論文の内容の要旨

論文題目　　融合変換による関数プログラムの最適化

氏　名　　尾上能之

1　研究の背景

関数型言語は，プログラム開発者に対して高いレベルでの柔軟性や信頼性を提供している．強い型付けシステムにより，プログラマは正しいプログラムを書くのが容易になる．遅延評価，抽象データ型，型の多様性のような特徴が，効率的なソフトウェア開発の手伝いをする．しかしその反面，これらの特徴はみな関数プログラム実行効率の悪化に強く影響してしまう．

これらの望ましい言語の特徴を保持したまま高速に実行可能なプログラムを作るには，コンパイラにおける最適化が十分に機能する必要がある．簡潔で抽象的に記述されたプログラムを入力として受け取り，できるだけ効率よくそのプログラムを実行させるようなコンパイラを構築しなければならない．プログラム最適化をプログラム変換で行なう手法では，多くのプログラムの改善が，対象となるプログラムのソースレベルでの変遷によって表わされる．この手法は強力で，いくつかの単純な正当性を維持した変換法則から構成される変換システムによって，多くの強力な最適化が記述され，その中には元のプログラムの複雑さのオーダを変えてしまうようなものまで存在する．

本論文では，関数型言語で記述されたプログラムを効率良く実行するための最適化手法を取り上げる．関数型言語では，プログラムによって作用を記述するのではなく，値を対象としてプログラムを記述する．すなわち，関数型言語で記述されたプログラムは関数定義とその値の操作を含み，これは手続き型言語では動作の列としてプログラムが構成されるのと対照的である．例えば，1 から n までの整数の二乗の和を求めるプログラムを，関数型言語の一つである Haskell で記述すると以下のようになる．

```
sumSq n = sum (map square [1..n])
square x = x * x
```

一方，これを手続き型言語の一つである C で記述すると，以下のようになる．

```
int sumSq(n)
int n;
{
    int i, sum;
    sum = 0;
    for (i=1; i<=n; i++)
      sum += i*i;
    return(sum);
}
```

両者を比べてみると，C のプログラムでは変数 sum に対して加算を繰り返す動作によって，結果を得ていることがわかる．これが Haskell だと，同様の処理を基本的な関数の組み合わせによって簡潔に記述することが可能である．

2 プログラム融合変換

我々の考えを説明するために，先ほど例として挙げた，1 から n までの整数の 2 乗の和を計算する関数 sumSq を考えてみよう．このプログラムは，リスト構造を扱う 3 つの再帰関数の組み合わせで定義されている．$[1..n]$ は 1 から n までの整数のリストを生成し，map square は最初に生成されたリストの各要素を 2 乗したリストを生成し，sum はそのリストの各要素を足した値を計算する．

このプログラムは可読性と高いレベルでの抽象化という点では優れている．プログラム sumSq は，数の生成，数の 2 乗，和の計算という 3 つの異なる処理を組み合わせて記述されているため，相対的に簡潔で書き易く，潜在的に再利用可能であるといえよう．その一方，このプログラムは本来不要であるはずの中間的なリスト構造を 3 つの関数間の情報伝達に用いているため，関数 $[1..n]$ はリスト $[1,2,\cdots,n]$ を関数 map square へ渡し，さらに関数 sum へリスト $[1,4,\cdots,n^2]$ を伝えている．

残念ながら，このような中間データリストは，例え遅延評価の枠組であっても生産され伝えられ捨てられる．これは実行時間やメモリ消費において多大な影響をもたらす．効率的なプログラムを得るためには，3 つの要素関数が 1 つにマージされ，中間データ構造の生成を抑えることが望ましい．これがプログラム融合変換 (program fusion) の目指すところである．

融合変換には大きくわけて 探索ベースの手法と運算的手法の 2 通りのアプローチがある．

伝統的な探索ベースの手法 [Wadler88,Chin92] では，再帰関数の組み合わせを 1 つの関数に変換する際に，unfold/fold 変換 [Burstall&Darlington77] を用いる．我々の例では，再帰関数 $[m..n]$, map square, sum を unfold し，展開された式に対して何らかの法則に基づいてそれを操作し，部分式からこれまでに unfold したパターンにマッチするようなら対応する関数適用で fold することによって新たな関数を得ることができる．

この手法が探索ベースと呼ばれるのは，基本的にすべての現われた関数呼出の履歴を保存し，fold の段階において関数定義を探索しなければならないからである．この関数呼出の履歴を保存しステップの制御を賢く行なわなければならないのは，無限な unfold を避けるためであるが，これにより実質的なコストや複雑度は増し，このためにこの手法を用いたシステムは実験用では存在するものの広く用いられるまでには至っていない．

我々の関心は，運算的 (calculational) な手法 [Takano&Meijer95,Hu et al.96] にある．これは，探索ベースの手法と比べて相対的に新しく注目されている手法で，変換過程に重点をおいた既存のものと異なり，各要素となる関数内に存在する再帰構成子 (hylomorphism, 以下 Hylo と略) の探索に着目している．従って融合変換自体は，単純な変換法則である酸性雨定理 (Acid Rain) [Gill et al.93,Takano&Meijer95] を単に適用するだけで行なうことが可能である．

