HYLO システムによるプログラム融合変換の実現

Implementation of Program Fusion by HYLO System

尾上 能之[†] 胡 振江[††] 武市 正人[†]
Yoshiyuki ONOUE Zhenjiang HU Masato TAKEICHI

†東京大学大学院工学系研究科計数工学専攻
Department of Mathematical Engineering and Information Physics,
Graduate School of Engineering, University of Tokyo
†† 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

関数プログラミングでは，単純な機能を持つ関数を組み合わせてプログラムを記述することにより，モジュラリティの優れた簡潔なプログラムを書くことができる．しかしこのように多数の関数を組み合わせると，関数間で受け渡しされる不要な中間データ構造が効率の弊害となる．この問題を解決するためにいくつかのプログラム融合変換 (fusion) が提案されてきたが，我々は hylomorphism という再帰形に注目した HYLO システムを実現し，システムが変換したプログラムのヒープ使用量を解析することによってシステムの有用性を示した．

1 はじめに

関数プログラミングでは，小さくて基本的な関数を組み合わせてプログラムを記述することが広く行なわれている．これは可読性やモジュラリティの面で優れている反面，多数の組み合わされた関数を実行することにより，関数間の情報の受け渡しに使われる中間データ構造が実行時に作られ，効率の低下の原因となる．

この問題の解決策として，従来から様々な方法が提案されてきた．これらの手法は，別個に定義された 2 関数 f, g の合成で表わされた式 $f \circ g$ を 1 つの新たに定義された関数 h に融合 (fusion) し無駄な中間データ構造を作らないようにする，という意味で融合変換 (program fusion) と呼ばれる．

初期の手法 [Wad88] [Chi92]は，基本的にすべての関数呼び出しを展開し次々に関数の融合を試みる．しかしこのままでは再帰関数を無限に展開してしまうため，展開した関数呼び出しの形をメモしておき，同じ形が再現したらそこで融合された新しい再帰関数を定義するような機構が必要である．このメモ化が複雑な処理であったため，この手法は実用化されるには至らなかった．

その後，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 [Gil93] [She93]が提案された．この手法ではメモ化は不要となり，単純な変換規則を局所的に繰り返し適用することにより関数間の融合が進んでいく．最近は構成的アルゴリズム論を用いることによって，除去される中間データ構造の種類がリストから一般のデータ構造へと拡張された手法 [Hu95] [Tak95]も提案されてきている．

本研究では，これらの手法を基にして開発された HYLO システム [Ono97]を紹介し，関数型言語の処理系に組み込まれているヒーププロファイリングの機能と組み合わせることによって，プログラムの実行効率の段階的な改善を試みることにする．

2 HYLO システム

融合変換を行なうためには，再帰関数の一般的な表現手法である hylomorphism(以下 hylo と略) と Acid Rain 定理を用いる． f, g を任意の再帰関数としその合成 $f \circ g$ に対して融合変換を行なうには， f, g を hylo を用いた式に書き換えて，2 つの hylo を Acid Rain 定理で 1 つにまとめあげればよい．

hylo は以下のように関数の 3 つ組で表わされる．この hylo の記述力は強力で，ある文法上の緩い制約を満たす任意の再帰関数が hylo 表現で表わすことが可能であることが知られている [Hu95]．

定義 1 (hylo の 3 つ組表現)
F, G を関手とし，関数 $\phi : G\,A \to A, \psi : B \to F\,B$ と自然変換 $\eta : F \dot{\to} G$, が与えられたとき，hylo $[\![\phi, \eta, \psi]\!]_{G,F} : B \to A$ は以下の等式における f の最小不動点として定義される．

$$f = \phi \circ (\eta \circ F\,f) \circ \psi \qquad \square$$

関数 ψ, η, ϕ はそれぞれ，引数の再帰データの分解，分解された各要素に対する処理，分解された再帰データの組立，の役割を果たす．式

$$[\![\phi_1, \eta_1, \psi_1]\!] \circ [\![\phi_2, \eta_2, \psi_2]\!]$$

を融合するためには，再帰データの組立処理である ϕ_2 と 分解処理である ψ_1 が互いに打ち消しあえば，中間データ構造を生成しなくてすむ．これを定式化したのが以下の Acid Rain 定理 [Tak95] である．

