プログラミング 10−10
（1996. 11. 1）

# Shortcut Deforestation におけるコストの比較

西田誠幸, 都倉信樹

大阪大学大学院基礎工学研究科情報数理系専攻ソフトウェア科学分野

**あらまし** 関数プログラミングでは小さな基本的な関数の組合せで大きなプログラムを構築する. 等式(変換規則) $f = g$ は $f$ と $g$ が等価であることを意味するが, プログラム $P$ 中で, これらの変換規則の左辺に相当する部分式を右辺に置き換える操作 $P \Rightarrow P'$ をプログラム変換という. 通常プログラム変換は, より高速なプログラムの導出を目的として行われる. A.Gill らによる Shortcut Deforestation は関数間で中間データ構造を受け渡しするようなプログラムを変換して, 中間データ構造を伴わないプログラムを導出する手法である. 本稿では関数プログラムの先行評価におけるコスト評価手法を用いて, Shortcut Deforestation の適用前後のプログラムのコストを評価比較することにより, Shortcut Deforestation の効率の改善度を量り, 中間データ構造の除去がプログラムの効率改善にどの程度貢献するかを述べる.
**キーワード** 関数プログラム, Shortcut Deforestation, プログラム変換, コスト

# Cost Analysis for Shortcut Deforestation

Seikoh NISHITA and Nobuki TOKURA

Division of Informatics and Mathematical Science
Graduate School of Engineering Science
Osaka University

**abstract** Program derivation for functional programs is to transform a program to other equivalent programs by local transformation. The goal of program derivation is to derive more efficient programs, i.e. to improve the efficiency of programs. Shortcut Deforestation , which is one of program derivation techniques, improves the efficiency by removing many of the intermediate data structures. In this paper, we describe the relation between the intermediate data structures removal in Shortcut Deforestation and the improvement of programs' efficiency. To estimate the program's efficiency , we use our method to evaluate costs for functional programs.

**keyword** functional program, Shortcut Deforestation, program derivation, cost

## 1. まえがき

一般に関数プログラミングでは小さな関数を組み合わせて大きなプログラムを構成することが行なわれる. この方法はプログラムの可読性を高め, さらにプログラムのモジュール化により再利用にも役立つ. しかしプログラム中の小さな関数の間でリスト(中間リスト)が受け渡される場合, プログラムを実行すると受け渡される中間リストの生成, リストの成分の読みとりが, 実行効率の低下を引き起こす.

中間リストなどの中間データ構造により低下する効率を改善するために, P.Wadler は, プログラム変換により中間データ構造を含まないプログラムを導出する手法, Deforestation を提案した [2]. しかし Deforestation では対象となるプログラムに対して treeless と呼ばれる性質を持つ一階のプログラムであるという制約を置いており, この制約を満たさないプログラムについては Deforestation が停止しなくなるなどの欠点を持つ.

これに対し, Gill らは削除する中間データ構造を中間リストのみと限定した Shortcut Deforestation(以下 SD 変換) を提案した [1]. SD 変換では削除する中間データ構造をリストに限定することによって, Deforestation では存在したプログラムに対しての制約を外すことができた. SD 変換ではプログラムに対して局所的な変換を繰り返すことによって中間リストを除去する.

これら一連のプログラムの変換は中間データ構造を除去することによって, より効率のよいプログラムを導出することを目的としている. 文献では各変換によりプログラム中の中間リストが減少することが述べられてはいるが, では中間データ構造の削除によって, はたしてどの程度効率は改善されるのであろうか. この疑問に答える報告として文献 [4] がある. 同文献では簡約段数を効率の評価尺度としてプログラムの効率の評価を行なっており, 形式的にいくつかのプログラムについて SD 変換による効率の改善度を報告している.

簡約段数以外の評価尺度として, アルゴリズムの分野などでよく用いられる時間計算量がある. 時間計算量に関しては既に本著者は文献 [3] においてプログラムを実行することなく形式的にコスト評価を行なう手法を提案しており, これを用いることによって, SD 変換に対する時間計算量を評価尺度とした解析を行なうことに適用することを試みる.

そこで本稿では, 先行評価における逐次時間計算量 (コスト) を評価する一つの手法を用いて, SD 変換の前後のコストを評価し, 変換前後でプログラムの効率がどのように変化するかを解析する. これにより SD 変換の有用性を明らかにする.

