D7-2

Shortcut Deforestation における効率の解析

Efficiency Analysis for Shortcut Deforestation

尾上 能之† 武市 正人†
Yoshiyuki ONOUE Masato TAKEICHI

†東京大学大学院工学系研究科情報工学専攻
Department of Information Engineering, Graduate School of Engineering
University of Tokyo

概 要

fold/unfold に基づくプログラム変換では，無限な展開を防ぐための folding の手間が大きいことが知られている．そこでリスト処理に関して一定の関数を用いることにより，folding を不要にした Shortcut Deforestation が提案された．本発表では，Shortcut Deforestation において各変換規則毎の簡約段数の増減に対して見積りを与えることにより，プログラムを実行することなく効率の改善度を測定できるようにした．

1 はじめに

関数型言語では，基本的な働きをする小さなプログラム (関数) を予め沢山用意し，それらを組み合わせてプログラムを書く．そこで，簡潔なプログラムが書ける，プログラムの再利用性が高い，などの特長がある．

これより「組み合わされた複雑な関数を実行時に毎回解釈するため，実行時のオーバヘッドが大きい」という問題点が生じる．また「小さな関数を組み合わせて用いるため，関数間のデータの受け渡しに本来は不要なリストを生成してしまうため，ヒープ使用量が増大する」という問題点もある．

そこで簡潔なプログラムを効率の良いプログラムに変換する手法として fold/unfold に基づくプログラム変換 [Bur77]や，これを応用した Deforestation [Wad88] が提案されたが，fold/unfold 変換の問題点でもある folding の手間が大きい，という欠点をかかえている．

これを解決する案として Shortcut Deforestation [Gil93](以降 SD 変換と略) が提案された．本発表ではこの SD 変換を対象にし，その変換の前後において対象となるプログラムの効率がどのように改善されるかを解析し，正確に求められるようにする．

2 SD 変換

Deforestation [Wad88] には，任意の中間的なデータ構造を除去できる，という特長がある反面，ある制約を満たさないプログラムを変換すると無限ループに陥る可能性がある，という欠点があった．

これに対して SD 変換では除去の対象となるデータ構造をリストに限定し，変換対象のプログラムに制約を与えることを不要とした．これを拡張し，リスト構造に限らず一般的なデータ構造に対して，SD 変換が行なえることが [Tak95]，[Lau95]で示されている．

その方法を簡単に説明すると，リストを消費 / 生成する関数はすべて `foldr/build`という決められた関数を用いて表現することにする．`build`の定義は `build` g = g `(:) []` で，g はリストを生成する際に用いられる `(:)` と `[]` を λ 抽象した関数であり，`build`を適用することによってはじめてリストが生成される．

内部的にリストの受け渡しが行なわれる場合は以

下のような foldr/build規則が適用できる．

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

これにより，build g で表わされるリストの (:), [] はそれぞれ k, z で置換されるので，不要なリストは生成されずに各要素に対して直接 k が適用できるようになる．

実際には SD 変換は以下の流れにそって行なわれる．なお本発表では，プログラムの表記に関数型言語 Haskell [Ham95] を用いる．

1. リストの内包表記 (list comprehension) の代わりに，リストの生成，消費にそれぞれ build, foldrを用いた定義に書き換える
2. リストを扱う関数を build, foldrを用いた定義へ書き換える
3. リストを扱う関数を unfold する
4. foldr/build規則を適用し，可能な β 簡約を行なう．この規則を適用する場所が存在する限り，4. の処理を繰り返す
5. 関数 buildを unfold する
6. 関数 foldrを unfold する

効率の観点から見ると，初めにリストを受け渡しする関数に対して統一的な扱いができるような準備をするため効率は悪化するが，その後はリストを媒介していた2関数を合成するなど可能な計算を進めることによって効率は単調に良化する．

3 効率の推定

3.1 効率の指標

効率を測る指標としては，簡約段数，所要時間，ヒープ使用量などが考えられる．ここでは関数型言語 Gofer [Jon94] における簡約段数の増減を調べることにする．Gofer は関数型言語 Haskell のサブセットになっており，最適化をしないオプションを選ぶことによってソースプログラムを忠実に実行することが可能で，内部の動作を知りたいときには便利な言語である．

簡約段数の数え方は，関数適用に対して unfold を一回行なうと一段簡約が進んだとみなす．より具体的には，引数の個数に関係なく関数適用の式に対して β 簡約を行なった回数が簡約段数になる．また Gofer ではすべての λ 式に対し内部的に名前付けをしていて，普通の関数と同様に扱っているため λ 式の適用も簡約段数に数えられる．

3.2 基本変換に関する簡約段数の増減

SD 変換を細分化し，簡約段数の増減にかかわるいくつかの簡単な変換を抽出し，これを基本変換と呼ぶことにする．第2節の結果より，与えられた式に対して SD 変換を行なうときに，各基本変換の起きる回数とそれに対する効率の増減を求めることができれば，全体的にどれだけ効率が改善するかが見積もれるはずである．そこで本節では，この基本変換について説明する．

