x を渡すと仮定）.

†7　関数 f 次第では末尾再帰になり得る．実際，3.1 節の最大公約数の例では，loop が末尾再帰になる．

```
let rec loop arg acc =
  if is_final arg then
    fold_left (fun x y -> f y x)
              (r arg) acc
  else loop (d2 arg) ((d1 arg) :: l)
```

図 4　リストを用いた累積による反復処理

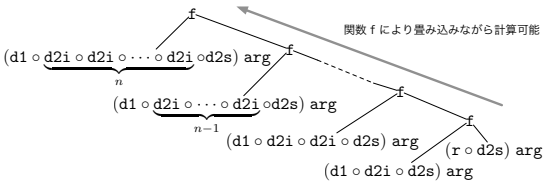

図 5　逆方向への累積計算のデータフロー

CPS 変換は一般にはプログラムの最適化（例えば実行速度の向上）には繋がらない．図 1 のプログラムの実行時にスタックフレーム中に保持されるデータ

$$\mathtt{d1}\ \mathtt{arg},\ (\mathtt{d1} \circ \mathtt{d2})\ \mathtt{arg},\ (\mathtt{d1} \circ \mathtt{d2} \circ \mathtt{d2})\ \mathtt{arg},\ \cdots$$

は，図 3 のプログラムの実行時にはヒープ領域に構成される関数クロージャの連鎖の内部に格納される．

3.3　リストを用いた累積表現

図 1 の関数の実行時にスタックフレームに格納されるデータを，明示的にリストにつないでヒープ領域に格納することができる．図 4 にその実現例を示す．計算処理がリストの構築（累積）と fold の二段階に明確に別れて，それぞれスタックフレームを消費することなく処理できる．

3.1 節に挙げた図 1 の具体化例はいずれも，図 4 においても同じ振る舞いをする．

3.4　逆方向への累積表現への書き換え

図 2 のデータフローグラフは，一見すると，左上から右下まで徐々に葉の値の計算を進めつつリストを構成し，続いて，右下から左上までリストを走査しながら累積計算（ここでは fold）をする必要があるように見える．しかしながら，もし任意の m について

```
let loop arg =
  let d = d2s arg in
  let p = is_final_i (d2i arg) in
  let rec loop' a acc =
    if p a then acc
    else loop' (d2i a) (f (d1 a) acc)
  in loop' (d2i d) (r d)
```

図 6　逆方向への累積を行う末尾再帰記述

$\underbrace{\texttt{d2} \circ \texttt{d2} \circ \cdots \circ \texttt{d2}}_{m}$ arg が短時間で（$O(m)$ より小さいコストで）計算可能であるならば，最初のステップでリストを構成することなく右下から左上まで計算を進めることができる．

一方，仮に d2 の逆関数 d2i が使用可能であるとすると，以下のように定義される関数 d2s を用いて，図 2 のデータフローグラフは図 5 のように表現できる．

$$\texttt{d2s} = \underbrace{\texttt{d2} \circ \texttt{d2} \circ \cdots \circ \texttt{d2}}_{n}$$

図 5 から明らかなように，この場合も，リストの中間表現を作ることなく，木の右下から左上に向かって計算を進めることができる．図 6 はこれを行うプログラムの例である．

3.1 節に挙げた図 1 の具体化例のうちいくつかは，図 6 を用いた計算ができる．

総和計算　関数 is_final_i，d2i，および d2s を次のように定義すると，図 6 の関数 loop は 2.1 節の sum と同じものになる（int → int）．

$$\begin{aligned} \texttt{is\_final\_i} &= \texttt{fun x y} \to \texttt{x} = \texttt{y} \\ \texttt{d2i} &= \texttt{fun x} \to \texttt{x} + 1 \\ \texttt{d2s} &= \texttt{fun x} \to 0 \end{aligned}$$

