博士論文

融合変換による
関数プログラムの最適化

指導教官　武市 正人 教授

東京大学 大学院工学系研究科

尾上 能之

2002 年 6 月

目次

第 1 章

序論

関数型言語に対する広範囲の関心にもかかわらず，その全体的な実行性能についてはまだいくつかの問題点が存在する．これはなぜなら，関数型言語は命令型言語に比べて実行コストが高く，多くの処理能力とメモリを必要とするからである．

とりわけ関数型言語で推奨されるプログラミングスタイルとして，データ構造上における変換の列としてプログラムを記述する方法がある．このパラダイムでは，プログラマは故意に余計な中間データ構造を導入していることになる．こうすることによってプログラムが簡潔になることに加え，よりモジュラー化されたプログラムを記述することができる．しかしその一方，これが関数型プログラムを実行する際の高いオーバヘッドへと直結してしまうのが問題である．この中間データ構造としては，リストや木などの再帰データ構造であることが多い．本論文では，このような中間データ構造を多用したプログラムに対し，その中間データを**自動的に**除去することによって，プログラムの実行効率を高める手法を扱うものである．

この手法はプログラム融合変換 (program fusion) として広く知られている．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 という再帰パターン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．

1.1 研究の背景

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行なう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．

このように関数型言語は手続き型言語と比べて，以下のような長所を備えている．

- 副作用がない
  バグの混入を未然に防ぐことができる．
- モジュール性
  関数プログラミングでは，モジュール性に優れた再利用可能なコード片を，中間データ構造を媒介として結び付けることにより，大きなプログラムを記述する．これにより関数やデータ型の集まりをモジュールとして構成的に分離することが可能になり，相互に参照可能な要素を制御することによって，構造的なプログラミングが可能になる．
- プログラムが短い
  上のモジュール性とも関連してくることだが，予め用意された多くの組み込み関数を利用することができ，それらを高階関数の機能を用いて上手に組み合わせることによって，手続き型言語のプログラムと比べて，大幅に簡潔なプログラムを記述することが可能である．

しかしその一方，以下のような短所を持ち合わせていることにも注意しなければならない．

- 実行時の効率
  関数型言語はどうしても C のような手続き型言語と比較して，効率が劣ることが多い．これはとくに，無限データ構造やリストを用いた関数合成スタイルなど，関数プログラミング特有の利点を活かしたものに起因することが多い．しかしこのような関数型言語特有の効率上の問題については，これまでの研究により大きな進歩がとげられてきた反面，まだ改良の余地が残されているのも事実である．本論文では，このような関数プログラムを効率良く実行させるための変換技法を提案することが貢献の一つである．
- 副作用の表現
  純粋な関数型言語の特徴は，その反面プログラミングを行なう際の苦手なところとなり得ることもある．時には，命令型言語のような記述方法を用いるほうが目的の動作を指定しやすいこともある．例えば，一意な名前をもつシンボルの生成や，入出力，対話的実行，破壊的代入などが挙げられる．

1.2 関数型言語とは

生産的なソフトウェア開発を行なうには，3 つの衝突する目的を常に考慮する必要があるであろう．信頼できるソフトウェアを開発すること，効率的なソフトウェアを開発すること，すばやくソフトウェアを開発すること，である．信頼できるソフトウェアを開発するには，ソフトウェア工学に対する組織されたアプローチ，すなわち高級言語や構成，抽象化などを使うことを必要とする．これらの技術はソフトウェアが可搬的であり拡張性がある必要がある場合にも本質的となる．しかし一般的な傾向として，高いレベルでの一般性や抽象化を用いると，そのプログラムを実行する際の効率が大きく損なわれることが多い．

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．ここで焦点を絞ることにしよう．例えば，$x+0$ は x と等しいことはわかるが，アルゴリズムの複雑さが改善するように変換するコンパイラは書けるであろうか？

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，本節の後で示すように，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．

透過性

プログラム最適化の技術は一般的に適用可能なものではないが，プログラムがいつ最適化されるかをプログラマが正確に知ることが必要である．ここではこれを**透過性**と呼ぶことにする．

多くの小さな低いレベルでの最適化は，一般的には透過的でない．しかしこれが受け入れられているのはその最適化の効果が，プログラムサイズの増加と同様に，平均的な線に落ち着くからである．しかし特定の最適化がより強力になり，透過性の概念がより重要になってきている．非透過的な最適化は予期しがたい結果を生成してしまいがちである．元のプログラムにおけるわずかな変化が，最適化の適用性が変化することによって，性能面で様々に変化してしまうこともあるであろう．

その主観的な性質のため，透過性は厳密に定義されることはない．実際，多くの最適化は十分な透過性と非透過性の二極の間に位置している．経験則による定義は，最適化が文法的な基準にあうように適用可能であることを保証するとき，これを透過的であるとする．これにより解析技術を用いる多くの最適化が非透過的になる．その一方プログラム変換に基づく最適化は透過的であることが多い．

この経験則には例外もある．例えば部分評価で用いられる束縛時解析は透過的に近い．なぜならプログラムを再構成する際，プログラマはその解析の結果を自由に利用することが可能だからである．プログラマは一般的に，純粋な変換ベースの最適化に安心することが多い．なぜなら最適化の効果が，単に元のプログラムの調査によって決定するからである．この理由により，最適化技術において，透過性が重要な役割をしているとみなすことができる．

中間データ構造の除去

中間的なデータ構造を多く含むアルゴリズムは，宣言的な形式で書かれたプログラムにはよく現われる．式は構成要素すなわちコンビネータで組合せられる．各コンビネータは固有の関数として機能する．

プログラム設計におけるこのようなモジュール性は，この例で示したような低い次元でだけではなく，多くのプログラムの構造において現われる．プログラムの各部分はより小さい成分から築かれていて，それが可読性，拡張性，ソフトウェア再利用性を高めているが，一方実行性能に対しては悪影響を及ぼしている．モジュール化によって発生した非効率性を除去するためにプログラムを書き換えてしまうと，プログラマはそのプログラムの読み易さや保守性を失ってしまうであろう．近代的なソフトウェア工学技術によると，プログラムは容易に保守が可能なように開発し，効率面で残された課題を解決するのはコンパイラ作成者のすべきこと

である，と指摘している．

1.3 プログラム融合変換

プログラム変換は，プログラミングパラダイムの要として提案されてきている．問題の素朴な仕様から効率的なプログラムを導出する手段として，公式かつ機械的にサポートされた流れ [PP93] である．

これは正しく理解しやすく効率的なプログラムを直接書き下そうとするものではない．その代わり，効率の問題を度外視し，できるだけ明白かつ理解しやすいプログラム (仕様) から始まり，変換戦略の集合に基づいて繰り返しプログラム変換を適用することによって，より効率のよいプログラムを生成することを目指す．

融合変換 (program fusion) は，効率的なプログラムを導出するためのプログラム変換戦略の中で最も有力なものの 1 つであり，複数の組み合わせからなるプログラムの断片を 1 つにまとめあげるもので，その一例として，マルチパスプログラムをシングルパスにすることによって，中間的なデータ構造を用いない効率的なプログラムを導くものなどが挙げられる．最近，関数型プログラミングの世界で注目を浴びている変換手法の一つであると言えよう．

この論文の目的は，我々が現在開発中である全自動変換システム HYLO の設計ならびに実装を報告するもので，本システムでは他の変換システムと違い，より体系的かつ一般的な手法で融合変換を実装している点が特徴である．