3.2.1 buildの導入 (B^{-1})/ 消去 (B)

```
build g where g = (\ c n -> ...)
```

という式が現われた場合，build, g はいつでも unfold することが可能である．すなわち

```
build g => g (:) [] => ..(:)..[]..
```

これを変換 B とし，この B によって2つの関数を unfold したので簡約段数は2段減少する．またこの変換の逆で buildを導入するものを B^{-1} とする．

3.2.2 foldrの導入 (F^{-1})/ 消去 (F)

リスト xs を消費する関数は

```
h xs where h []     = z
           h (x:xs) = k x (h xs)
```

のように定義されることが多い．この定義は，リストの各要素に対し順々にある一定の処理をすることを仮定しているので，リストの中の特定の要素だけを用いる関数 (head など) はこのような定義をすることはできない．foldrを用いた定義から，この式への変換

```
foldr k z xs =>
    h xs
    where h []     = z
          h (x:xs) = k x (h xs)
```

では，foldrが unfold される代わりに h が導入され，この面での簡約段数の増減はない．

この変換を逆にみると，h の定義の2行目において第1引数に x，第2引数に (h xs) がくるような定義のときは k の簡約が済んでいることになるが，それ以外の場合は foldrを用いた式に抽象化する際 k がラムダ式である必要がある．そこで以下のようにラムダ式 k を簡約することによって，更に変換を進めることができる．

```
foldr k z xs =>
    h xs
    where h []     = z
          h (x:xs) = ..x..(h xs)..
```

ここでリスト xs の長さが n のとき，この変換を

表 1　filter に対する基本変換の内訳

変換名	B^{-1}	F_n^{-1}	U	FB_n	FB_1	B	F_n
生起回数	$n+2$	1	$2n+2$	1	n	1	1
段数 / 回	$+2$	$+n$	-1	$-(n+3)$	-4	-2	$-n$
計 $5n+3$	$2(n+2)$	n	$-(2n+2)$	$-(n+3)$	$-4n$	-2	$-n$

表 2　unlines に対する基本変換の内訳

変換名	CN_1	B^{-1}	F_n^{-1}	U	FB_n	FB_{m+1}	FB_1	B	F_n
生起回数	n	$2n+2$	2	$3n+2$	1	n	n	1	1
段数 / 回	$+2$	$+2$	$+n$	-1	$-(n+3)$	$-(m+4)$	-4	-2	$-n$
計 $(m+6)n+3$	$2n$	$4(n+1)$	$2n$	$-(3n+2)$	$-(n+3)$	$-(m+4)n$	$-4n$	-2	$-n$

```
>           where g []     = '\n' : h xss
>                 g (y:ys) = y : g ys
```

この変換前後の関数 unlines を，長さ m の文字列を n 個並べたリストに適用したところ，簡約段数は変換前が $(2m+8)n+5$，変換後が $(m+2)n+2$ となり，その差 $(m+6)n+3$ は表 2による解析結果と等しいことがわかる．

4.3　n-queens 問題

8-queens の問題に対し，再帰的に求める候補のリストの長さが既知であるものと仮定して，この解析を行なった．結果は -2.41×10^5 段で，実際に gofer で実行した結果の -0.86×10^5 段 (変換前 13.87×10^5，変換後 13.01×10^5) の約 3 倍になっている．これは遅延評価のために，リストをすべてたどらなかったケースが多かったことが原因である．

表 3　queens 10 の実行 ([Gil93]のデータ)

	変換前	変換後
所要時間 [s]	24.4	8.8
消費ヒープ量 [MB]	179	36

[Gil93]には，Glasgow Haskell Compiler(ghc) に SD 変換を組み込み，10-queens を計算した結果が記してある (表 3)．この結果では所要時間が約 1/3 になっていることから，コンパイラに組み込むことによって，ghc における最適化との相乗効果が現われていることがわかる．

5　今後の課題

本発表では．SD 変換を行なう際に簡単な規則に対して簡約段数の増減を与えることによって，変換の前後における簡約段数の差を求められるようにした．これによりプログラムを実行しなくても変換の効果を調べることができるようになった．

効率を測る指標として簡約段数の減り具合を予測することによって，時間的な効率の改善を近似できるようになったので，今後の課題としては，ヒープ使用量など空間的な効率の改善を近似できるようにすることが考えられる．ただしこれは処理系が式を評価するために内部で行なっている処理に大きく依存する．また関数型言語の多くには，プログラムを高速に実行できるようにコンパイラが用意されている．このコンパイルしたプログラムモジュールに対しても効率の改善度を予測することも考えられる．

参考文献

[Bur77] Burstall, R. M. and J. Darlington: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, Vol. 24, No. 1, pp. 44–67, 1977.

[Gil93] Gill, A., J. Launchbury, and S. L. P. Jones: A Short Cut to Deforestation, in *FPCA '93*, pp. 223–232, ACM Press, 1993.

