プログラム融合変換の実用的有効性の検証

尾上 能之　胡 振江　岩崎 英哉　武市 正人

プログラム融合変換は，複数の関数を合成した形から単一の関数に融合することにより，関数間で受け渡しされる中間データ構造を除去する手法で，関数プログラミングなどで有用であることが期待されている．これまでに素朴なプログラムに対する融合変換の有効性を示した例はあるものの，ベンチマークのような大規模なプログラムに変換を適用した例は少ない．そこで本研究では，hylomorphism という再帰形に注目した融合変換システムを実現し，実用的な規模のプログラムに対して融合変換が有効であるか否かを検証する．

1 はじめに

プログラム融合変換 (program fusion) は，別個に定義された二つの関数 f, g の合成で表わされた式 $f \circ g$ を，新たに定義された単一の関数 h に融合することにより，関数間で受け渡しされる中間データ構造を除去する手法である．例えば，リスト xs の中のある要素が述語 p をみたすか否かを調べる関数 any は，

```
any p xs = or (map p xs)
```

と簡潔に定義できる反面，関数 map と or の間で本来不要なリストが受け渡しされてしまう．これを

```
any p []     = False
any p (x:xs) = p x || any p xs
```

のような定義に変換し中間データ構造の生成を防ごうとするのが，融合変換の目的である．

この融合変換を実現するための一手法として，構成的アルゴリズム論を用いる hylo fusion [11] [4] [9] が提案されている．この手法では，融合の対象となる再帰関数を hylomorphism (以下 hylo と略) と呼ばれる一般的な表現で表わした後，酸性雨 (Acid Rain) 定理を適用することによって hylo の合成式を融合することが可能になる．

本研究では，この hylo fusion を既存のコンパイラに埋め込んだ融合変換システム HYLO を実現し，いくつかの実用的な規模のプログラムを本システムで変換することにより，その有効性を検証する．

2 HYLO システム

HYLO システムの実装は，当初プロトタイプ上で実験を行なってきた [10]．このプロトタイプはインタプリタに基づいていたため実装が容易であり，小さな例に対して融合変換の効果をみるには十分であったが，既存のコンパイラと比べて遅く，広範囲の例題に適用することが難しかった．そこでより汎用的なシステムにする目的も兼ねて，プログラミング言語 Haskell の処理系の一つで，研究 / 応用の面も含め現在もっとも広く用いられているコンパイラである Glasgow Haskell Compiler(GHC) [2] に融合変換の処理を組み込むことにした．対象としたのは GHC-4.04 patchlevel 1 である．

図 1 は GHC の内部構成を示している．入力された Haskell プログラムを，Core 言語 / STG 言語を経て

Verification for Practical Effectiveness of Program Fusion.

Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, 東京大学 大学院工学系研究科, Graduate School of Engineering, University of Tokyo.

コンピュータソフトウェア, Vol.17, No.3 (2000), pp.81–85.
[小論文] 1999 年 8 月 9 日受付.

最終的に C のコードに落とし，gcc などの C コンパイラを用いて実行モジュールを生成する．Core 言語 / STG 言語においては，正格性解析や共通部分式の削除，更新部の解析やラムダ持ち上げ (lambda lifting) などの複数の最適化変換が適用されるが，我々はこの Core 言語上の最適化パスの最後に融合変換のアルゴリズムを埋め込んだ．追加した部分は図 2 のような流れになっており，前節の例は以下のように融合変換される (記号の詳細については [5]参照)．

or

$$
\begin{aligned}
&= \{\text{元の定義}\}\\
&\quad \lambda zs.\ \texttt{case}\ zs\ \texttt{of}\\
&\qquad [\,] \ \texttt{->}\ False\\
&\qquad x : xs \ \texttt{->}\ x \,||\, or\ xs\\
&= \{\text{1. 再帰関数から hylo の導出}\}\\
&\quad [\![False \triangledown (||),\ id,\ out_{\mathsf{L}}]\!]_{\mathsf{L},\mathsf{L}}
\end{aligned}
$$