我々の考えを説明するために，先ほども例として挙げた，1 から n までの整数の 2 乗の和を計算する例 [Wad88] を考えてみよう．

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

このプログラムは 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 要素を組み合わせたものと比べ，相対的に簡潔で書き易く，潜在的に再利用可能である．

その一方，このプログラムは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．これが融合変換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 [Wad88, Chi92] 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 [BD77] 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，運算的な手法 [SF93, GLJ93, TM95, LS95, HIT96b, HIT96a] 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)[GLJ93, TM95] 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が，これを実際に Hylo 融合変換を用いてどのように変換されるかを説明することにしよう．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．融合変換の詳しい手順は以下のようになる．

1. **再帰的定義から *Hylo* の導出**
   まず最初に，要素となるすべての再帰関数を Hylo と呼ばれる特定の再帰形式で現わす．この Hylo は ϕ, η, ψ の 3 つ組を特殊な括弧で囲った $[\![\phi, \eta, \psi]\!]$ のように現わす．
   実際，よく用いられるほとんどすべての再帰関数はこの Hylo を用いて記述することが可能である [BdM94]．また我々は，再帰的に定義された関数から Hylo を導出するアルゴリズムを提案した [HIT96b]．`sum`, `map square`, `enumFromTo` に対して Hylo を導出した後は，以下のような式の組合せになり各 Hylo が各関数に対応することになる．
   $$[\![\phi_1, \eta_1, \psi_1]\!] \circ [\![\phi_2, \eta_2, \psi_2]\!] \circ [\![\phi_3, \eta_3, \psi_3]\!].$$

2. **データ生成と消費の捕獲**
   2 つの Hylo の組合せに対し，酸性雨定理と呼ばれる有効な法則がある (2.2 節参照)．この定理を用いると，2 つの Hylo の組合せは，ある一定の条件の元で 1 つの Hylo に融合することが可能になる．この定理は，$[\![\phi, \eta, \psi]\!]$ に含まれる ϕ と ψ が，それぞれ $\tau\, in$, $\sigma\, out$ のような形で表わされることを期待している．
   ここでは，in がデータ構成子を out がデータ消滅子を表わすものとし，関数 τ, σ は多様型をもち，データ構造の生成と消費の手段 (scheme) を捕獲するために用いられる．酸性雨定理を適用するために，関数 ϕ_i, ψ_i からそれぞれ τ_i, σ_i を導出する必要があり，結果として以下のような式となる．
   $$[\![\phi_1, \eta_1, \sigma_1\, out]\!] \circ [\![\tau_2\, in, \eta_2, \sigma_2\, out']\!] \circ [\![\tau_3\, in', \eta_3, \psi_3]\!]$$

3. **酸性雨定理の適用**
   これまでのステップ 1,2 の結果，融合変換を行なうべき酸性雨定理が適用できる．現在の例に対し後者の関数結合を融合すると，プログラムは以下のようになる．
   $$[\![\phi_1, \eta_1, \sigma_1\, out]\!] \circ [\![\phi_2', \eta_2', \psi_2']\!].$$
   ステップ 2 と 3 を上記の式に繰り返し適用すると，最終的に 1 つの Hylo が導かれる．
   $$[\![\phi_1', \eta_1', \psi_1']\!].$$

4. ***Hylo* の展開**
   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．

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

   結果として生成されたプログラム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かである．

1.4 研究の位置付け

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 [GLJ93, TM95]，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 (3.4章参照)．
- 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 [GLJ93, Gil96] 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るものと期待される．

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，既存のコンパイラ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 融合変換との比較も行なった．
- 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示してある．

1.5 本論文の構成

本論文の構成は以下のようになっている．まず第 2 章において，これまでに行なわれてきた構成的アルゴリズム論に基づくプログラム変換について解説し，必要な記法や定理などについて本節にまとめて記述しておくことにする．ここで hylomoriphim の詳細や，融合変換の際に重要な役割を果たす酸性雨定理などについても触れる．次に第 3 章において，Hylo 融合変換を行なうための流れと，そのために必要なアルゴリズムについて解説する．従来は理論上で融合変換を行なうための条件が示されているだけで，その具体的な手法は示されていなかったのに対し，本論文では融合変換を行なうために必要な手順をアルゴリズムとして明示しているのが特長である．第 4 章では，HYLO システムの実装という面に着目してその概要を与える．実装は既存の Haskell コンパイラである GHC に対し，我々の融合変換の処理を埋め込む形で行なった．これにより広範囲のプログラムに対して，我々の手法を試すことが可能になるという長所がある．そして第 5 章では，プログラム変換の例と実験の結果が示す．これは n クィーン問題のようなトイプログラムだけに留まらず，数千行からなるような大規模な Haskell のプログラムに対しても我々の手法が適用できることを示しているだけではなく，融合変換を行なうことによって実際にヒープ使用量が減少することを確認することにもなっている．第 6 章では，我々の用いた構成的アルゴリズム論に基づく融合変換に加え，従来からある探索型の手法なども含めて，融合変換に関する最近の研究動向を広く紹介する．最後に第 7 章で，HYLO システムに対する結論と，現在残された問題点さらに今後の発展について述べる．

第 2 章

予備知識

本章では，関数プログラミングの基本的な概念や，これまでに行なわれた構成的アルゴリズム論に関する研究の紹介をする．また，この論文中で用いる記法や背景となる定理などについて概説する．

2.1 関数プログラミング

関数型言語では，プログラミングは定義の集まりとして構成され，式を評価するためにコンピュータを用いることになる．プログラマの主な役割は，与えられた問題を解くための関数を構成することであり，この関数は，多くの補助的な関数とともに構成され，通常の数学的な原則にのっとった記法に従って表記される．コンピュータの主な役割は，評価器 (evaluator) または計算器 (calculator) として機能することであり，式を評価しその結果を出力する役目を果たす．この観点からみれば，コンピュータは通常の電卓と何ら変わりはない．このような電卓と関数プログラム計算器の最大の違いは，計算器ではユーザが自分で関数を定義することが可能であり，これがプログラムの表現能力を著しく高めている．式は，プログラマによって定義された関数の名前を含むことが可能で，与えられた定義を用いて評価され，これは式を印字可能な形式に変換する簡略化規則に相当する．

2.1.1 式と値

式 (expression) の概念は，関数プログラミングで中心的役割を果たす．その最大の特徴は，式は値 (value) を表わすためだけに使われるということである．これを言い換えると，式の意味は値であり，その値を得るために実際にどのような手続きが行なわれたとしても，それ以外には何の影響も及ぼさないということになる．さらに，式の値はその式を構成する要素にのみ依存し，その部分式は同じ値をもつ他の式と自由に入れ換え可能である．

簡約 (reduction) という概念は，式が意味を保存したままそのもっとも簡潔な形式になるまで置き換えられることによって評価される過程を意味する．すなわち，基本機能として備わっている規則やユーザが定義で与えた規則を用いることによって，単純な置換と簡略化の処理によって，式の評価は進められる．

2.1.2 型

値の世界は，型 (type) と呼ばれる系統立てされた集まりに分類することが可能である．すべての正しく定義された式には，その式の構成要素のみから推論される一つの型が割り当てられるということは，重要な性質である．この性質はとくに，強い型決め (strong-typing) と呼ばれる．この強い型決めを用いることの最大の利点は，型の割り当てられない任意の式が正しく定義されてない式を表わすことになり，評価の対象から外されることになる．そのような式は値をもたず，たんに不当なものとして扱われる．