定理 1 (Acid Rain)

$$(a)\quad \frac{\tau : \forall A.\ (F\,A \to A) \to F'\,A \to A}{\begin{array}{r}[\![\phi_1, \eta_1, out_F]\!]_{G,F} \circ [\![\tau in_F, \eta_2, \psi_2]\!]_{F',L} \\ = [\![\tau(\phi_1 \circ \eta_1), \eta_2, \psi_2]\!]_{F',L}\end{array}}$$

$$(b)\quad \frac{\sigma : \forall A.\ (A \to F\,A) \to A \to F'\,A}{\begin{array}{r}[\![\phi_1, \eta_1, \sigma out_F]\!]_{G,F'} \circ [\![in_F, \eta_2, \psi_2]\!]_{F,L} \\ = [\![\phi_1, \eta_1, \sigma(\eta_2 \circ \psi_2)]\!]_{G,F'}\end{array}} \qquad \square$$

この定理を適用するためには，合成する 2 つの hylo における ψ_1, ϕ_2 が $\sigma\,out_F, \tau\,in_F$ のような特別な形に導出可能でなければならないという条件がある．この導出のアルゴリズムや他の条件などについては [Ono97]を参照されたい．任意の hylo からこのような形が導出できるわけではなく，導出に失敗した場合 Acid Rain 定理による融合は行なわれず最終的には元の 2 関数の合成 $f \circ g$ に書き戻される．

HYLO システムは，入力として関数型言語 Gofer で書かれたプログラムを受け取り，不要な中間データ構造を生成しないより効率の良いプログラムに変換して出力する．システムによる変換の過程は以下のような流れによって構成されている．

1. パース，変数名の付替，型検査などの前処理
2. 再帰関数から hylo の導出
3. 定理を適用しやすいように hylo を変形
4. 2 関数に対し定理を適用し 1 関数に融合
5. まだ定理を適用可能な式が残っている ⇒ 3. へ
6. 最終的に hylo 表現を元の再帰的定義に戻す

3 最適化の流れ

3.1 Step 0: n-queens プログラム

ここでは変換の対象として以下の n-queens のプログラムを扱うことにする．

```
main = (print . sum . concat . queens) 10
queens 0 = [[]]
queens (m+1)
    = [ p++[n] | p<-queens m, n<-[1..10],
                 safe p n ]
safe p n
    = all not [ check (i,j) (m,n)
              | (i,j) <- zip [1..] p ]
      where m = 1 + length p
check (i,j) (m,n)
    = j==n || (i+j==m+n) || (i-j==m-n)
```

実験に用いたマシンは Sun Ultra2 で，コンパイラには Glasgow Haskell Compiler (ver.0.29) と添付されるヒーププロファイリングツールを用いた．コンパイル時のオプションは '-O' とプロファイリング用に '-prof -auto-all' を指定した．これによりトップレベルで定義されたすべての関数についてのヒープ使用に関する情報が出力される．

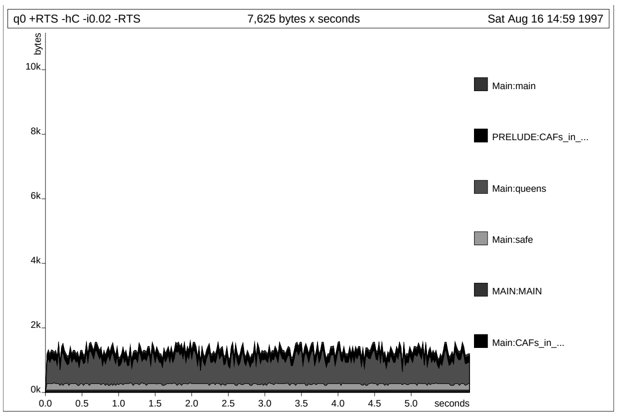

図 1 変換前のプログラム

表 1　所要時間とヒープ使用量 (total)

	Step 0	Step 1	Step 2	Step 3	Step 4
所要時間 [sec]	5.86	2.62	2.60	1.86	1.52
ヒープ使用量 [MB]	112.22	28.26	32.64	13.98	9.38

表 1 は，段階的に変換されたプログラムを実行した際の所要時間と使われたヒープの総量を示している．また図 1 は，実行時に各時点で生きているクロージャ (live closure) がヒープ中で占める量を時間経過と共に示してある．なおグラフの縦横の尺度は，この後各種変換を施した後の実行結果と比較するためにすべて同じにしてある．