map p

$$
\begin{aligned}
&= \{\text{元の定義}\}\\
&\quad \lambda zs.\ \texttt{case}\ zs\ \texttt{of}\\
&\qquad [\,] \ \texttt{->}\ [\,]\\
&\qquad x : xs \ \texttt{->}\ p\,x : map\ p\ xs\\
&= \{\text{1. 再帰関数から hylo の導出}\}\\
&\quad [\![[\,] \triangledown \lambda(x, xs).(p\,x : xs),\ id,\ out_{\mathsf{L}}]\!]_{\mathsf{L},\mathsf{L}}\\
&= \{\text{2. 融合の準備}\}\\
&\quad [\![in_{\mathsf{L}},\ id + \lambda(x, xs).(p\,x, xs),\ out_{\mathsf{L}}]\!]_{\mathsf{L},\mathsf{L}}
\end{aligned}
$$

any

$$
\begin{aligned}
&= \{\text{元の定義}\}\\
&\quad or\ .\ map\ p\\
&= \{\text{インライン展開}\}\\
&\quad [\![False \triangledown (||),\ id,\ out_{\mathsf{L}}]\!]_{\mathsf{L},\mathsf{L}}\\
&\qquad \circ\ [\![in_{\mathsf{L}},\ id + \lambda(x, xs).(p\,x, xs),\ out_{\mathsf{L}}]\!]_{\mathsf{L},\mathsf{L}}\\
&= \{\text{3. 酸性雨定理の適用}\}\\
&\quad [\![False \triangledown (||),\ id + \lambda(x, xs).(p\,x, xs),\ out_{\mathsf{L}}]\!]_{\mathsf{L},\mathsf{L}}\\
&= \{\text{4. 使用済み hylo の展開}\}\\
&\quad f\ \texttt{where}\\
&\qquad f = \lambda zs.\ \texttt{case}\ zs\ \texttt{of}\\
&\qquad\qquad [\,] \ \texttt{->}\ False\\
&\qquad\qquad x : xs \ \texttt{->}\ p\,x \,||\, f\ xs
\end{aligned}
$$

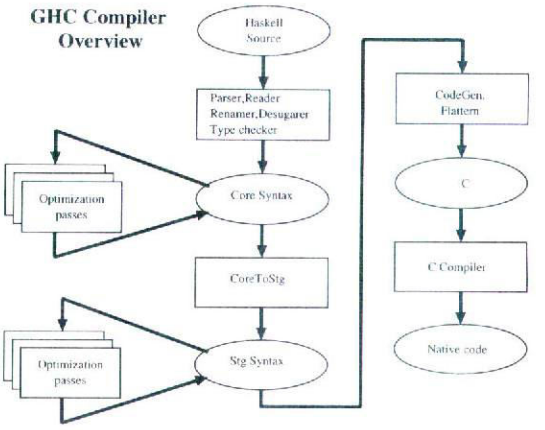

図 1 GHC の構造

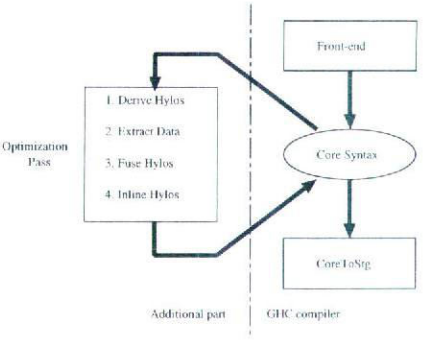

図 2 追加した変換部

変換部は，コンパイラの他の部分と同様に Haskell 言語で記述されており，行数は 1800 行ほどである．

従来のシステムでは，組込関数に対する融合変換を行なうことができなかったが，新しいシステムでは以下の関数について融合変換を行なうことが可能になっている．

`foldr, map, (++), concat, filter,`
`iterate, repeat, and, or, any, all`

また hylo fusion ではユーザ定義の関数についても融合変換を行なうことが可能であり，これは現在実装されている他の手法と比べて大きな特長の一つとなっている．