2.1.3 関数と定義

関数プログラミングにおいて，もっとも重要な役割をはたす値といえば，それは関数値にほかならない．数学的にいえば，関数 f は，始域の型 S の要素から終域の型 T の 1 つの要素への対応の集まりを表わす値であり，通常

$$f : S \to T$$

として表記される．これは，関数 f が S 型の引数をとり，T 型の結果を返すことを意味する．仮に x を型 S の変数としたとき，関数プログラミングの世界では通常，関数適用 $f(x)$ を $f\ x$ として表わすことが多い．

関数定義の効率

本論文で重要な点として，関数が表わす値とその定義の仕方は異なることがあるということを注意しなければならない．ある目的をすべき関数を考えるとき，その定義方法は数多く存在することになる．例として，引数を 2 倍する関数は，以下の 2 通りの方法で定義するとが可能である．

$$\begin{aligned} double\ \ x &= x + x \\ double'\ x &= 2 \times x \end{aligned}$$

この 2 関数は結果を得るための異なる過程を用いているが，数学的な観点からみれば，$double = double'$ として同じ関数を意味するものとみなせる．しかし通常，一つの関数に対し複数の定義が行なえたとしても，各定義の間には評価する際の効率という点で差異が生じるものである．つまり，評価器は $doublex$ の形の式を $double'x$ よりも効率良く簡約することが可能かもしれない．しかしながら，効率という概念は関数の値それ自身に依存しているものではなく，与えられた定義の形や評価器の特性などの外因によって定まるものである．

定義の形式

多くの場合，関数の値は場合分けを用いて定義することになる．例として最小値を求める関数 min を考えよう．

$$\begin{aligned} min\ x\ \ y &= x, \text{if } x \leq y \\ &= y, \text{if } x \geq y \end{aligned}$$

この定義は 2 つの式から構成され，それぞれがガードと呼ばれる真理値の式で区別される．定義の最初の等式は，式 $x \leq y$ が真となるときに式 $minxy$ の値が x になることを示している．二番目の等式は，式 $x \geq y$ の値が真となるときに，式 $minxy$ の値が y になることを示している．この関数 min と同じ働きをもつ関数は，以下のように otherwise を用いて記述することも可能である．

$$\begin{aligned} min\ x\ \ y &= x, \text{if } x \leq y \\ &= y, \text{otherwise} \end{aligned}$$

また明示的に case 文を用いることによって，以下のようにも定義することができる．

$$\begin{aligned} min\ x\ \ y = &\text{case } x \leq y \text{ of} \\ &True \to x \\ &False \to y \end{aligned}$$

また， if-then-else 構文を用いることも可能である．

$$min\ x\ \ y = \text{if } x \leq y \text{ then } x \text{ else } y$$

他のよく用いられる定義の記法として，局所定義 (local definition) がある．これを用いると，ある部分でのみ用いる局所関数を用いた式を記述することが可能になる．

$$\begin{aligned} f\ x = &\mathit{double}\ x + y \\ &\text{where } \mathit{double}\ x = x + x \\ &\qquad y = 2 + x \end{aligned}$$

ここでは予約語 where を用いることによって，2 つの局所的な定義を f の定義の右辺式を対象とする文脈 (有効範囲) においてのみ利用可能にしている．ここでは where 節全体が，この式の一部分であることを明示するために字下げしてあることに注意されたい．

関数表記の慣習

関数適用は通常引数との間に空白を空けることによって表わし，引数を囲む括弧は用いない．すなわち $f\ a$ が $f\ (a)$ を意味する．関数はカリー化されており，左側に優先的に結合する．すなわち，$f\ a\ b$ は $(f\ a)\ b$ を意味する．関数適用の優先度は，他のどの演算子よりも強いものとみなされ，$f\ a \oplus b$ は $f\ (a \oplus b)$ ではなく $(f\ a) \oplus b$ として解釈される．

関数結合演算は，小さい丸記号 $\circ$ を用いて表わす．この演算子の定義は $(f \circ g)\ a = f\ (g\ a)$ である．関数結合は結合則をもった演算子であり，以下の関係が成り立つ．

$$f \circ (g \circ h) = (f \circ g) \circ h$$

この結合性をいかして，通常複数の関数結合が組み合わせて用いられる場合は，括弧を省略して表記することが多い．関数結合演算子の単位元は，すべての要素をそれ自身に返す恒等関数であり，これを id で表わす．

$$id \circ f = f \circ id = f$$

の関係が任意の関数 f に対して成立する．

一般的な二項演算子を表わすものとして，$\oplus, \otimes, \odot$ などを用いることにする．二項演算子はセクション化することが可能で，$\oplus$ のような中置二項演算子から，以下のような一引数関数を導くことも可能である．

$$(a \oplus)\ b = a\ \oplus\ b = (\oplus\ b)\ a = (\oplus)\ a\ b$$

前置記法の二項演算子 f は，バッククオートで囲むことによって，中置演算子として利用することができる．

$$f\ x\ y = x\ `f`\ y$$

二つの関数 $f : S \to T$ と $g : S \to T$ があったとき，もしその二関数がすべての引数に対して同じ結果を返すとき，これらを等しい関数とみなす．

$$\frac{\forall x \in S.\ \ f\ x = g\ x}{f = g}$$

この規則とは逆に，二つの関数が等しければその関数を同じ引数に適用した結果も等しいという関係も成り立つ．

$$\frac{f = g}{\forall x \in S.\ \ f\ x = g\ x}$$

2.2 構成的アルゴリズム論

再帰は関数定義において重要な役割を担い，かつ表現される定義式の形式を制限することはほとんどない．最近では，多くの研究 [MFP91, SF93] によって，再帰がある特定の形式で構造化されることが示されている．

Hylo はそのような特定の再帰形式の 1 つであり，ふだん利用されているほぼすべての再帰関数がこの形式で現わされることがわかっている [MFP91, BdM94, HIT96b]．概要としては，Hylo とは関数が以下のような再帰的手法で定義されることをいう．

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

この式の右辺は以下のように読むことができる．まず ψ によって入力から構造 F を生成する．次に $F\, f$ によって，構造 F におけるすべての再帰要素に対して関数 f を適用する．その後，η で構造 F を構造 G へと変換させる．最後に，ϕ で構造 G をまとめあげるとそれが最終的な結果となる．ϕ, η, ψ, G, F が定まると，元の関数 f が一意に決定されることから，今後 $f = [\![\phi, \eta, \psi]\!]_{G,F}$ のように現わすことにする．

Hylo は多くの算術的法則があることが知られている ()2.5節参照) ので，プログラム変換を行なうのが容易になっている．詳しくは，プログラム運算に関する既存研究 [MFP91, Fok92, TM95] をみることにしよう．

2.3 関手 (Functors)

内関手 (endofunctor) は，型定義においてデータ構造と制御構造の両方を捕獲することができる．本論文では，すべてのデータ型は以下の 4 つの基本関手から構成される内関手によって定義されるものと仮定する．このような内関手は多様関手 (polynomial functor) として知られている．