Step 0 でのヒープ使用総量が 112.22MB と多いのにもかかわらず図 1 が背の低いグラフになっているのは，生きているクロージャのみを対象にしていて作業領域などの使われたすぐ後に不要になる領域は数えないためである．従って Step 0 では実行時に大量のヒープを割り当てては使い捨てる処理，すなわち不要な中間データ構造を大量に生成しているものとみなすことができる．

3.2　Step 1: 補助関数の定義をローカルに変更

前節のプログラムでは関数 `queens,safe,check` が大域的に定義されていたが，次はこの 3 関数の定義を `main` の `where` 節へ移動することによって局所定義にしたプログラムを実行した．

結果は Step 0 と比較して実行時間が半分以下，ヒープ使用量も 74% 減少した．これは局所的に定義された関数が `main` 内での利用に制限されることにより，さらに最適化を進めることが可能になったためである．我々の HYLO システムで生成されるプログラムもこのようにすべての補助関数を局所的にもつ形をしており，ここで見られたようなローカル定義によるメリットを受けることになる．

3.3　Step 2: システムによる一部の式の融合

HYLO システムは，$f\ (g\ x)$ のような式に対しこれを $(f \circ g)\, x$ であるとみなして自動的に $f \circ g$ の融合変換を行なおうとするが，オプションによってこの機能を無効にし明示的に $f \circ g$ と関数合成で指定された場合にみ融合変換を許すことも可能である．本節ではこの機能を用いて，関数 `safe` の内部までは融合変換させないという条件で実験を行なった．

結果は表 1 より Step 1 とほぼ同等でヒープ量については 15% ほど増加してしまった．これは関数 `safe` を除いた場合，融合変換の可能な場所がほとんど存在しなかったことが原因の 1 つである．また図 2 と図 3 を比較すると生きているクロージャのヒープ上に占める量は大幅に増加しているが，ヒープの総使用量の増加が 15% であったことから考えると，不要な中間データ構造に要した使い捨てヒープの使用量は逆に減少しているものと思われる．

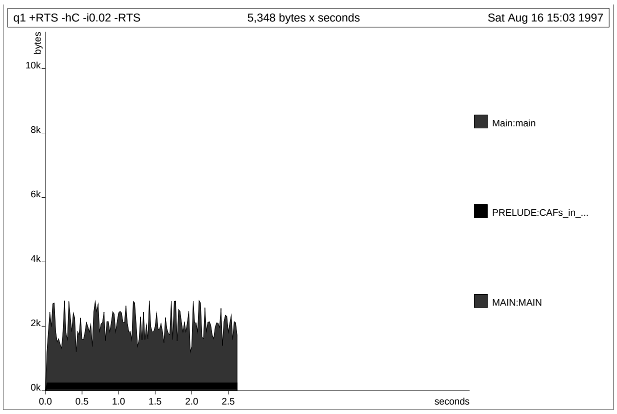

図 2　関数を局所定義にする

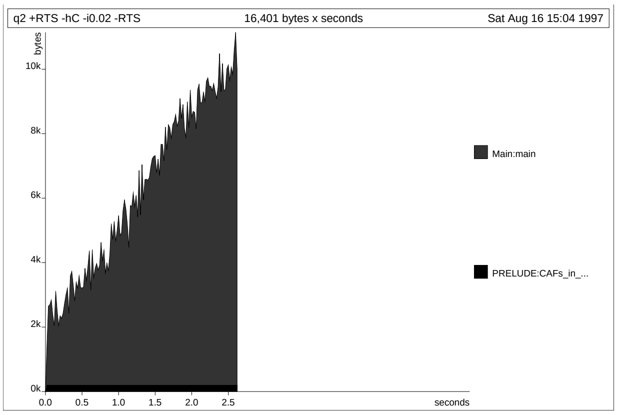

図 3　式 (`safe p n`) を除く融合変換

3.4　Step 3: システムによる融合変換

Step 2 のような制限を加えず HYLO システムですべての可能な融合変換を行なったプログラムを実

行した．その結果は Step 2 と比較して時間で 28%, ヒープ量で 57% の改善となっている．これだけ大幅に改善された理由は，関数 `queen` の中で繰り返し用いられている関数 `safe` について，定義の右辺にある `all not` とリストの内包表記が融合された効果が繰り返しによって増幅されたことにある．またヒーププロファイリングの結果は紙面の都合上省略するが，次節の図 4 を一回り大きくしたものに近い形状をしている．