3 実験

本節では，我々の提案する手法の有効性を調べるために，実用的な規模のプログラムをコンパイルし，融合変換の適用の有無に応じた効率の改善度を調べた．

表 1 spectral ベンチマーク (一部)

プログラム	内容	行数
ansi	文字端末制御	125
awards	受賞者決定手法	111
cse	共通部分式削除	464
expert	エキスパートシステム	525
life	ライフゲーム	53
multiplier	二進積算器	498
power	べき級数展開	138
puzzle	組合せパズルの全解探索	170
rewrite	等式書換システム	631
scc	グラフ強連結分解	100

例として用いたプログラムは，Glasgow 大学で開発された NoFib benchmark suite [8] である．これは Haskell で書かれたプログラムの集まりで，プログラムの規模に応じて imaginary, spectral, real の 3 群から構成されている．今回は，中規模に相当する spectral 群の中から hartel を除く 37 種類のベンチマークプログラムを対象とした．その中のいくつかについては，表 1 に動作内容と行数を示してある．

実験は以下のような手順で行なった．

1. 融合変換の適用 / 非適用に応じて各プログラムをコンパイルし，要した時間と実行モジュールのサイズを比較する
2. 上の二つのモジュールを実行させ，要した時間とヒープの総使用量を比較する

実験に用いた計算機は PC/AT 互換機 (PentiumII Xeon 450MHz, メモリ 512MB) である．ただし実行時のヒープ使用量はコンパイラにのみ依存し，異なる機種でも常に同じ値となることが知られている．

3.1 コンパイル時

表 2 は，ベンチマークの各プログラムに対し融合変換を追加したことによるコンパイル時間と実行モジュールサイズの変化を示している．表の中の数値は，融合変換を適用しなかった場合に対する比率を表し，時間の短い順に並べてある．なお時間の計測は OS の time コマンドを用いて 3 回計測したものを平均している．なおコンパイルに要した時間は，短いもので 5 秒，長いもので 80 秒であった．

これより，多くのプログラムでコンパイル時間の増減は ±5% の範囲に収まっていることがわかる．時間が減少しているものは，融合変換によりプログラムが簡略化された結果，その後のコンパイル時間が短縮し，融合変換を追加したことによるオーバヘッドを上回ったことを示している．

表 2 コンパイル時間，実行モジュールサイズの変化

プログラム	時間	サイズ	プログラム	時間	サイズ
eliza	0.97	0.98	cryptarithm1	1.01	1.00
life	0.97	0.99	cryptarithm2	1.01	1.00
gcd	0.98	0.99	fft2	1.01	1.00
rewrite	0.98	0.95	mandel	1.01	1.00
atom	0.99	0.99	pretty	1.01	1.00
minimax	0.99	0.98	primetest	1.01	1.00
simple	0.99	0.95	banner	1.02	1.00
awards	1.00	1.00	circsim	1.02	0.98
calendar	1.00	0.99	fibheaps	1.02	1.00
expert	1.00	0.99	fish	1.02	1.00
knights	1.00	0.99	mandel2	1.02	1.00
multiplier	1.00	0.99	sorting	1.02	1.00
para	1.00	0.99	boyer	1.03	1.00
scc	1.00	1.00	power	1.03	1.00
sphere	1.00	0.99	ansi	1.05	1.00
treejoin	1.00	1.00	constraints	1.05	1.01
boyer2	1.01	1.00	puzzle	1.16	1.04
cichelli	1.01	1.00	cse	1.17	1.08
clausify	1.01	0.99			

また puzzle, cse ではコンパイル時間が大幅に増加している．これは，引数が構成子適用式である hylo 関数適用式のインライン展開による最適化を行なっているのが原因であるが，この場合でも実行モジュールサイズの増加は時間の増加よりも緩やかであることから，大きな問題ではないとみなせる．よって，融合変換の処理を追加してもプログラムのコンパイル時に大きな不利益を与えない，と考えられる．