## 2. Shortcut Deforestation

Shortcut Deforestation を導入するのに A.Gill らはリストを構成する cons (:) の効率の悪さをまず主張している. cons の実行には, リストの要素を保存するための記憶領域の確保, リスト要素の格納, 逆にリストを参照する際生じるリスト要素の取り出し, 確保した領域の解放などが内在している. 中間リストを含むプログラムを実行するとリストを構成するのに cons を繰り返し行なうため, 中間リストを伴わない等価なプログラムに比べ効率が悪い. そこで中間リストを伴わないプログラムを導出するため, SD 変換ではプログラムにおいてリストを生成/消費する関数を全て build /foldr という決められた関数で書き表す. これによりプログラム中で中間リストを生成消費する部分プログラムを検出し, 局所変換により効率の良いプログラムを導出する.

値 $a$ の型が $\alpha$ であることを $a :: \alpha$ と記述することにすると, 関数 build は次の式で定義される.

$$\begin{aligned} &\text{build} :: ((\alpha \to \beta \to \beta) \to \beta \to \beta) \to \text{List } \alpha \\ &\text{build } g = g\ (:)\ \mathit{nil} \end{aligned}$$

List $\alpha$ は要素型 $\alpha$ のリスト型を示す. リスト $x ::$ List $\alpha$ に対して, build $g = x$ が成り立つとき, 関数 $g$ は $x$ の構成子 (:) と $\mathit{nil}$ を $\lambda$ 抽象した関数である. build は $g$ に (:) と, $\mathit{nil}$ を適用することによって, 元のリスト $x$ を得る.

ここで List $\alpha$ が多重リストの場合について考える. 多重リストとはリストを要素に持つリストのことで, ここでは整数などの基底型 $A$ のリスト $u ::$ List $A =$ List $^{(1)}A$ を特に 1 重リスト, $l$ 重リスト ($l = 1, 2, \ldots$) $v ::$ List $^{(l)}A$ を要素にもつリスト $w ::$ List $^{(l)}$ を $(l+1)$ 重リストと呼び, これらを合わせて多重リストと呼ぶことにする. 多重リスト build $g = x ::$ List $^{(l)}A$ について build が適用する (:), $\mathit{nil}$ の型は

$$\begin{aligned} (:) \quad &:: \text{List } ^{(l)}A \to \text{List } ^{(l+1)}A \to \text{List } ^{(l+1)}A \\ \mathit{nil} \quad &:: \text{List } ^{(l+1)}A \end{aligned}$$

となるが, この式は build が適用する (:) と $\mathit{nil}$ が (要素リストを構成する構成子ではなく) List $^{(l+1)}A$ 型のリストを構成することを意味する. この意味で build の (:), $\mathit{nil}$ を最外リスト構成子と呼ぶことにする.

多重リスト $x$ の成分リストに属するものを含め全てのリスト構成子を build を用いて表すには, 成分リストそれぞれを build により書き換える. このようにリスト $x$ を build を用いて記述することによってリストの生成が明示される.

関数 foldr は次の式で定義される畳み込みの演算である.

$$\begin{aligned}&\text{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to \text{List}\ \alpha \to \beta\\&\text{foldr}\ k\ z\ nil\ \ = z\\&\text{foldr}\ k\ z\ (a : x) = k\ a\ (\text{foldr}\ k\ z\ x)\end{aligned}$$

中間リストを伴わないプログラムを導出するための foldr/build 規則 (等式) を次に示す.

$$\text{foldr}\ \ k\ z\ (\text{build}\ g) = g\ k\ z$$

"foldr は (:), $nil$ をそれぞれ $k, z$ に置き換える働きを持つので, build が $g$ に適用する (:), $nil$ を $k, z$ に置き換えるのと等価である." これが等式の直観的な意味である.

SD 変換は 一般に次のような 4 つの手順で行なわれる.

**手順 1** リストを扱う関数を build ,foldr を用いた定義に書き換える.

**手順 2** リストを扱う関数を unfold する.

**手順 3** foldr/build 規則を適用する.

**手順 4** build, foldr を unfold する.

build ,foldr を用いた定義をリストを扱う各関数で用意することによって, SD 変換は機械的に行なうことができ, 文献 [1] では SD 変換を関数型言語 Haskell のコンパイラへ組み込んだことが報告されている.