[Ham95] Hammond, K., L. Augustsson, B. Boutel, W. Burton, J. Fairbairn, J. Fasel, A. Gordon, M. M. Guzmán, J. Hughes, P. Hudak, T. Johnsson, M. Jones, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, S. P. Jones, and P. Wadler: *Report on the Programming Language Haskell, Version 1.3*, 1995.

[Jon94] Jones, M. P.: *Gofer 2.30a release notes*, 1994.

[Lau95] Launchbury, J. and T. Sheard: Warm Fusion: Deriving Build-Catas from Recursive Definitions, in *FPCA '95*, pp. 314–323, ACM Press, 1995.

[Tak95] Takano, A. and E. Meijer: Shortcut Deforestation in Calculational Form, in *FPCA '95*, pp. 306–313, ACM Press, 1995.

[Wad88] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees, in Ganzinger, H. ed., *ESOP '88 2nd European Symposium on Programming*, LNCS 300, pp. 344–358, Nancy, France, 1988, Springer-Verlag.

F_n と呼ぶことにする．k はリストの各要素に対して適用されるので，変換 F_n による簡約段数の増減は $-n$ である．変換 F に対しても逆変換を F_n^{-1} と呼ぶことにする．

3.2.3 foldr/build規則の適用 (FB)

以下のような foldr/build規則の適用と1回の β 簡約を含めて変換 FB_n とする．n は build g によって生成されるリストの長さである．

```
foldr k z (build g) => g k z => ..k..z..
```

規則の適用後の式に注目すると，適用前に比べ foldr, build, g が消えている．この内 buildと g は1回適用されるので2段減少するのに加え，foldrはリストの各要素と z に対してあわせて $n+1$ 回適用されていたものを省いていることになる．よって変換 FB_n 1回につき $n+3$ 段簡約段数が減少していることになる．

これは k がリストの要素すべてに適用される場合を想定しているが，遅延評価による実行の場合このような仮定は成り立たないので，実際は最大 $n+3$ 段減るということになる．

3.2.4 その他一般 (U, CN)

リストを扱う関数は build, foldrを用いて定義されるので，再帰的に定義されているものはなく，変換の最初にすべて unfold される．これを変換 U とし，この変換を一度行なうと簡約段数は1段減少する．また foldr/build規則を適用することによって，β 簡約が可能になる場合があり，これも変換 U で表わすことにする．

それと SD 変換の最後に，残された foldr を unfold する際

```
foldr (:) [] xs => xs
```

という変換も行なう．リスト xs の長さが n のとき，これを変換 CN_n とし，簡約段数は foldrの適用回数に相当する $n+1$ 減少する．

3.3 プログラム全体に対する簡約段数の増減

前節の基本変換に対する簡約段数の増減を考慮しつつ SD 変換を行なうことによって，評価しようとしているプログラム全体における簡約段数の増減が測れることになる．

これより SD 変換を行なった効果として現われる簡約段数の減少は，以下のようにまとめられる．

- 高度に抽象化した関数 (map, filter など) を用いるためには λ 式の利用が不可欠で，これが簡約段数の増加につながっていたが，SD 変換を行なうと λ 式による抽象化が不要になる場合が多い．
- 中間的に媒介される内部構造を除去することにより，ヒープ使用量の減少とともに，簡約段数を減らすことができる．

4 変換の実例

4.1 変換の例 (1) filter

変換の元になるのは以下のように定義された filter のプログラムである．

```
> filter p xs = concat (map f xs)
>               where f x = if (p x) then [x]
>                                    else []
> map f xs = h xs
>            where h []     = []
>                  h (x:xs) = f x : h xs
> concat xs = foldr (++) [] xs
> xs ++ ys = h xs
>            where h []     = ys
>                  h (x:xs) = x : h xs
```

このプログラムを第3.2節の変換規則に従って変換すると，基本変換の内訳は表1のようになる．これより全体として簡約段数は $5n+3$ 減少する．

```
> filter p xs =
>   h xs
>   where h []     = []
>         h (x:xs) = if (p x) then x : (h xs)
>                             else h xs
```

この変換前後の関数 filter に対して filter id [True,..,True] を gofer(2.30a) 処理系 で実行すると，長さ n のリストに対して簡約段数は変換前が $9n+6$，変換後が $4n+3$ 段で，半分以下になった．この変換による減少分は $5n+3$ であり，解析による結果と等しいことがわかる．

4.2 変換の例 (2) unlines

次は，文字列のリストを入力とし各文字列の末尾に改行文字を加えた上で1つのリストにつなげ直す関数 unlines を考える．unlines の定義が以下のようになる．map, concat, ++ の定義は filter の例と同じである．

```
> unlines ls =
>   concat (map (\ l -> l ++ ['\n']) ls)
> unlines ls =
>   h ls
>   where h []       = []
>         h (xs:xss) =
>           g xs
```