- 型 X 上の**恒等**関手 I とその動作は以下のように定義される:

$$I\ X = X, \quad I\ f = f$$

- 型 X 上の**定数**関手 $!A$ とその動作は以下のように定義される:

$$!A\ X = A, \quad !A\ f = id$$

  ここで id は恒等関数を現わす．

- 2 つの型 X, Y 上の**積**関手 $X \times Y$ とその動作は以下のように定義される:

$$\begin{array}{lcl} X \times Y & = & \{(x, y) \mid x \in X,\, y \in Y\} \\ (f \times g)\ (x, y) & = & (f\, x,\, g\, y) \\ \pi_1\ (a, b) & = & a \\ \pi_2\ (a, b) & = & b \\ (f \vartriangle g)\ a & = & (f\, a,\, g\, a) \end{array}$$

- 2 つの型 X, Y 上の**直和**関手 $X + Y$ とその動作は以下のように定義される:

$$\begin{array}{lcl} X + Y & = & \{1\} \times X \cup \{2\} \times Y \\ (f + g)\ (1, x) & = & (1,\, f\, x) \\ (f + g)\ (2, y) & = & (2,\, g\, y) \\ (f \triangledown g)\ (1, x) & = & f\, x \\ (f \triangledown g)\ (2, y) & = & g\ y. \end{array}$$

ここでは積と直和を 2 引数に対して定義したが，一般の n 引数に対しても自然な形で拡張できる．例えば，n 引数に対する直和は以下のように定義される．

$$\sum_{i=1}^{n} X_i = \bigcup_{i=1}^{n} (\{\mathsf{i}\} \times X_i)$$

$$\left(\sum_{i=1}^{n} f_i\right)\ (\mathsf{j}, x) = (\mathsf{j},\, f_j\, x) \qquad \text{for } 1 \le \mathsf{j} \le n$$

2.4 関手の初期不動点に対応するデータ型

データ型は，その型の各要素がどのように構成されるかを示す各データ構成子の集まりである．データ型の定義は内関手によって捕獲される [Fok92]．ここでは具体的な例として，各要素の型が A であるようなコンスリストのデータ型を考えると，その型は以下のように定義される．

$$List\ A = Nil\ |\ Cons\,(A,\ List\ A).$$

我々の枠組では，以下のような多様型をもつ関手を用いて，データ型の再帰構造を捕獲する．

$$F_{L_A} = !\mathbf{1}\ +\ !A \times I$$

ここで $\mathbf{1}$ は最終オブジェクトを現わし () に対応する．厳密に言えば，Nil は 0 引数の構成子として $Nil\,()$ と記述すべきで，本論文では $t\,()$ の形式は簡略化して t と略記することにする．

実際，F_{L_A} の定義は $List\ A$ の元の定義から自動的に導出可能である [SF93]．それに加えて，$List\ A$ のデータ構成子である $in_{F_{L_A}}$ を以下のように定義する．

$$in_{F_{L_A}} = Nil \triangledown Cons.$$

従って $List\ A = in_{F_{L_A}}\,(F_{L_A}\,(List\ A))$ となる．$in_{F_{L_A}}$ の逆関数は $out_{F_{L_A}}$ と表記され，$List\ A$ のデータ消滅子 (data destructor) は以下のようになる．

$$\begin{array}{lcl} out_{F_{L_A}}\ Nil & = & (1,()) \\ out_{F_{L_A}}\ (Cons\,(a,as)) & = & (2,(a,as)). \end{array}$$

他の例として，二分木のデータ型は以下のようになり，

$$Tree\ a\ =\ Leaf\ |\ Node\ (a, Tree\ a, Tree\ a),$$

以下の内関手によって捕獲される．

$$F_T\ =\ !\mathbf{1}\ +\ !a \times I \times I.$$

一般的に，関手 F はその最小不動点としてデータ型を決定するので，F によって決まるデータ型を μF と表記することにする．

2.5 Hylomorphism

Hylomorphism (以降 Hylo と略) は，有名な *catamorphism* や *anamorphism* を特別な場合として含むような一般的な再帰構造を表わし，以下のような三つ組で定義される [TM95].

Definition 1 (Hylomorphism in triplet form)
F,G を 2 つの関手とする．与えられた $\phi : G\,A \to A$, $\psi : B \to F\,B$ と自然変換 $\eta : F \dot{\to} G$ に対し，$\mathrm{Hylo}[\![\phi, \eta, \psi]\!]_{G,F} : B \to A$ は，以下の等式の最小不動点として定義される．

$$f = \phi \circ \eta \circ F\,f \circ \psi$$

□

Hylo は優れた記述力を持ち，実用的に用いられる多くの再帰関数 (例, 基本関数) をこの形式で表わすことが可能である [HIT96b]．これは，効率的に関数プログラムを操作するために理想的な再帰形式であるとみなすことができる．なお，文脈から自明な場合，添字の G,F を省略することにする．

Hylo は非常に一般的な形式をしているが，その特別な場合として以下の形式は多くの場合用いられることがある．

Definition 2 (Catamorphism $(\!|-|\!)$***, Anamorphism*** $[\!(-)\!]$***)***

$$\begin{array}{lcl}(\!|\phi|\!)_F & = & [\![\phi, id, out_F]\!]_{F,F} \\ [\!(\psi)\!]_F & = & [\![in_F, id, \psi]\!]_{F,F}\end{array}$$

□

Hylo には多くの有用な変換法則が知られている．例えば *Hylo Shift Law* は以下のようになる．

$$[\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi \circ \eta, id, \psi]\!]_{F,F} = [\![\phi, id, \eta \circ \psi]\!]_{G,G}$$

この法則は，自然変換が Hylo の内部で自由に移動可能であるという非常に便利な性質を示している．また他の有用な法則として酸性雨定理 (*Acid Rain Theorem*)[TM95] が知られている．この法則は 2 つの Hylo の結合が 1 つに融合される性質を示している．

Theorem 1 (Acid Rain)

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

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

□

第 3 章

運算的融合変換

本節では，HYLO システムを実現するためのいくつかの重要なアルゴリズムを示す．とくにデータ生成と消費を捕獲する多様型関数を導出するためのアルゴリズムに焦点をあてることにする．Hylo の導出や展開に関するアルゴリズムは，[HIT96b] で述べられたものを拡張して用いることにする．

3.1 HYLO 融合変換の概要

図 3.1 に，現在開発中の HYLO 変換システムの流れを示す．このシステムは関数型言語 Gofer [Jon94] で記述されており，Haskell など他のシステムにも容易に移植可能なように設計されている．HYLO システムは Gofer で書かれたプログラムを入力として受け取り，融合変換を用いることによって不要な中間データ構造を生成しないように効率の改善されたプログラムを結果として出力する．