## 3. コスト評価の指針

SD 変換のコストを解析するため本稿では文献 [3] のコスト評価法を用いる. これは関数プログラムを先行評価する時間計算量を評価する方法であり, リストの大きさ (サイズ) により決定される時間計算量の評価を行なう.

### 3.1 構造の大きさ

リストの構造の大きさをサイズと呼ぶことにする. ここでは DS 変換についてコストの評価をするので, 変換で扱うリストの構造の大きさに着目する. すなわち, DS 変換についてリストの構造の大きさに依存するコストを評価することで, DS 変換の効率改善の度合を量ることにする.

**[定義 1]** サイズの型 Size を整数型とする.

$$\text{Size} = \mathbb{N}$$

$l$ 重リストからリスト中の全要素数を返す関数を $\text{size}_l$ とする.

$$\text{size}_l :: (\text{List}\ ^{(l)} A) \to \text{Size}$$

関数 $\text{size}_l$ を次の式で定義する.

$$\begin{aligned}&\text{size}_l(nil) &&= 0\\&\text{size}_0 a &&= 1\\&\text{size}_{l+1}(b : x) &&= \text{size}_l b + \text{size}_{l+1} x\\&&&\qquad (l = 0, 1, \ldots)\end{aligned}$$

ただし $a$ は基底型の値 ($a :: A$), $b$ は $l$ 重リスト ($b :: \text{List}\ ^{(l)}$) $x$ は $(l+1)$ 重リスト ($x :: \text{List}\ ^{(l+1)}$) である.

特に $\text{size}_1$ を size と略記する. □

定義の $\text{size}_l$ は $l$ 重リスト中の全要素数を返す関数である. $(l+1)$ 重リストの各要素リストは $l$ 重リストであり, 各要素リストの全要素数を $\text{size}_l$ で求めそれらの総和をとることによって $(l+1)$ 重リスト中の全要素数を求めることができる. とくに size はリストの要素数, すなわちリストの長さを返す関数である. サイズは単なる長さとは異なりコストに影響を及ぼす引数の性質を表すものなので, ここではリストの長さとリストのサイズを区別することにする.

### 3.2 コスト評価関数

関数プログラムを先行評価, すなわちプログラムのもっとも内側の簡約項から順に簡約する評価法のもとで要する時間計算量を評価する関数を cost とする. 関数 cost はプログラムとプログラムの引数から時間計算量を返す関数である. すなわち,

$$\text{cost} :: (A \to B) \to A \to \mathbb{N}$$

である. リストのサイズに依存して決定されるコストを評価するということは既に前節で述べた. これを cost を使って表現すると次のようになる. すなわち, $\text{cost}[\![p]\!]a$ は同じプログラム $p$ についても引数 $a$ によって一般に変化するが, 以下の議論ではこれを単純化して, "size $a$ = size $b$ ならば $\text{cost}[\![p]\!]a = \text{cost}[\![p]\!]b$ が成立する, つまり cost は引数のサイズにのみ依存するという前提で議論を行なう.

cost の性質を述べる. まず cost は関数適用の依存関係がない二つの関数からなるプログラムのコス

トを, 二つの関数それぞれのコストの和とする. すなわち関数の直積 $\langle f,g\rangle(\stackrel{def}{=}\lambda x.(fx,gx))$ では, 関数 $f,g$ はそれぞれ独立に引数に適用されるが, このプログラムに対して

$$\mathrm{cost}[\![\langle f,g\rangle]\!]x=\mathrm{cost}[\![f]\!]x+\mathrm{cost}[\![g]\!]x$$

が成立する. また cost は先行評価における時間計算量を評価するので, 合成関数のように関数適用に依存関係があるようなプログラムの項のコストを内側から評価する関数である. 例えばプログラム $f\cdot g$ と引数 $x$ に対して,

$$\mathrm{cost}[\![f\cdot g]\!]x=\mathrm{cost}[\![g]\!]x+\mathrm{cost}[\![f]\!](gx)$$

が成立する.

ここでは紙面の都合上 cost の形式的な定義を省略する. cost の定義については文献 [3] に形式的に述べている.