ベキ乗　次の定義により，図 6 の関数 loop は整数のペア (n, m) を受け取り n の m 乗（n^m）を計算する関数となる（int * int → int）．

$$\begin{aligned} \texttt{is\_final\_i} &= \texttt{fun x y} \to \texttt{x} = \texttt{y} \\ \texttt{d2i} &= \texttt{fun (n, m)} \to \texttt{(n, m} + 1\texttt{)} \\ \texttt{d2s} &= \texttt{fun (n, m)} \to \texttt{(n, 0)} \end{aligned}$$

フィボナッチ数列　次のように定義すれば，図 6 の関数 loop は，非負整数 $n > 0$ を受け取りフィボナッチ数列の n 番目の値と $n-1$ 番目の値のペアを計算する関数となる（int → int * int）．

$$\begin{aligned} \texttt{is\_final\_i} &= \texttt{fun x y} \to \texttt{x} = \texttt{y} \\ \texttt{d2i} &= \texttt{fun n} \to \texttt{n} + 1 \\ \texttt{d2s} &= \texttt{fun n} \to 1 \end{aligned}$$

非斉次線形差分方程式　次のように定義すれば，図 6 の関数 loop は，非負整数 n を受け取り，非斉次線形差分方程式 $a_{n+1} - 2a_n - 2n + 3 = 0, a_0 = 4$ の第 n 項 a_n を求める関数となる（int → int）．

$$\begin{aligned} \texttt{is\_final\_i} &= \texttt{fun x y} \to \texttt{x} = \texttt{y} \\ \texttt{d2i} &= \texttt{fun n} \to \texttt{n} + 1 \\ \texttt{d2s} &= \texttt{fun n} \to 0 \end{aligned}$$

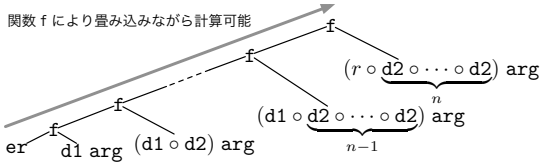

図 7　f が結合的で左単位元を持つ場合の処理の流れ

```
let loop arg =
  let rec loop' a acc =
    if is_final a then (f acc (r a))
    else loop' (d2 a) (f acc (d1 a))
  in loop' arg er
```

図 8　f が結合的で左単位元を持つ場合の末尾再帰記述

3.5　リストを構築しない末尾再帰

関数 f が結合的でかつ左単位元を持つ場合，すなわち，f a (f b c) = f (f a b) c かつ任意の a について f er $a = a$ となる er が存在する場合，図 2 のデータフローグラフに基づく計算は図 7 のように行うことができる．この場合，図 8 のように，中間的なデータを構成することなく，末尾再帰により，計算できる．

3.1 節に挙げた図 1 の具体化例のうちいくつかは，図 8 を用いた計算ができる．

総和計算　er = 0 とすると，図 8 の関数 loop は 2.1 節の sum と同じものになる（int → int）．

リストの長さ　er = 0 とすると，図 8 の関数 loop はリストの長さを計算する関数となる（α list → int）．

ベキ乗　er = 1 とすると，図 8 の関数 loop は，整

数のペア (n, m) を受け取り n の m 乗（n^m）を計算する関数となる（`int * int → int`）．

最大公約数 `er = 0` とすると，図 8 の関数 `loop` は，二つの非負整数 (n, m) を受け取り両者の最大公約数を返す関数となる（`int * int → int`）．

3.6 プログラム変換による最適化の可能性

3.1 節で述べた形式のプログラムは末尾再帰ではなく，スタックオーバーフローを引き起こし得る．スタックフレームの大量消費を避けたいだけであれば，3.2 節や 3.3 節で述べた形式への変換を機械的に行えば済む．しかしその場合，所要記憶量に関する改善は見込めず，高速化は望めない．一方で，プログラムの構成要素が分析でき，関数 `d1`，`d2`，`f` の性質が単純であることを見出せれば，3.4 節や 3.5 節で述べた形式への書き換えによって，スタックオーバーフローの心配を排除しつつ高速化を実現できる．