先程の例でいうと，sumSq は sum . map square . enumFromTo の関数結合で表わされる．なお enumFromTo m n は $[m..n]$ を関数形として明記した等価な表現とする．これを実際に Hylo 融合変換を用いて変換するには，3 つの各再帰関数から Hylo 関数に変形する．次に各 Hylo 関数が融合に適した条件を備えているか判定し，備えているときは酸性雨定理に従って 2 つの Hylo を 1 つの Hylo に変換することができる．この処理を繰り返した上で，最終的に結果として残った Hylo を元の形である再帰的な定義に書き換え直すことによって，Hylo 構造に含まれる非効率的な部分を除去する．この例では，以下のプログラムが変換結果として生成され，これが元のものと比べてより効率的であることは，すべての中間データ構造が除去されたことから明らかである．

```
sumSq n = sumSq' (1,n)
          where
          sumSq' (m,n) = case (m>n) of
                           True  -> 0
                           False -> square m + sumSq' (m+1, n)
```

3 本論文の貢献

運算的な手法は，より現実的なものとして議論されてきたが，我々の知っている範囲では，この手法に基づき実装された融合変換システムはこれまで存在しなかった．この融合変換システムの設計と実装において困難であるのは，以下の 2 点にまとめられる．

- 変換法則を実装するための構成的アルゴリズムの開発
  プログラム運算は，プログラム変換の一種で，豊富に用意された変換法則に基づくプログラムへの操作が適用されることによって進行する．しかしこれらの法則は，基本的に自動変換ではなく手によって変換されるように導かれるよう開発されたものであり，その多くは構成的ではない．したがって，全自動融合変換システムを実装するためには，まずそれらの非構成的変換法則を実装するための構成的アルゴリズムを開発しなければならない．

- 実用的な利用のための実装上の問題点
  実用的な融合変換システムの設計実装にあたって，多くの実用的問題点を考慮しなければならない．とくに言語設計とアルゴリズム設計の 2 点の仕様に焦点をあてることにする．言語設計は，対象とする問題を指定するのに便利なだけでなく，一般的かつ十分な機能をもつものでなくてはならない．アルゴリズム設計は，単純なトイプログラムだけではなく，実用的な規模のプログラムに対しても適用可能なものであるようにしなければならない．

本論文の主な貢献は，これまで理論としてしか示されることのなかった構成的アルゴリズム論に基づく融合変換に対し，その具体的な手順をアルゴリズムとして示すことにある．また，ここで示したアルゴリズムを，広く利用されている既存のコンパイラに最適化処理として埋め込むことによって，その有効性の検証も行なった．これらの事項に関して，以下においてより詳しく説明する．

Hylo融合変換の理論的背景の確立

- 構成的アルゴリズム論に基づく融合変換の実装への試み
  まず，構成的アルゴリズム論に基づいた運算的アプローチの概念を融合変換システムの実装に用いた最初の試みである．これは，理論的な関心にとどまっていた従来の探索型手法とは大きく異なる点である．また， shortcut 融合変換 [Gill et al.93,Gill96] はすでに処理系の内部に実装済で広く利用されているが，カテゴリ理論を用いてさらに拡張している Hylo 融合変換では，リスト以外の中間再帰構造も除去可能であったり，変換対象の関数に予め前処理しておく必要がない，などの利点があることが知られている．

- 融合変換を行なうために不足していたアルゴリズムの提示
  酸性雨定理を適用するために必要な ϕ, ψ 部から τ, σ を導出するためのアルゴリズムを，入力となる式の場合分けとして明確に示した．また Hylo における ψ 側の再構成アルゴリズムを示し，これによって ϕ, ψ の簡略化がより進みさらに融合変換可能な箇所が増えたことを確認した．

汎用処理系に対する Hylo融合変換の実装

- GHC に対する融合変換の実装
  HYLO 融合変換を独自のシステムではなく，既存の Haskell コンパイラ (GHC) 内部に最適化変換の一種として埋め込むことによって，様々な範囲のプログラムに適用可能な一般的システムとすることができた．なお，この実装は Haskell コンパイラに強く依存するものではなく，他の処理系や他の言語にも拡張できることを明記しておく．

- WWW インターフェースの実装
  GHC は効率の良いコードを生成するなどの長所も多く備えているがファイルサイズも大きく，初心者が手軽にインストールして使うには難しい側面もある．したがって，誰にでも我々の融合変換の動作を確認してもらえるように，Web サーバで CGI スクリプトを用意し，外部から我々の Web サーバにアクセスすると CGI を経由して Hylo 融合変換を行なえるような枠組を作成した．

- NoFib ベンチマークに対する効果測定
  これまでに多かった n クィーン問題への融合変換のようなトイプログラムでの例だけではなく，数千行からなるような規模の大きいプログラムも含む Haskell の標準的ベンチマークである NoFib に対して，我々の融合変換の処理を適用し，そのヒープ使用量などの減少度を調べた．また既に実装されている shortcut 融合変換融合変換との比較も行ない，我々の手法の有効性を確認した．

- 実装して明らかになったノウハウの提示
  例えば Hylo 融合変換をどのタイミングで実行するのが一番効果が得られるか，また Hylo 関数の引数が構成子式だった場合に，それを一段簡約することによって融合変換可能な箇所が増え，全体としてどれほど効率が上がるか，などの実装を行なったことによる様々な試行の結果も合わせて提示することができた．