HYLO システムは最初に入力として，関数型言語 Gofer [Jon94] で記述されたプログラムを受け取る．そし

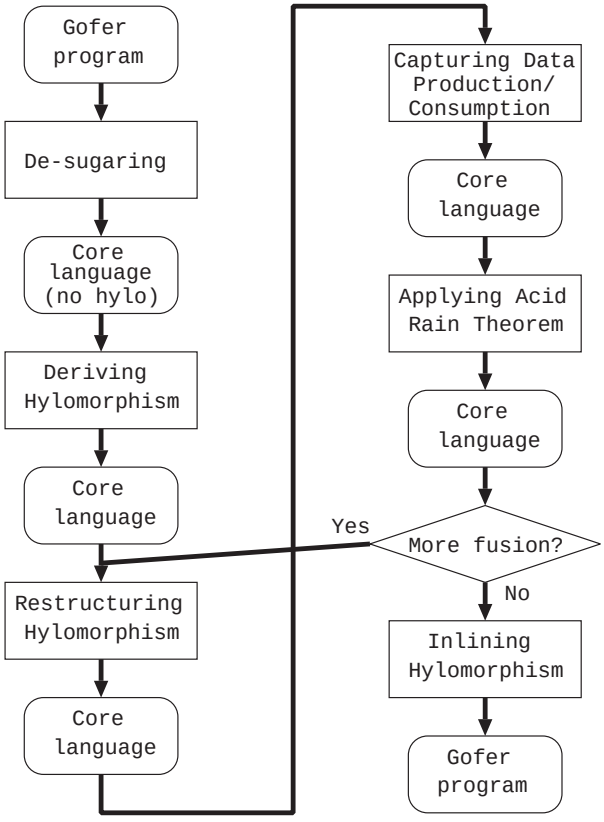


図 3.1. Overview of HYLO System

$Prog$	::=	$Def;\ Prog \mid Def$	Program
Def	::=	$v = t$	Definition (non-mutual)
t	::=	v	Variable
	$\mid$	l	Literal (Int,Float,Char,...)
	$\mid$	$(t_1, \ldots, t_n)$	Term tuple
	$\mid$	$\lambda v_s.t$	Lambda expression
	$\mid$	let $v = t_1$ in t_0	Let expression (non-recursive)
	$\mid$	case t_0 of	Case expression
		$p_1 \to t_1; \cdots; p_n \to t_n$	
	$\mid$	$v\ t_1 \cdots t_n$	Function application (saturated)
	$\mid$	$C\ t_1 \cdots t_n$	Constructor application (saturated)
	$\mid$	$t_0\ t_1$	Application (general)
	$\mid$	$[\![t_\phi, t_\eta, t_\psi]\!]$	Hylo representation
	$\mid$	(n, t)	Expression with tag
v_s	::=	$v \mid (v_1, \cdots, v_n)$	Argument bounded by λ
p	::=	v	Variable
	$\mid$	$(p_1, \ldots, p_n)$	Tuple
	$\mid$	$C\ p_1 \cdots p_n$	Constructor pattern

図 3.2. HYLO Core Language

て Gofer の処理系に手を加えたものがこのシステムの front end として機能し，パーズ，変数名の付替，型チェック，desugar などの処理をまとめて行なう．その結果 core 言語で記述されたプログラムが新たに生成される．

本システムが扱う core 言語を図 3.2 に示す．この言語は関数型言語 GHC [Tea96] の処理系の内部で用いられている言語に類似するように作られている．主な違いは本論文で提案する hylo 関数を追加したものになっており，この式が融合変換の際に重要な役割を果たす．説明を簡略化するため，用いるデータ構造は単一再帰のみに，再帰関数は相互再帰を含まないように限定することにする．これは標準的な組化手法によって，相互再帰に定義された関数は相互再帰を含まない形に変換できることが知られているので，一般性を損ねることはない．さらに入れ子構造になった case 式はそれを平坦化 (flatten) したものに変換済であることを仮定する．

このように GHC の内部表現に近い言語を用いているので，将来的には既存のコンパイラに対して，最適化処理の一部として組み込み易くなっているのが特長である．

次に core 言語のプログラムを受け取ると，この中に含まれる再帰的に定義された関数が hylo 関数へと変換される．2.5 節で説明したように，hylo 関数は再帰構造を内包するような表現で，この一般的な形式を用いることによって，2 つの再帰関数間の合成を定める有用な規則を適用することができるようになる．この hylo 関数の導出に関するアルゴリズムの詳しい説明は，3.2 節で行なう．この処理の後からは，hylo 関数も含まれる core 言語のプログラムが変換の対象となる．

次に hylo 関数の再構成を行なう．hylo 関数の合成を行なうには，関数がその合成に適した構造を持ってい

ないといけないのだが，この再構成を行なうことにより合成に適した構造を導きやすくなる．再構成の処理は，hylo 関数 $[\![\phi, \eta, \psi]\!]$ において ϕ, ψ の部分から再帰に依存しない計算部分を抽出し，η に移動することによって行なわれる．この処理を行なうことによって ϕ, ψ は簡略化され，次の τ, σ の導出の処理が容易になる．この hylo 関数の再構成に関するアルゴリズムの詳しい説明は，3.3 節で行なう．

次に ϕ, ψ からデータの生成と消費を捕獲する．これにより，ϕ, ψ はそれぞれ $\tau in_F, \sigma out_F$ の形に変換され，Acid Rain 定理を適用することができるようになる．τ, σ は ϕ, ψ から型 μF に依存する部分を抽象化した関数になっており，引数に応じて型を決定する．このデータの生成と消費の捕獲に関するアルゴリズムの詳しい説明は，3.5,3.5.2 節で行なう．

次に Acid Rain 定理を適用し関数の合成を行なう．定理を適用する毎に hylo 関数は 1 つずつ減少していくので，この定理の適用は有限回で終了する．これ以上合成できる関数が存在しなくなったときは，次の展開 (inlining) 処理に進む．まだ合成すべき hylo 関数が残っているときは，ここで新たに生成された hylo 関数に対して再構成の処理を行なう．Acid Rain 定理の適用については， 3.6 節で述べる．

最後に，これまで合成定理を適用するために用いてきた hylo 表現をより親しみやすい再帰関数による定義に inline 展開し，Gofer のプログラムとして出力する．この処理はシステムの back end として機能し，front end と back end の部分をそのまま既存の関数型言語のコンパイラで置き換えれば，hylo–fusion の機能を備えたコンパイラを生成することができる．

3.2 Hylo の導出

hylo を導出するアルゴリズムは，[HIT96b] に示されているアルゴリズムを，我々の言語に合うように拡張したものを用いている．この拡張したアルゴリズムを図 3.3 に示す．詳しい説明は [HIT96b] にあるので省略するが，大まかに説明すると，このアルゴリズムによって典型的な再帰関数定義から hylomorphism が以下のように導出される．

$$f = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to t_1; \cdots; p_n \to t_n$$

$$\begin{aligned}
f = \quad & \{\text{Definition of } f\} \\
& \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to t_1; \cdots; p_n \to t_n \\
= \quad & \{\text{Trick 1: Replacing } t_i \text{ with } g_i\ t_i'\} \\
& \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to g_1\ t_1'; \cdots; p_n \to g_n\ t_n' \\
= \quad & \{\text{Using separated sum}\} \\
& (g_1 \triangledown \cdots \triangledown g_n) \circ (\lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1,\ t_1'); \cdots; p_n \to (\mathsf{n},\ t_n')) \\
= \quad & \{\text{Trick 2: Replacing } g_i \text{ with } \phi_i \circ F_i\ f\} \\
& (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ ((F_1 + \cdots + F_n)\ f) \circ \\
& \qquad (\lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1,\ t_1'); \cdots; p_n \to (\mathsf{n},\ t_n')) \\
= \quad & \{\text{Auxiliary definitions } \phi, F, \text{and } \psi\} \\
& \phi \circ F\ f \circ \psi \\
= \quad & \{\text{Definition of hylomorphism}\} \\
& [\![\phi, id, \psi]\!]_{F,F} \\
& \text{where } F = F_1 + \cdots + F_n \\
& \qquad \phi = \phi_1 \triangledown \cdots \triangledown \phi_n \\
& \qquad \psi = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1,\ t_1'); \cdots; p_n \to (\mathsf{n},\ t_n')
\end{aligned}$$