### 3.3 SD 変換と等価な規則

SD 変換のコストを解析するには foldr/build 規則のコストを評価しなくてはならないが, 本稿で用いるコスト評価法は一階の関数をコスト評価の対象とするため, foldr や build のコストは評価できない. よって SD 変換の手順 1 や手順 4 の実行前後のコストをそのままでは比較できない. また SD 変換はプログラムの引数リストをプログラムとまとめて, build, foldr に書き換えてしまい, SD 変換前のプログラムの引数とプログラムが, 変換後には明示的に分割して表現されなくなる可能性がある. そこで SD 変換の各手順におけるコストを解析せずに SD 変換全体におけるプログラムの変化をプログラム引数 (これは変換前後で共通にする) を明示的に分けた形で表現し, コストの増減を解析する. そのため, build を拡張し, 先の SD 変換と同一の意味を持つ次の拡張 2を示す.

**[拡張 2]** build を次のように拡張する.

$$\begin{aligned}&\mathrm{build}\ ::((\alpha\to\beta\to\beta)\to\beta\to\gamma\to\beta)\to\gamma\to\mathrm{List}\ \alpha\\&\mathrm{build}\ h=h\ (:)\ nil\end{aligned}\qquad\square$$

ある関数 $f::\gamma\to\mathrm{List}\ \alpha$ がもともとの build によって,

$$\begin{aligned}fx\ &=\mathrm{build}\ g\\&\quad\ \mathbf{where}\ g::(\alpha\to\beta\to\beta)\to\beta\to\beta\end{aligned}$$

と表現されるとき, $x::\gamma$ が仮引数の場合には $g$ にも仮引数 $x$ が内在するはずである. 明示的に,

$$fx=\mathrm{build}\ g(x)$$

と記述すると, $f$ を直接次の式で表現できる.

$$f=\lambda v.(\mathrm{build}\ g(v))$$

この式を, build の拡張によって,

$$f=\mathrm{build}\ (\lambda cn.(\lambda v.(g(v)\ c\ n)))$$

と記述しようというのが拡張の意味である. この式のうち $(\lambda cn.(\lambda v.(g(v)\ c\ n)))$ は, $f$ の build を用いた定義と対応するものであるが, このことをより明示的に表すために次の形式的な定義を行なう.

**[定義 3]** $\mathcal{H}[\![f]\!]$ を $f$ の build ,foldr による定義への書き換えとして定義する. すなわち,

$$fx=\mathrm{build}\ g\Rightarrow\mathcal{H}[\![f]\!]=\lambda cn.(\lambda v(g[v/x])\ c\ n)\qquad\square$$

$g[v/x]$ は $g$ に内在する仮引数 $x$ から $v$ への置換である. $\mathcal{H}[\![f]\!]$ を使って $f=\mathcal{H}[\![f]\!]\ (:)\ nil$ と表現することができる.

build の拡張から, $\mathcal{H}[\![f]\!]$ の定義までを直観的に説明するために写像の例を示す.

build の拡張により foldr/build 規則は, 引数とは独立に次の変換規則として表現できる.

**[規則 4]** もとの foldr/build 規則と同様に, 変換規則として次の等式が成り立つ.

$$(\mathrm{foldr}\ \ k\ z)\cdot(\mathrm{build}\ h)=h\ k\ z$$

また関数 $h::\mathrm{List}\ \beta\to\gamma$ に対してある二項演算子 $k::\beta\to\gamma\to\gamma$ と値 $z::\gamma$ が存在して, 任意の $a::\beta, x::\mathrm{List}\ \beta$ について等式,

$$\begin{aligned}h\ nil\ \ &=z\\h\ (a:x)\ &=k\ a\ (hx)\end{aligned}$$

を満たすとき, $h$ とリストを返す関数 $f::\alpha\to\mathrm{List}\ \beta$ に対して次の等式が成り立つ.

$$h\cdot f=\mathcal{H}[\![f]\!]\ k\ z\qquad\square$$

foldr/build 規則 $\mathrm{foldr}\ \ k\ z(\mathrm{build}\ g)=g\ k\ z$ はリスト $(\mathrm{build}\ g)$ に foldr を適用した結果が $g\ k\ z$ に等しいことを示す規則だったが, 規則4 は関数 $(\mathrm{foldr}\ \ k\ z)$ と関数 $(\mathrm{build}\ g)$ との合成が関数 $(h\ k\ z)$ に等しいことを示す.

規則 4の後半は手順 1 から 4 をまとめたものであり SD 変換全体を通してのコストの変化を解析す

るには規則 4についてコスト解析してもよい. 関数 $f, h, k$ が一階の関数であれば, 規則 4の等式は一階の関数からなる等式となり, 我々のコスト評価法によるコスト解析が可能である.