再帰プログラムを 3.1 節の形式として捉える際の，関数 `d1` や `d2`，`f` の抽出の仕方は，一意ではない．例えば 3.1 節で挙げたフィボナッチ数列のプログラム例では `f` を行列とベクトルの積で表したが，2 行 2 列の行列積でも表現できる．こうすると，`f` は結合的で左単位元を持つので，3.5 節で述べた形式への書き換えができる．

`d1` が定数関数の場合には，図 2，図 5，および図 7 において反復のたびに計算される値は全て等しくなるので，反復ごとに値を計算する必要がなく，それに応じた最適化ができる．（`d2` $=$ `fun x` $\rightarrow$ `x` の場合もそうだが，それは無限ループを意味する）．

`f` が可換な場合にも計算順序の変更が可能であり，リストを作ってから fold するような 2 段階の処理は不要になる．ただし，計算順序の変更は，浮動小数点データ型を用いた演算では一般にはプログラムの等価変換とは言えない．また，整数演算のみを用いる場合でも，オーバーフローなどの演算例外が生じるか否かなどの挙動の違いを生じさせ得るため，プログラム変換をすべきか否かの判断は慎重にする必要がある．

再帰で構成されたループの最適化を実現するには，`f`，`d1`，`d2`，`is_spec` を自動抽出した上で，3.4 節や 3.5 節で述べた形式への書き換えができるか否かの判断，そして，書き換えた場合の性能の変化の見積もりができる必要がある．

```
type ('a, 'b, 'c) pr_spec =
  { is_final: 'a -> bool;
    r: 'a -> 'b;
    d1: 'a -> 'c;
    d2: 'a -> 'a;
    f: 'c -> 'b -> 'b }

let genPR_spec spec =
  let rec loop arg =
    if spec.is_final arg
      then spec.r arg
      else spec.f (spec.d1 arg)
                  (loop (spec.d2 arg))
  in loop

let sum_spec =
  { is_final = (fun i -> i = 0);
    r = (fun s -> s);
    d1 = (fun x -> x);
    d2 = (fun x -> x - 1);
    f = (fun x y -> x + y) }

let sum = genPR_spec sum_spec
```

図 9　実測に用いた再帰プログラム（図 1 に関する部分）

4 実験：再帰で構成されたループの実行速度

再帰で構成されたループについて，記述の仕方の違いによるパフォーマンスの変化を調査するため，3.1 節で列挙した反復処理の例に対し，実行に要する時間を測定した．プログラムは図 9 のように（明示的な最適化をすることなく）記述した．コンパイルには OCaml 4.05.0 の `ocamlopt` を使用し，プロセッサ 3.3GHz Core i7，メモリ 16GB，macOS 10.12.6 という環境で実行した．計時には OCaml の `Sys.time` を用いた．結果は表 1 に示す通りである．

表 1 のデータの中には反復回数が少ないものも多い（† を付けたもの）．また，測定に用いたプログラムは，実際にプログラム変換を適用したものではなく，図 9 のような方法で生成したものであり，十分にチューニングされたものとは言い難い．しかしながら，表 1 から，機械的に CPS 変換を施すこと（図 3）や中間データの格納場所をヒープに移すだけの変換（図 4）には高速化の効果はほぼ無いということが，

表 1　再帰で構成されたループの実行に要する時間の調査結果

アプリケーション	図 1	図 3 (CPS)	図 4 (ヒープ)	図 6 (逆関数)	図 8 (順序変更)	備考
総和	0.78	4.20 (5.4)	2.94 (3.8)	0.72 (0.9)	0.72 (0.9)	0 から 10 万までの和，1000 回.
リストの長さ	1.03	5.87 (5.7)	3.32 (3.2)		0.64 (0.6)	長さ 10 万のリスト，1000 回.
ベキ乗 †	0.28	0.34 (1.2)	0.34 (1.2)	0.79 (2.8)	0.28 (1.0)	3 の 30 乗，100 万回.
リストの連結	8.58	13.66 (1.6)	7.69 (0.9)			長さ 10 万のリスト 2 つ，1000 回.
フィボナッチ数列 †	0.79	1.01 (1.3)	0.98 (1.2)	0.77 (1.0)		fib 100，100 万回.
要素挿入	3.02	5.74 (1.9)	3.23 (1.1)			リスト長 10 万，中央に挿入，1000 回.
最大公約数 †	0.09	0.10 (1.1)	0.10 (1.0)		0.08 (0.9)	gcd(51,100)，100 万回.
線形差分 †	0.37	0.48 (1.3)	0.47 (1.3)	0.38 (1.0)		50 番目の要素，100 万回.

