HYLO システムによるプログラム融合変換の実現

尾上 能之　胡 振江　武市 正人

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を阻害することになる．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実行時間と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 [1] [7]は，基本的には fold/unfold 変換に基づいて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となる．このメモ化が複雑な処理でありコストが高いため，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 [2] [5]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．最近で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 [3] [6]も提案されてきている．

本研究では，構成的アルゴリズム論の手法を基にして開発された HYLO システム [4]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 [6]を用いる．任意の再帰関数 f，g 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

Implementation of Program Fusion by HYLO System
Yoshiyuki Onoue, Zhenjiang Hu, Masato Takeichi, 東京大学大学院工学系研究科, School of Engineering, University of Tokyo.

コンピュータソフトウェア, Vol.15, No.6 (1998), pp.52–56.
[小論文] 1997 年 8 月 15 日受付.

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, 文法上の緩い制約を満たす任意の再帰関数が hylo 表現で表わせることが知られている [3].

定義 1 (hylo の 3 つ組表現)

F, G を関手とし, 関数 $\phi : GA \to A, \psi : B \to FB$ と自然変換 $\eta : F \dot{\to} G$ が与えられたとき, hylo 関数 $[\![\phi, \eta, \psi]\!]_{G,F} : B \to A$ 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.

$$f = \phi \circ (\eta \circ F f) \circ \psi$$

□

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, 結果を表す再帰データの組立, の役割を果たす. 2つの hylo 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 定理である.

定理 1 (Acid Rain)

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

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

□

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 [4]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よ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. 構文解析, 変数名の付け替え, 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. 定理を適用可能な式が残存 ⇒ 3. へ戻る
6. 最終的に hylo 表現を元の再帰的定義に戻す

3 実験例: n-queens プログラムの最適化

本節では, HYLO システムの効果を調べるために, 例を用いて実際にプログラム変換を行ない, 効率が改善されていく様子を示す. HYLO システムの実現方法については, 本稿では省略するので, 詳しくは [4]を参照されたい. 変換の対象としたのは n-queens のプログラムで, これは [2]でも変換例として挙げられており, 人間が手で変換を行なうにはやや複雑な規模の例であることから, システムの有用性を調べるのに適しているものと思われる.

3.1 Step 0: 初期プログラム

以下の n-queens のプログラムを元にして, 今後プログラム変換を行なっていく.

```
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)
```

実験に用いた計算機は 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.