トリック 1, 2 は，もし各 t_i に対して $t_i = (\phi_i \circ F_i\ f)\ t_i'$ を満たすような ϕ_i, F_i, t_i' を見つけることができれば，この定義から Hylo が導出可能であることを示している．この導出は，各項 t_i におけるアルゴリズム $\mathcal{D}$

$$
\begin{aligned}
&\mathcal{A}[\![\lambda v_{s_1} \cdots \lambda v_{s_m}.\text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!]\, f \\
&\qquad = \lambda v_{s_1} \cdots \lambda v_{s_{m-1}}.\, [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F} \\
&\quad \textbf{where } (\{v_{i_1}, \ldots, v_{i_{k_i}}\}, \{(v'_{i_1}, t_{i_1}), \cdots, (v'_{i_{l_i}}, t_{i_{l_i}})\}, t'_i) = \mathcal{D}[\![t_i]\!]\, \{\}\, f \ (i = 1, \ldots, n) \\
&\qquad \phi_i = \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ t'_i \\
&\qquad t''_i = (v_{i_1}, \ldots, v_{i_{k_i}}, t_{i_1}, \ldots, t_{i_{l_i}}) \\
&\qquad \psi = \lambda v_{s_m}.\text{case } t_0 \text{ of } p_1 \to (1, t''_1); \cdots; p_n \to (\mathsf{n}, t''_n) \\
&\qquad F_i = !\mathbf{1}, \text{ if } k_i = l_i = 0 \\
&\qquad F_i = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} \ \ (I_1 = \cdots = I_{l_i} = I), \text{otherwise} \\
&\qquad F = F_1 + \cdots + F_n \\
&\qquad \Gamma(v) = \text{return } v\text{'s type}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{D}[\![v]\!]\, s_l\, f &= \text{if } v \in \mathit{global\_vars} \cup s_l \text{ then } (\{\}, \{\}, v) \text{ else } (\{v\}, \{\}, v) \\
\mathcal{D}[\![l]\!]\, s_l\, f &= (\{\}, \{\}, l) \\
\mathcal{D}[\![(t_1, \ldots, t_n)]\!]\, s_l\, f &= (s_1 \cup \cdots \cup s_n, c_1 \cup \cdots \cup c_n, (t'_1, \ldots, t'_n)) \\
&\quad \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f \ (i = 1, \ldots, n) \\
\mathcal{D}[\![\lambda v_s.t]\!]\, s_l\, f &= (s, c, \lambda v_s.t') \\
&\quad \text{where } (s, c, t') = \mathcal{D}[\![t]\!]\, (s_l \cup Var(v_s))\, f \\
\mathcal{D}[\![\text{let } v = t_1 \text{ in } t_0]\!]\, s_l\, f &= (s_0 \cup s_1, c_0 \cup c_1, \text{let } v = t'_1 \text{ in } t'_0) \\
&\quad \text{where } (s_1, c_1, t'_1) = \mathcal{D}[\![t_1]\!]\, s_l\, f, (s_0, c_0, t'_0) = \mathcal{D}[\![t_0]\!]\, (s_l \cup \{v\})\, f \\
\mathcal{D}[\![\text{case } t_0 \text{ of } p_1 \to t_1 &= (s_0 \cup \cdots \cup s_n, c_0 \cup \cdots \cup c_n, \text{case } t'_0 \text{ of } p_1 \to t'_1; \cdots; p_n \to t'_n) \\
; \cdots; p_n \to t_n]\!]\, s_l\, f &\quad \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!](s_l \cup Var(p_i))f \ (i{=}0, \ldots, n), p_0{=}() \\
\mathcal{D}[\![v\ t_1 \cdots t_n]\!]\, s_l\, f &= \text{if } v = f \text{ then} \\
&\qquad \text{if } n = m \ \wedge\ t_i = v_{s_i} \ (1 \le \forall i \le m-1) \\
&\qquad\ \text{then } (\{\}, \{(u, t_m)\}, u) \\
&\qquad\ \text{else } \mathit{error} \text{ (f should have saturated args and induct on last)} \\
&\quad \text{else } (s_0 \cup \cdots \cup s_n, c_0 \cup \cdots \cup c_n, t'_0\ t'_1 \cdots t'_n) \\
&\quad \text{where } (s_0, c_0, t'_0) = \mathcal{D}[\![v]\!]\, s_l\, f \\
&\qquad\quad (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f, (i = 1, ..., n) \\
&\qquad\quad u \text{ is a fresh variable} \\
\mathcal{D}[\![C\ t_1 \cdots t_n]\!]\, s_l\, f &= (s_1 \cup \cdots \cup s_n, c_1 \cup \cdots \cup c_n, C\ t'_1 \cdots t'_n) \\
&\quad \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f \ (i = 1, \ldots, n) \\
\mathcal{D}[\![t_0\ t_1]\!]\, s_l\, f &= (s_0 \cup s_1, c_0 \cup c_1, t'_0\ t'_1)) \\
&\quad \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f \ (i = 1, 2) \\
Var(x) &= \text{set of variables in } x
\end{aligned}
$$

図 3.3. Deriving Hylomorphism

に強く依存している．これは結果として，t_i の中の自由変数の集合で f の再帰呼出に現われないもの，t_i の中の未使用変数と関数 f の引数のペアの集合，f の再帰呼出において未使用変数に対応して置換された項，からなる三つ組を返す．ここで関数 f がその最後の引数について再帰をしていて他の引数は各再帰呼出の間変更されないという仮定をしているが，これは一般性を損ねてはいないことに注意したい．アルゴリズム $\mathcal{A}$ は名前変更プロセスが終了したプログラムに適用されるので，すべての束縛変数は一意な変数になっているとする．

どのようにこのアルゴリズムが働くかを，1.3節の ssf のプログラムを例にして解説する．ssf は，関数 $sum, map, upto$ を用いていて，パターンマッチを用いて以下のように定義される．

$$\begin{array}{rcl} sum & = & \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil \to 0 \\ & & \qquad\qquad Cons\ (a, as) \to plus\ (a, sum\ as) \\ map & = & \lambda g.\ \lambda xs.\text{case}\ xs\ \text{of}\ Nil \to Nil \\ & & \qquad\qquad Cons\ (a, as) \to Cons\ (g\ a,\ map\ g\ as) \\ upto & = & \lambda(m, n).\ \text{case}\ (m > n)\ \text{of}\ True \to Nil \\ & & \qquad\qquad False \to Cons\ (m, upto\ (m + 1, n)). \end{array}$$

ここでこの定義から sum を選び，アルゴリズム $\mathcal{A}, \mathcal{D}$ を順に適用すると以下のようになる．

$$\begin{array}{c} \mathcal{D}[\![0]\!]\ \{\}\ sum = (\{\}, \{\}, 0) \\ \mathcal{D}[\![plus\ (a, sum\ as)]\!]\ \{\}\ sum = (\{a\}, \{(v_1', sum\ as)\}, plus\ (a, v_1')) \end{array}$$

ここで

$$\begin{array}{ll} t_1' = (), & \phi_1 = \lambda().\ 0 = 0 \\ t_2' = (a, as), & \phi_2 = \lambda(a, v_1').\ plus\ (a, v_1') = plus \end{array}$$

そして

$$\begin{array}{rcl} \psi & = & \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil \to (1, ()) \\ & & \qquad\qquad Cons\ (a, as) \to (2, (a, as)) \\ & = & out_F \\ F & = & !\mathbf{1} + !Int \times I. \end{array}$$