## 4. コスト解析

### 4.1 コスト評価の準備

文献 [3] では議論を簡潔にする目的から : のコスト $\mathrm{cost}[\![:]\!](a,x)$ を定数としている. しかし本稿では定数とはしないで, 引数 $(a,x)$ に依存するコストとする. これは SD 変換の : のコストが効率に大きく影響するという立場を尊重するためである.

コストの解析を簡潔に行なうために cost の他にもう一つコスト関数 cost′ を導入する. これはリストを返すプログラムのコストを評価する関数で, cost とほとんど同じ関数だが, 異なる点は後者が最外リスト構成子 : のコストを 0 とする点である.

**[定義 5]** $\mathrm{cost}' :: (\alpha \to \mathrm{List}\ \beta) \to \alpha \to \mathbb{N}$ を次の式で定義する.

$$
\begin{array}{l}
\mathrm{cost}'[\![:\cdot\langle f,g\rangle]\!]a = \mathrm{cost}[\![f]\!]a + \mathrm{cost}'[\![g]\!]a \\
\mathrm{cost}'[\![K_{nil}]\!]a \quad = \mathrm{cost}[\![K_{nil}]\!]a \\
\quad \textbf{where} \quad f \ :: \alpha \to \beta \\
\qquad\qquad\ \ g \ :: \alpha \to \mathrm{List}\ \beta
\end{array}
\qquad \square
$$

$\mathrm{cost}[\![:\cdot\langle f,g\rangle]\!]a$ は $\mathrm{cost}[\![:]\!](fa, ga) + \mathrm{cost}[\![f]\!]a + \mathrm{cost}[\![g]\!]a$ となるところ, 定義では $\mathrm{cost}'[\![:]\!]$ の項がないので, cost′ は : のコストを 0 とする. また $\beta$ がリスト型のとき $f$ はリストを返すが, 定義では $f$ のコストを cost によって評価するため $f$ の中で適用される : のコストを評価する. 一方最外リストを返す $g$ のコストは cost′ によって評価されるので $g$ 中の : のコストは 0 とされる. 故に cost′ は最外リストの : のコストを 0 とするコスト評価関数である.

### 4.2 SD 変換のコスト評価

規則 4 のコストの解析を 2 段に分けて行なう. すなわち式

$$
\begin{array}{lll}
 & h \cdot f & \\
= & (foldr\ k\ z) \cdot (\mathrm{build}\ \mathcal{H}[\![f]\!]) & (1) \\
= & \mathcal{H}[\![f]\!]\ k\ z & (2)
\end{array}
$$

の各等号の間のコスト比較を行なうことにする.

#### 4.2.1 等式 1のコスト比較

等式 (1) のコストを解析するのにまず $\mathrm{cost}'[\![f]\!]x$ について考える. $fx = a_1 : (a_2 : \cdots : (a_n : nil))$ とおくとき, 先行評価におけるコストを評価するので $\mathrm{cost}'[\![f]\!]x$ は $fx$ の成分 $a_1, a_2, \ldots, a_n$ を構成するのに要する時間計算量をコストとして返す. よって $f$ のコストは $\mathrm{cost}'[\![f]\!]x$ と $a_1, a_2, \ldots, a_n$ からリストを構成するのに要するコストの和として表現できる.

$$
\begin{array}{l}
\mathrm{cost}[\![f]\!]x = \mathrm{cost}'[\![f]\!]x + \mathcal{C}[\![(:), nil]\!]\,(fx) \\
\textbf{where} \\
\quad \mathcal{C} :: (\alpha \to \beta \to \beta) \to \beta \to \mathrm{List}\ \alpha \to \mathbb{N} \\
\quad \mathcal{C}[\![c,n]\!]\ nil \quad = 0 \\
\quad \mathcal{C}[\![c,n]\!]\ (a:x) \\
\qquad = \mathcal{C}[\![c,n]\!]\ x + \mathrm{cost}[\![c]\!](a, (foldr\ c\ n)x)
\end{array}
$$

$\mathcal{C}[\![(:), nil]\!]$ はリストの要素から, : によりリストとして構成するのに要するコストを返す.