数値は `Sys.time` で計測した実行時間（単位は秒）．第 3, 4, 5, 6 カラムのカッコ内の数値は，第 2 カラムの数値に対する比．アプリケーションは 3.1 節で列挙したもの．プログラムは，3 節で述べた各方式のものを図 9 のような記述で構成．空欄は各方式で対応していないもの．† は「反復処理」の回数が 100 回以下のもの．備考欄の回数は，時間測定の際に同一プログラムの実行を何回繰り返したか．

大雑把な傾向として言える．一方で，関数 `f` の性質を踏まえて計算順序を変更することは大幅な高速化に繋がる可能性があると言えそうである．

5　関連研究

関数プログラミングは，単純な部品である関数を組み合わせて大きなプログラム（関数）を構成することを基本的な方法論として持つ．`fold` や `map` 等の高階関数を組み合わせて，帰納的に定義された代数的データ型のデータを処理する考え方は，様々なアルゴリズムの簡潔な実装を可能にする [1].

プログラム中での関数（手続き）呼び出しの「遅さ」に関する議論は古くからある [15]．また，実行速度向上のための再帰プログラムに対するプログラム変換の研究も永く行われている [5] [3].

部品の組み合わせで構成されたプログラムは簡潔で見通しの良いものとなり易いが，部品間でのデータのやりとりに無駄が生じる場合があり，これを排除するための手法は広く研究されている．関数プログラミングにおいては中間データは典型的にはリストや木であり，プログラム変換によって不要な木を取り除く手法は「森林伐採（deforestation）」[18] として知られる．初期には，木の無いプログラム同士の組み合わせで木の無いプログラムを構成するなどの手法が検討された [18]．その後，リストを含む代数的データ型に対する累積計算に注目してデータの生産者と消費者がプログラム中で露わになっている部分を「foldr/build ルール」によって変換する手法が研究された [8] [12]．この方法はさらに hylo fusion として一般化されている [17]，一方，任意の再帰プログラムを foldr/build 形式に変換する方法も研究されている [10] [4] [11] [9]．文献 [9] では，全ての実践的な再帰アルゴリズムを構造的な hylomorphism に変換する方法を示している．これらのプログラム変換手法は，複雑なものからシンプルなものまで様々である．実測に基づく評価も行われている [8] [4] [21] [11].

森林伐採を含め，プログラムの定式化や変換の技法の発展には構成的アルゴリズム論 [6] [19] の考え方の寄与が大きい．また，プログラムの定式化においては，圏論的なアイデアが広く活用されている．一般的な代数的データ型に対する累積計算は定式化され [2] [7]，グラフの一般的探索を行う関数の再帰的定義に hylo fusion が適用できることも示されている [20]．大規模グラフ処理向けの並列頂点主体グラフ処理の定式化も行われている [22].

6　おわりに

本稿では，再帰プログラムの高速化のための実用的なプログラム変換の実現を目的とし，具体例を挙げながら，再帰プログラムの挙動の分析と最適化手法の検討事項について述べた．

関数型言語によるプログラムの性質は既存研究で

よく調べられており，累積計算の定式化やプログラム変換による最適化などについての文献は豊富にある．本稿で述べた内容と深く関連する話題について，深い検討が各方面でなされている様子が伺える．

中間データの排除や並列化などのプログラム変換においては一般に，プログラムの意味を変えないことが大前提とされる．しかしながら，高水準言語によるプログラミングでは，低レベルな視点での挙動を保つ必要が無い場面も多いと考えられる（例えば「結果のリストがソートされていればよい」とか「ソートは安定ソートでなくてもよい」など）．こういった仕様の「緩さ」や許容範囲を踏まえて，等価でない変換も考慮しつつ，実行速度向上のためのプログラム変換を実現する方法について，引き続き検討したい．

謝辞 本研究は日本学術振興会学術研究助成基金助成金（課題番号：17K00106）の助成を受けている．

参 考 文 献

[1] Bird, R.: *Pearls of Functional Algorithm Design*, Cambridge University Press, （山下伸夫訳, 関数プログラミング 珠玉のアルゴリズムデザイン, オーム社, 2014） edition, 2010.