これより，以下のような sum に対する Hylo を導出できた．

$$sum \quad = \quad [\![0 \triangledown plus, id, out_F]\!]_{F,F}.$$

同様に，$map, upto$ に対する Hylo も導出できる．

$$\begin{array}{rcl} map & = & \lambda g.\ [\![\phi, id, out_F]\!]_{F,F} \\ & & \text{where}\ \phi = \lambda().\ Nil \triangledown \lambda(v, v').\ Cons\ (g\ v, v') \\ upto & = & [\![in_F, id, \psi]\!]_{F,F} \\ & & \text{where}\ \psi = \lambda(m, n).\ \text{case}\ (m > n)\ \text{of} \\ & & \qquad\qquad True \to (1, ()) \\ & & \qquad\qquad False \to (2, (m, (m + 1, n))) \end{array}$$

ここで

$$\begin{array}{rcl} F & = & !\mathbf{1} + !Int \times I\ (=\ F_{L_{Int}}) \\ in_F & = & Nil \triangledown Cons \\ out_F & = & \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil \to (1, ()) \\ & & \qquad\qquad Cons\ (a, as) \to (2, (a, as)) \end{array}$$

3.3 Hylo の再構成

一般的に，Hylo$[\![\phi, \eta, \psi]\!]_{G,F}$ の振舞いは以下のように解釈することができる: ψ は再帰構造を生成し，η はある構造を他の構造へと変換し，ϕ は生成された再帰構造を結果となるよう操作する．ここで ϕ と ψ は η が行なう計算を含むことが可能である．再構成の目的は，ϕ, ψ からできる限り多くの計算を抽出し，それを η へ

移動することである．この操作により，ϕ, ψ が単純化され，次の段階で行なうデータの生成/消費の手順を示す τ, σ の導出が容易になるという利点がある．ϕ に対する基本的な再構成のアルゴリズムは，[HIT96b] に示されているが，そのアルゴリズムを拡張したものを図 3.4 に示す．[HIT96b] では提示されていなかった ψ に対する再構成のアルゴリズムは，図 3.5 に示す．

$\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$ を再構成するアルゴリズム $\mathcal{S}_\phi$ は，ラムダ式 ϕ_i の本体 t_i に対して外部アルゴリズムである $\mathcal{E}$ を適用する．アルゴリズム $\mathcal{E}$ は各 t_i の中で再帰変数を含まない最大部分項を検出し，その部分項を新しい変数で置き換えた新しい項 t'_i を生成し，$\mathcal{S}_\phi$ がそれを新しいラムダ式 ϕ'_i の本体とする．詳細をみると，アルゴリズム $\mathcal{E}$ が入力として項 t_i と再帰変数の集合を受け取り，結果として以下のペアを返す．ペアの中身は，対応する新しい変数で置き換えられた再帰変数を含まないような最大部分項の集合と，項 t_i に含まれる部分項を対応する新しい変数で置換して生成された新しい項の 2 要素から構成される．この部分項で行なわれる計算の処理は，これによって η の要素に移動したことになる．

例として，前節で得られた以下の Hylo を再構成してみよう．

$$map\ g = [\![Nil \triangledown \lambda(a, v'_1).\ Cons\,(g\,a, v'_1), id, out]\!]$$

この例の場合，$\phi_1 = \lambda().\ Nil$ で $\phi_2 = \lambda(a, v'_1).\ Cons\,(\underline{g\,a}, v'_1)$ となる．ここで v'_1 は再帰変数であり，唯一の極大非再帰部分項には下線が引いてある．我々のアルゴリズムを適用すると，$g\,a$ が ϕ_2 の外へと移動されて，

$$\begin{aligned}\phi_2 &= \phi'_2 \circ \eta_{\phi_2} \\ &\quad \text{where } \phi'_2 = \lambda(u_{2_1}, v'_1).\, Cons(u_{2_1}, v'_1) \\ &\qquad\ \ \eta_{\phi_2} = \lambda(a, v'_1).\, (g\,a, v'_1)\end{aligned}$$

最終的に以下の構造をもつ Hylo となる．

$$map\ g = [\![Nil \triangledown \lambda(u_{2_1}, v'_1).\ Cons(u_{2_1}, v'_1), id + \lambda(a, v'_1).(g\,a, v'_1), out]\!]$$

この変換は，$(map\ g)$ の ϕ を $Nil \triangledown \lambda(u_{2_1}, v'_1).\ Cons(u_{2_1}, v'_1)$ (i.e., in) のように簡略化していて，これにより酸性雨定理の適用が容易になる．

双対的に，アルゴリズム $\mathcal{S}_\psi$ は ψ を以下のような典型的な形式に再構成する．

$$\psi = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1, t_1); \cdots; p_n \to (\mathsf{n}, t_n).$$

これは，t_i の構成要素で再帰部分 (恒等関手 I に対応している) に関係をもたない $t_{i_1}, \cdots, t_{i_{k_i}}$ から自由変数を抽出する．新しい項 t'_i は単に自由変数の供給元としてのみ動作し，実際の計算は生成された η_{ψ_i} によって行なわれる．η_{ψ_i} を η 部に移動することによって，ψ から多くの計算が取り除かれることになる．

3.4 データ生成/消費の手法の捕獲

融合変換を行なう酸性雨定理を適用するために，与えられた Hylo$[\![\phi, \eta, \psi]\!]_{G,F} : A \to B$ に含まれる ϕ と ψ が，それぞれ $\tau\, in_{F_B}$ や $\sigma\, out_{F_A}$ のように記述される必要がある．ここで τ, σ は多様型関数で，F_A, F_B はそれぞれ型 A, B を定義する関手である．このような τ, σ を導出するアルゴリズムは以下のようになり，その証明は [HIT96b] で示されている．

***Theorem 2* (多様型関数の導出)**
以上の条件のもとに，τ, σ は以下の 2 つの法則によって定義される．

$$\frac{\forall \alpha.\ \ ([\![\alpha]\!])_{F_B} \circ \phi = \phi' \circ G\,([\![\alpha]\!])_{F_B}}{\tau = \lambda \alpha.\ \phi'}, \qquad \frac{\forall \beta.\ \ \psi \circ [\![\beta]\!]_{F_A} = F\,[\![\beta]\!]_{F_A} \circ \psi'}{\sigma = \lambda \beta.\ \psi'}$$

□

$$\mathcal{S}_\phi[\![[\![\phi_1 \triangledown \cdots \triangledown \phi_n, \eta, \psi]\!]_{G,F}]\!] = [\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, (\eta_{\phi_1} + \cdots + \eta_{\phi_n}) \circ \eta, \psi]\!]_{G',F}$$

where

$$G_1 + \cdots + G_n = G$$
$$!\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} = G_i$$
$$\lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ t_i = \phi_i \text{ (assume } v'_{i_1}, \ldots, v'_{i_{l_i}} \text{ are recursive variables)}$$
$$(\{(u_{i_1}, t_{i_1}), \ldots, (u_{i_{m_i}}, t_{i_{m_i}})\}, t'_i) = \mathcal{E}[\![t_i]\!]\{v'_{i_1}, \ldots, v'_{i_{l_i}}\}$$
$$\eta_{\phi_i} = \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).(t_{i_1}, \ldots, t_{i_{m_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}})$$
$$G'_i = !\Gamma(u_{i_1}) \times \cdots \times !\Gamma(u_{i_{m_i}}) \times I_1 \times \cdots \times I_{l_i}$$
$$G' = G'_1 + \cdots + G'_n$$
$$\phi'_i = \lambda\ (u_{i_1}, \ldots, u_{i_{m_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ t'_i$$