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. この

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

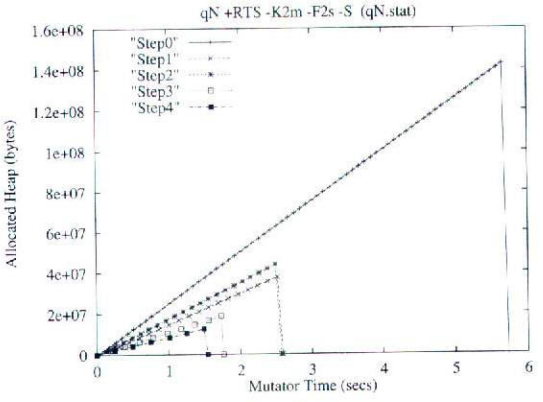

図1 ヒープ割当量の時間変化

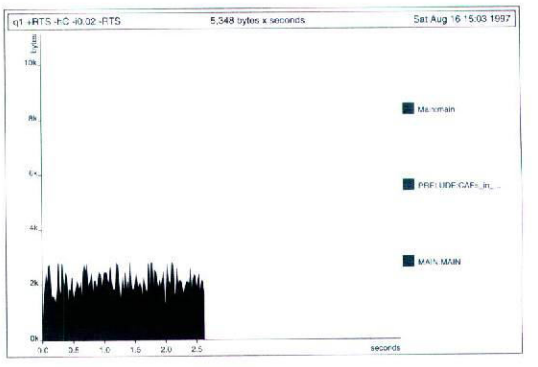

図3 Step 1: 関数を局所定義にする

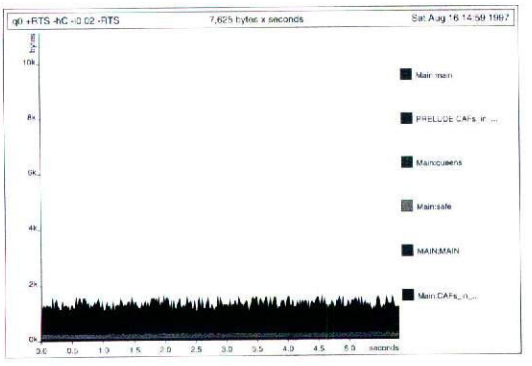

図2 Step 0: 変換前のプログラム

割り当てられたヒープの総量を時間別に図示したものが図1である．GHC のプロファイリング機能には，実行時における生きているクロージャ (live closure) の量を表示する機能はあるが，プログラム開始時からの累積したヒープ割り当て量を表示する機能はないので，GHC のランタイムライブラリ `libHSrts.a` に手を加えることによって図1のデータを出力させた．プログラム終了時のヒープ量が表1の数値よりも大きくなっているのは，図1のデータにはプロファイリングのためのオーバヘッドも含まれているためである．

また図2~5は，GHC のプロファイリング機能を用いて，実行時の生きているクロージャがヒープ中で占める量を時間経過と共に示したものである．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．

図1の Step 0 と図2を比較し，ヒープの累積使用量が多いのにもかかわらず図2が低い水平なグラフになっているのは，図2で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．したがって，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様子がわかる．

3.2 Step 1: 補助関数を局所的に定義

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と思われる．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，

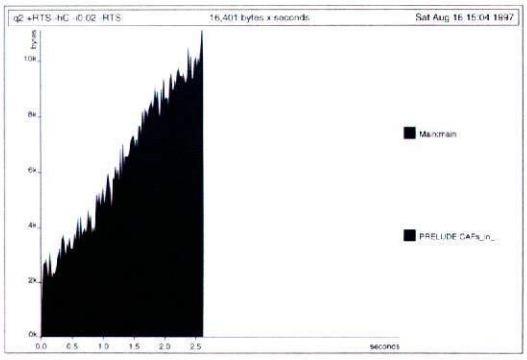

図4 Step 2: 式 (safe p n) を除く融合変換

ここで見られたような局所定義による最適化の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

Step 2 では, Step 1 のプログラムに対し最適化の範囲を限定して融合変換を行なった. 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の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 Step 1 とほぼ同等で, ヒープ量については 15% ほど増加している (表 1). これは, Step 1 におけるリストの内包表記 (list comprehension) が Step 2 では展開されなくなった結果, 内包表記に対するコンパイラの最適化が行なわれなくなったことが原因と思われる. また図 3と図 4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% と少ないことから, 本来は必要のない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. すなわちグラフが右上がりになっていることが, 不要な中間データを生成しないプログラムに改善したことを表わしているといえる.

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 5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,

$$[\![\phi, \eta, \psi]\!]\ (C_i\ t_1 \ldots t_n)$$

のように構成子式 $(C_i\ t_1 \ldots t_n)$ 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場合があるので, Step 4 ではこれを行なうことにする. 通常 hylo 関数の引数は再帰データ型であるため, 構成子の引数 $t_1, \ldots, t_n$ 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 [4]の Fig.10 にある関数 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.

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示した. 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しているが, 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.

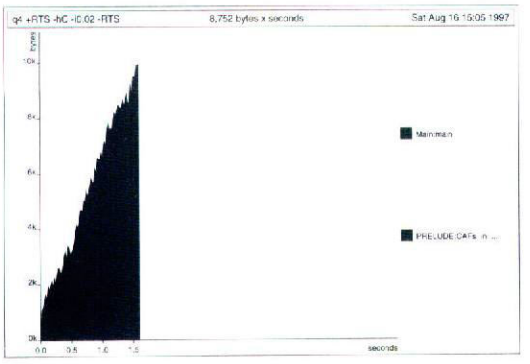

図5 Step 4: 再帰関数の構成子式への適用

今後の課題としては，まず第一により多くのさまざまなプログラム例について同様の実験を行ない，システムの有効性を検証することが挙げられる．これにより，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がわかるであろう．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．さらに，現在このシステムは構文解析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 GHC 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

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

[2] 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.

[3] Hu, Z., Iwasaki, H., and Takeichi, M. : Deriving Structural Hylomorphisms from Recursive Definitions, *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, PA, ACM Press, May 1996, pp. 73–82.

[4] Onoue, Y., Hu, Z., Iwasaki, H., and Takeichi, M. : A Calculational Fusion System HYLO, *IFIP TC 2 Working Conference of Algorithmic Languages and Calculi*, Feburary 17–22, 1997.

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

[6] 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.

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