同様に $\mathcal{H}[\![f]\!]\ (:)\ nil$ についても次の式が成り立つ.

$$
\begin{array}{l}
\mathrm{cost}[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x = \mathrm{cost}'[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x \\
\qquad\qquad\qquad\qquad + \mathcal{C}[\![(:), nil]\!](\mathcal{H}[\![f]\!]\ (:)\ nil\ x)
\end{array}
$$

ここで $\mathcal{H}[\![f]\!]\ (:)\ nil\ x$ は $fx$ に等しいので $\mathcal{C}[\![(:), nil]\!]\,(\mathcal{H}[\![f]\!]\ (:)\ nil\ x)$ は $fx$ を構成するのに要するコスト $\mathcal{C}[\![(:), nil]\!]\ (fx)$ に等しい. 一方 $\mathrm{cost}'[\![f]\!]x$ と $\mathrm{cost}'[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x$ は同じ $a_1, a_2, \ldots, a_n$ を生成するコストだが, 生成の仕方は一般に異なるのでコストが同一にはならない.

これらのことから等式 1の両辺にリスト $x$ を適用したときのコストとして次の結果が得られる.

$$
\begin{array}{ll}
 & \text{(左辺のコスト)−(右辺のコスト)} \\
= & \mathrm{cost}[\![h]\!](fx) - \mathrm{cost}[\![foldr\ k\ z]\!](fx) \\
 & + \mathrm{cost}'[\![f]\!]x - \mathrm{cost}'[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x
\end{array}
$$

この結果は等式 1のコストの差が foldr , build による $h$,$f$ の定義に依存して決まることを示している.

#### 4.2.2 等式 2のコスト比較

等式 1 同様に $\mathcal{H}[\![f]\!]\ k\ z$ のコストは, リスト $(fx)$ の要素を生成するコストと, $k$ による畳み込みのコストの和として表現できる.

$$
\mathrm{cost}[\![\mathcal{H}[\![f]\!]k\,z]\!]x = \mathrm{cost}'[\![\mathcal{H}[\![f]\!](:)nil]\!]x + \mathcal{C}[\![k,z]\!](fx)
$$

foldr $k\ z$ のコストは文献 [3] により $\mathcal{C}[\![c,n]\!]$ を用いて次の式で表現できる.

$$
\begin{array}{l}
\mathrm{cost}[\![foldr\ k\ z]\!]x \\
= \mathcal{C}[\![k,z]\!]\ x + \mathcal{D}\ x + \mathrm{cost}[\![K_n]\!](nil) \qquad (3)
\end{array}
$$

$$
\begin{array}{ll}
\textbf{where} & \\
D & :: (\mathrm{List}\ \alpha \to \beta) \to (\mathrm{List}\ \alpha) \to \mathbb{N} \\
\mathcal{D}(a:x) & = \mathcal{D}\ x + \mathrm{cost}[\![\ is_{nil}\ ]\!](a:x) \\
 & \quad + \mathrm{cost}[\![\mathrm{hd}]\!](a:x) + \mathrm{cost}[\![\mathrm{tl}]\!](a:x) \\
\mathcal{D}\ nil & = \mathrm{cost}[\![\ is_{nil}\ ]\!]nil
\end{array}
$$

ただし $is_{nil}$ は入力が空リストであるという述語, $K_n$ は $n$ を返す定数関数, $hd, tl$ はそれぞれリ

ストのヘッドとテールを返す関数である. $\mathcal{D}\ x$ は $hd, tl, is_{nil}$ によってリスト $x$ を各要素に分解するコストを表す.

以上から等式2にリスト $x$ を適用したときのコストを解析する.

(左辺)

$$\begin{aligned}&\mathrm{cost}[\![(foldr\ k\ z)\cdot(\mathcal{H}[\![f]\!]\ (:)\ nil)]\!]x\\ =\ &\{\text{関数の合成}\}\\ &\mathrm{cost}[\![foldr\ k\ z]\!](fx)+\mathrm{cost}[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x\\ =\ &\{\text{先の結果}\}\\ &\mathcal{C}[\![k,z]\!]\ (fx)+\mathcal{D}\ (fx)+\mathrm{cost}[\![K_n]\!](nil)\\ &+\mathrm{cost}'[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x+\mathcal{C}[\![(:),nil]\!]\ (fx)\end{aligned}$$

(右辺)

$$\begin{aligned}&\mathrm{cost}[\![\mathcal{H}[\![f]\!]\ k\ z]\!]x\\ =\ &\{\text{先の結果}\}\\ &\mathrm{cost}'[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x+\mathcal{C}[\![k,z]\!]\ (fx)\end{aligned}$$