3.2 実行時

前節で作成した各実行モジュールに対し，実行時のヒープ使用量を調べるためのオプションを指定して実行し，比率に応じて並べたものが表 3 である．また図 3 の実線は，変換無のときのヒープ使用量を 1 としたときの減少度 (1 − (比率)) をグラフ表示したもので，上に伸びるほどヒープ使用量が減少したことを示している．

表 3 より，37 種中 15 のプログラムで融合変換による効果が見受けられた．中でも 5% 以上改善したものも 7 種あり，これらのプログラムはいずれも内部でリストを扱う関数を多用していたため，融合変換によって無駄

表 3 ヒープ使用量の変化

プログラム	変換無 [byte]	変換有 [byte]	比率
ansi	27,016	20,684	0.77
gcd	30,465,228	25,778,708	0.85
rewrite	21,918,608	18,748,324	0.86
awards	123,528	108,632	0.88
puzzle	109,961,516	98,393,088	0.90
multiplier	105,577,196	95,506,784	0.91
eliza	280,792	256,036	0.91
circsim	1,323,405,040	1,266,436,300	0.96
knights	1,057,940	1,028,308	0.97
sphere	43,891,448	42,670,840	0.97
expert	112,372	110,500	0.98
simple	282,834,504	278,782,800	0.99
fibheaps	75,264,712	74,300,676	0.99
cse	545,020	539,752	0.99
life	223,503,124	222,088,040	0.99
banner	81,920	81,848	1.00
⋮	⋮	⋮	⋮
treejoin	57,568,156	57,568,156	1.00

な中間データが生成されなくなった効果が表れている．また比率が 1.00 のプログラムでは，ヒープ使用量が微かに減少しているものと変化しないものの二種類のみが存在し，増加しているものは現れなかった．

また図 4 の実線は，変換無のときの実行時間を 1 としたときの減少度 (1 − (比率)) をグラフ表示したものである．なお実行時間が 1 秒未満のものについては，値が小さく参考にならないのでグラフから除いてある．これより circsim, multiplier, puzzle のようにヒープ使用量が減少すると，実行時間も短縮する傾向にあることがわかる．しかし cichelli, para, power のように融合変換を行なった結果，遅くなる例も現れた．理論的には，融合変換により実行時間が遅くなることは考えにくく，実装に問題があると考えられるので今後改善していきたい．

4 おわりに

融合変換を関数型言語の処理系に実装する研究として，以下のものが知られている．

foldr/build fusion [3] は，リストを扱う基本的な関数 `foldr` で関数の再帰構造を抽象化し，抽象化された関数だけを融合の対象とする手法である．