3.5 Step 4: 再帰関数の構成子式への適用

これまでは 2 つの hylo 関数の融合のみを考えてきたが，プログラムを変換した結果

$$[\![\phi, \eta, \psi]\!]\,(C_i\ t_{i_1} \ldots t_{i_{n_i}})$$

のように構成子式に hylo を適用した形の式が現われることがある．この場合，引数の構成子が判明していることにより実際の関数適用が可能なのでこれを行なうことにする．構成子の引数 $t_{i_1}, \ldots, t_{i_{n_i}}$ の中で再帰的な部分に対しては hylo 関数 $[\![\phi, \eta, \psi]\!]$ が適用されるので，その部分についてはさらに融合変換を進めることも可能になる．

現在システムで生成するプログラムは，この構成子式に対する処理も標準で行なっている．この Step 4 で得られた最終的なプログラムは論文 [Ono97] Fig.10 の関数 `queens_transformed` に相当する．このプログラムの実行結果は表 1 より Step 1 と比較して時間で 42% ヒープ量で 67% も減少している．これより HYLO システムによるプログラム融合変換の有用性が示された．

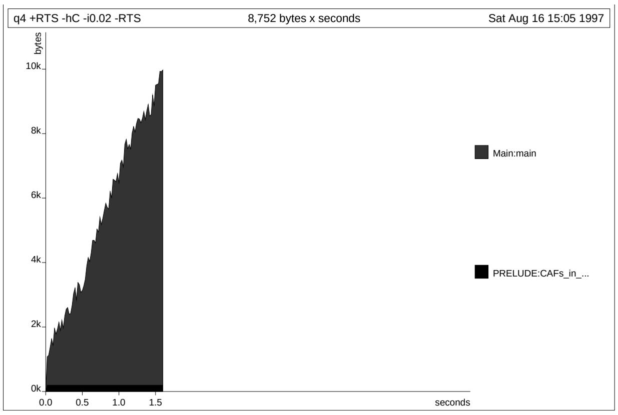

図 4　再帰関数の構成子式への適用

4 考　　察

本研究では n-queens 問題のプログラムに対し HYLO システムで融合変換を行なうことによって，実行時間とヒープ使用量が減少することを確認した．この n-queens 問題のプログラムはチェス盤の様子を整数のリストのリストで表わすのだが，このリスト処理を行なう関数間で受け渡しされる不要な中間データ構造が除去されたことになる．また入れ子になったリストの各段において融合変換が行なわれたことも大幅な効率の改善につながった原因の 1 つといえよう．

今後の課題としては，まず HYLO システムが効果的な変換を行なえるようなプログラムの特徴について調べることが挙げられる．n-queens のように入れ子になったデータ構造の各段で融合変換が可能なものは，大幅に効率が改善できるものと期待される．更に，現在このシステムはパースから最終的にソースコードを出力するまで単体のシステムとして動いているが，実用化に向けて既存のコンパイラの 1 パスとして組み込むことも予定している．

参考文献

[Chi92] Chin, W.: Safe Fusion of Functional Expressions, in *Proc. Conference on Lisp and Functional Programming*, San Francisco, California, 1992.

[Gil93] Gill, A., J. Launchbury, and S. Jones: A Short Cut to Deforestation, in *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 223–232, Copenhagen, 1993.

[Hu95] Hu, Z., H. Iwasaki, and M. Takeichi: Deriving Structural Hylomorphisms from Recursive Definitions, Technical Report METR 95–11, Faculty of Engineering, University of Tokyo, 1995.

[Ono97] Onoue, Y., Z. Hu, H. Iwasaki, and M. Takeichi: A Calculational Fusion System HYLO, in *IFIP TC 2 Working Conference of Algorithmic Languages and Calculi*, 1997.

[She93] Sheard, T. and L. Fegaras: A Fold for all Seasons, in *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 233–242, Copenhagen, 1993.

[Tak95] Takano, A. and E. Meijer: Shortcut Deforestation in Calculational Form, in *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 306–313, La Jolla, California, 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.