[2] Bird, R., de Moor, O., and Hoogendijk, P.: Generic programming with relations and functors, *Journal of Functional Programming*, Vol. 6, No. 1, pp. 1–28, 1996.

[3] Burstall, R. M. and Darlington, J.: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, Vol. 24, No. 1, pp. 44–67, 1977.

[4] Chitil, O.: Type Inference Builds a Short Cut to Deforestation, *ICFP'99*, 1999.

[5] Darlington, J. and Burstall, R. M.: A system which automatically improves programs, *Acta Informatica*, Vol. 6, No. 1, pp. 41–60, 1976.

[6] Fokkinga, M. M.: *Law and Order in Algorithmics*, PhD Thesis, Dept. Inf. University of Twente, 1992.

[7] Gibbons, J.: Generic Downwards Accumulations, *Science of Computer Programming*, pp. 37–65, 2000.

[8] Gill, A., Launchbury, J., and Peyton Jones, S. L.: A short cut to deforestation, *FPCA '93 Proceedings of the conference on Functional programming languages and computer architecture*, 1993.

[9] Hu, Z., Iwasaki, H., and Takeichi, M.: Deriving structural hylomorphisms from recursive definitions, *ICFP '96 Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pp. 73–82, 1996.

[10] Launchbury, J. and Sheard, T.: Warm Fusion: Deriving Build-Catas from Recursive Definitions, *FPCA '95 Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pp. 314–323, 1995.

[11] Németh, L.: *Catamorphism-Based Program Transformations for Non-Strict Functional Languages*, PhD Thesis, University of Glasgow, 2000.

[12] Sheard, T. and Fegaras, L.: A Fold for All Seasons, *FPCA '93 Proceedings of the conference on Functional programming languages and computer architecture*, pp. 233–242, 1993.

[13] Shinn, A., Cowan, J., and Gleckler, A. A.(eds.): *Revised^7 Report on the Algorithmic Language Scheme*, 2013.

[14] Shivers, O.: The Anatomy of a Loop: A story of scope and control, *ICFP'05*, 2005.

[15] Steele Jr., G. L.: Debunking the "Expensive Procedure Call" Myth, *AI Memo 443, MIT AI Lab*, 1977.

[16] Steele Jr., G. L. and Sussman, G. J.: LAMBDA: The Ultimate Imperative, *AI Memo 353, MIT AI Lab*, 1976.

[17] Takano, A. and Meijer, E.: Shortcut Deforestation in Calculational Form, *FPCA '95 Proceedings of the seventh international conference on Functional programming languages and computer architecture*, 1995.

[18] Wadler, P.: Deforestation: Transforming programs to eliminte trees, *Theoretical Computer Science*, Vol. 73, pp. 231–248, 1990.

[19] 岩崎英哉: 構成的アルゴリズム論, コンピュータソフトウェア, Vol. 15, No. 6, pp. 57–70, 1998.

[20] 篠埜功, 胡振江, 武市正人: グラフの探索関数の再帰的定義と変換, コンピュータソフトウェア, Vol. 17, No. 3, pp. 2–19, 2000.

[21] 尾上能之, 胡振江, 岩崎英哉, 武市正人: プログラム融合変換の実用的有効性の検証, コンピュータソフトウェア, Vol. 17, No. 3, pp. 81–85, 2000.

[22] 森畑明昌: 頂点主体グラフ処理の構成的アルゴリズム論に基づく 定式化, 日本ソフトウェア科学会第 *33* 回大会講演論文集, 2016.