- リスト以外の中間データを除去できない
- 用意された組込関数だけしか融合できない

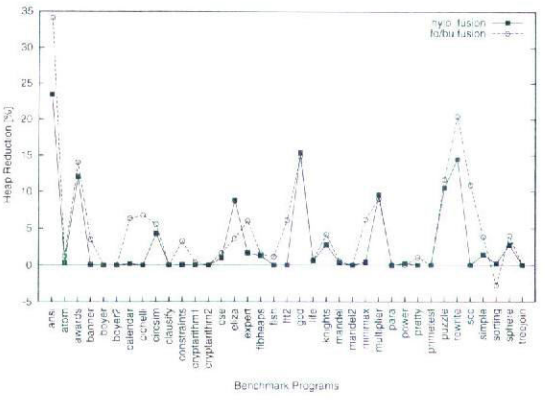

図 3 融合変換によるヒープ使用量の減少度

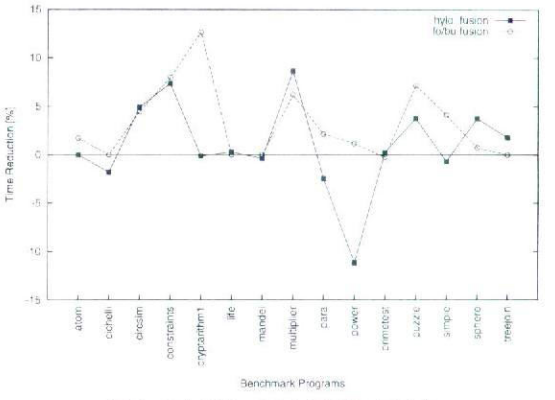

図 4 融合変換による実行時間の減少度

などの欠点があるものの，動作原理が簡単であることから，いくつかの実装 [1]が試みられている．

上の手法がもつ二つの欠点を解決するために提案されたのが warm fusion [6] である．この手法も GHC 上での実装 [7]が行なわれている．

一方，我々の用いている hylo fusion も上記二つの欠点を克服している．今回の実験により，コンパイル時の損失も少なく，実行時には多くのプログラムに対してヒープ使用量を減少させる効果があるのに加え，増加する例も無いことがわかった．よって実用的な規模のプログラムに対しても，有効な最適化変換であるといえよう．

我々が変換を埋め込む対象とした最新の GHC-4.04 では，新しく追加された機能である規則に基づく式の書換えを用いて，foldr/build fusion が実現されている．この変換を適用したときの比率が，図 3,4 に破線で示

してある．二つの手法を比較すると，elize, multiplier など hylo fusion が勝っているプログラムがあるものの，全般的に foldr/build の方が良い結果を残している．これは [m..n]，zip など，現在我々のシステムでは融合変換ができない，よく用いられる組込関数に対する融合も可能になっていることに起因する．例えば foldr/build を cichelli に適用すると，関数 [m..n] と map の融合変換が発生しヒープ使用量が減少する．

このような規則に基づく方式では，融合したい関数を該当する型も含めた規則としてユーザが明示的に指定することによって，ユーザ定義関数でも融合変換可能になるが，この規則は多様型を正しく記述する必要があり容易ではない．これに対し，hylo fusion ではこの処理を自動で行なうことから，今後システムを改善し融合変換可能な関数を広げられれば，利用者を問わず，foldr/build よりも効率のよいプログラムを生成できるようになると考えられる．

参考文献

[1] Chitil, O. : Type Inference Builds a Short Cut to Deforestation, *ACM SIGPLAN Notices, Proceedings of ICFP '99*, Vol. 34, No. 9 (1999), pp. 249–260.

[2] *The Glasgow Haskell Compiler.* http://www.haskell.org/ghc/.

[3] Gill, A., Launchbury, J. and Jones, S. : A Short Cut to Deforestation, *Proc. Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, June 1993, pp. 223–232.

[4] Hu, Z., Iwasaki, H. and Takeichi, M. : Deriving Structural Hylomorphisms from Recursive Definitions, *ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, Philadelphia, USA, ACP Press, May 1996, pp. 73–82.

[5] 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 (1998), pp. 57–70.

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

[7] Nemeth, L. and Jones, S. P. : A design for warm fusion, *The 10th International Workshop on Implementations of Functional Languages*, 1998, pp. 381–393.

[8] *The NoFib benchmark suite.* http://www.dcs.gla.ac.uk/fp/software/ghc/nofib.html.

[9] Onoue, Y., Hu, Z., Iwasaki, H. and Takeichi, M. : A Calculational Fusion System HYLO, *IFIP TC2 WG2.1 Algorithmic Languages and Calculi* (Bird, R. and Meertens, L. (eds.)), Alsace, France, Chapman & Hall, Feburary 17–22 1997, pp. 76–106.

[10] 尾上能之，胡振江，武市正人：HYLO システムによるプログラム融合変換の実現，コンピュータソフトウェア，Vol. 15, No. 6 (1998), pp. 52–56.

[11] Takano, A. and Meijer, E. : Shortcut Deforestation in Calculational Form, *Proc. Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995, pp. 306–313.