(左辺のコスト) − (右辺のコスト)
$= \mathcal{D}\ (fx)+\mathrm{cost}[\![K_n]\!](nil)+\mathcal{C}[\![(:),nil]\!]\ (fx)$

等式2のコストの差には中間リスト $(fx)$ を構成するコスト $\mathcal{C}[\![(:),nil]\!]$ と $(fx)$ からリストの各要素に分解するコスト $\mathcal{D}\ (fx)$ が現われており, これは中間リストを含まないプログラムを導出することによる効率化と符合している.

#### 4.2.3 SD 変換によるコストの変化

前節までの結果から次の定理が成り立つ.

[定理 6] 規則4の左辺のコストと右辺のコストの差は次の式で表される.

(左辺のコスト) − (右辺のコスト)
$= \mathcal{C}[\![(:),nil]\!]\ (fx)+\mathcal{D}(fx)+\mathrm{cost}[\![K_n]\!]\ nil-\Delta(x)$

where $\Delta(x) = \mathrm{cost}[\![foldr\ k\ z]\!](fx)-\mathrm{cost}[\![h]\!](fx)$
$+\mathrm{cost}'[\![\mathcal{H}[\![f]\!]\ (:)\ nil]\!]x-\mathrm{cost}'[\![f]\!]x$ □

コストの差が正のとき, 規則4の変換によりコストが減少する. $\Delta(x)$ はプログラムを build ,foldr を用いた定義で書き換える際のコストの増加量を意味する. すなわち定理6はプログラムの書き換えによるコストの増加量と, プログラムから中間リストを除去することによるコストの減少量のうち, 減少分が大きければ SD 変換全体のコストが減少することを示している.

またこの式から次の事柄が分かる

- 関数 $h, f$ をそれぞれ $foldr k\ z, \mathcal{H}[\![f]\!]$ に変換する際のコストの増加量によって, SD 変換のコストの増減が決まる.
- 不等式の右辺は中間リストを生成するコスト $\mathcal{C}[\![(:),nil]\!]\ (fx)$ と生成した中間リストからその成分を取り出すためのコスト $C_0 + C_1 \times \mathrm{size}\ x$ の和となっている. これは文献 [1] において主張される "中間リストを生成しないプログラムへ変換することによる効率の改善" に対応している.
- 特に関数 $h, f$ が元々 $foldr, \mathcal{H}[\![f]\!]$ によって定義されるとき SD 変換によりコストが常に減少する. 減少量は中間リストの生成とその成分の取り出しに要するコストである.
- 関数 $h, f$ を $foldr, \mathcal{H}[\![f]\!]$ に変換する部分は, 元の SD 変換では手続き 1 に対応する部分である. すなわち関数 $h$ を $foldr$ の形式で書き換えるのは $h$ の foldr による定義にしたがって行なわれる. したがってこの書き換えが効率を損なわないように行なわれれば (効率を損なわないよう $h$ を foldr により定義しておけば), SD 変換によりコストは減少するといってよい.

## 5. あとがき

本稿では先行評価による SD 変換のコストを解析した. 結果として,SD 変換のために用意される, build, foldr による各関数の書き換えが, 効率を損なわないものである限り, SD 変換のコストが減少することが形式的なコスト評価により明らかになった. また中間リストの生成と消費分のコストが, SD 変換全体のコストに大きく寄与していることも確かめられた.

## 参考文献


[1] A.J.Gill, J.Launchbury, and S.L.Peyton Jones. "A short cut to deforestation," Proc. Functional Programming and Computer Architecture, pp.223–232., June 1993.

[2] P.Wadler, "Deforestation: Transforming programs to eliminate trees." in LNCS, vol.300, ed. H.Ganzinger, pp.344–358, Springer Verlag, Berlin, 1998.

[3] 西田誠幸, 辻野嘉宏, 都倉信樹, "関数プログラムの先行評価における逐次時間計算量評価手 法について," Technical report, vol.96-ICS-2, 大阪大学基礎工学部情報工学科, July 1996.

[4] 尾上能之, 武内正人, "Shortcut deforestation における効率の解析," In 日本ソフトウェア科学会第 *12* 回大会論文集, no.D7-2, pp.261–264, Sept, 1995.