Assume that u is a fresh variable in the following:

$$\begin{aligned}
\mathcal{E}[\![v]\!]\ s_r &= \text{if } Var_{s_r}(v) \text{ then } (\{(u, v)\}, u) \text{ else } (\{\}, v) \\
\mathcal{E}[\![l]\!]\ s_r &= (\{(u, l)\}, u) \\
\mathcal{E}[\![(t_1, \ldots, t_n)]\!]\ s_r &= \text{if } \forall i. Var_{s_r}(t'_i) \text{ then } (\{(u, (t_1, \ldots, t_n))\}, u) \\
&\qquad \text{else } (w_1 \cup \cdots \cup w_n, (t'_1, \ldots, t'_n)) \\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r\ (i = 1, \ldots, n), \\
\mathcal{E}[\![\lambda v.t]\!]\ s_r &= \text{if } Var_{s_r}(t') \text{ then } (\{(u,\ \lambda v.t)\}, u) \text{ else } (w, \lambda v.t') \\
&\quad \text{where } (w, t') = \mathcal{E}[\![t]\!]\ s_r \\
\mathcal{E}[\![\text{let } v = t_1 \text{ in } t_0]\!]\ s_r &= \text{if } Var_{s_r}(t'_0) \wedge Var_{s_r}(t'_1) \text{ then } (\{(u, \text{let } v = t_1 \text{ in } t_0)\}, u) \\
&\qquad \text{else } (w_0 \cup w_1, \text{let } v = t'_1 \text{ in } t'_0) \\
&\quad \text{where } (w_1, t'_1) = \mathcal{E}[\![t_1]\!]\ s_r,\ \ (w_0, t'_0) = \mathcal{E}[\![t_0]\!]\ s_r \\
\mathcal{E}[\![\text{case } t_0 \text{ of } p_1 \to t_1 & = \text{if } \forall i. Var_{s_r}(t'_i) \text{ then } (\{(u, \text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n)\}, u) \\
\quad ; \cdots ; p_n \to t_n]\!]\ s_r & \qquad \text{else } (w_0 \cup \cdots \cup w_n, \text{case } t'_0 \text{ of } p_1 \to t'_1; \cdots; p_n \to t'_n) \\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r\ (i = 0, \ldots, n), \\
\mathcal{E}[\![v\ t_1 \cdots t_n]\!]\ s_r &= \text{if } \forall i. Var_{s_r}(t'_i) \text{ then } (\{(u, v\ t_1 \cdots t_n)\}, u) \\
&\qquad \text{else } (w_0 \cup \cdots \cup w_n, t'_0\ t'_1 \cdots, t'_n) \\
&\quad \text{where } (w_0, t'_0) = \mathcal{E}[\![v]\!]\ s_r,\ \ (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r\ (i = 1, \ldots, n), \\
\mathcal{E}[\![C\ t_1 \cdots t_n]\!]\ s_r &= \text{if } \forall i. Var_{s_r}(t'_i) \text{ then } (\{(u, C\ t_1 \cdots t_n)\}, u) \\
&\qquad \text{else } (w_1 \cup \cdots \cup w_n, C\ t'_1 \cdots t'_n) \\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r\ (i = 1, \ldots, n), \\
\mathcal{E}[\![t_0\ t_1]\!]\ s_r &= \text{if } Var_{s_r}(t'_0) \wedge Var_{s_r}(t'_1) \text{ then } (\{(u, t_0\ t_1)\}, u) \\
&\qquad \text{else } (w_0 \cup w_1, t'_0\ t'_1)) \\
&\quad \text{where } (w_0, t'_0) = \mathcal{E}[\![t_0]\!]\ s_r,\ \ (w_1, t'_1) = \mathcal{E}[\![t_1]\!]\ s_r \\
\mathcal{E}[\![[\![t_0, t_1, t_2]\!]_{F_0,F_1}]\!]\ s_r &= \text{if } \forall i. Var_{s_r}(t'_i) \text{ then } (\{(u, [\![t_0, t_1, t_2]\!]_{F_0,F_1})\}, u) \\
&\qquad \text{else } (w_0 \cup w_1 \cup w_2, [\![t'_0, t'_1, t'_2]\!]_{F_0,F_1})) \\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r\ (i = 0, 1, 2), \\
\mathcal{E}[\![(\mathsf{n}, t)]\!]\ s_r &= \text{if } Var_{s_r}(t') \text{ then } (\{(u, (\mathsf{n}, t))\}, u) \text{ else } (w, (\mathsf{n}, t')) \\
&\quad \text{where } (w, t') = \mathcal{E}[\![t]\!]\ s_r \\
Var_{s_r}(t) &= t \text{ is a variable } \ \wedge\ \ t \notin s_r
\end{aligned}$$

図 3.4. Restructuring Hylomorphisms — ϕ part

$$
\begin{array}{l}
\mathcal{S}_\psi[\![[\![\phi,\eta,\psi]\!]_{G,F}]\!] = [\![\phi,\eta\circ(\eta_{\psi_1}+\cdots+\eta_{\psi_n}),\psi']\!]_{G,F'} \\
\quad \text{where} \\
\quad\quad \lambda\, v_s.\text{case } t_0 \text{ of } p_1 \to (1,t_1);\cdots;p_n \to (\mathsf{n},t_n) = \psi \\
\quad\quad F_1+\cdots+F_n = F \\
\quad\quad !\Gamma(t_{i_1})\times\cdots\times !\Gamma(t_{i_{k_i}})\times I_1\times\cdots\times I_{l_i} = F_i \\
\quad\quad (t_{i_1},\ldots,t_{i_{k_i}},tt_{i_1},\ldots,tt_{i_{l_i}}) = t_i \\
\quad\quad \{v_{i_1},\ldots,v_{i_{m_i}}\} = \mathcal{FV}[\![(t_{i_1},\ldots,t_{i_{k_i}})]\!]\, global\_vars \\
\quad\quad \eta_{\psi_i} = \lambda(v_{i_1},\ldots,v_{i_{m_i}},v'_{i_1},\ldots,v'_{i_{l_i}}).(t_{i_1},\ldots,t_{i_{k_i}},v'_{i_1},\ldots,v'_{i_{l_i}}) \\
\quad\quad t'_i = (v_{i_1},\ldots,v_{i_{m_i}},tt_{i_1},\ldots,tt_{i_{l_i}}) \\
\quad\quad F'_i = !\Gamma(v_{i_1})\times\cdots\times !\Gamma(v_{i_{m_i}})\times I_1\times\cdots\times I_{l_i} \\
\quad\quad F' = F'_1+\cdots+F'_n \\
\quad\quad \psi' = \lambda\, v_s.\text{case } t_0 \text{ of } p_1 \to (1,t'_1);\cdots;p_n \to (\mathsf{n},t'_n)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{FV}[\![v]\!]\, s_g & = & \text{if } v\in s_g \text{ then } \{\} \text{ else } \{v\} \\
\mathcal{FV}[\![l]\!]\, s_g & = & \{\} \\
\mathcal{FV}[\![(t_1,\ldots,t_n)]\!]\, s_g & = & s_1\cup\cdots\cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\, s_g\ (i=1,\ldots,n) \\
\mathcal{FV}[\![\lambda v.t]\!]\, s_g & = & \mathcal{FV}[\![t]\!]\,(s_g\cup\{v\}) \\
\mathcal{FV}[\![\text{let } v=t_1 \text{ in } t_0]\!]\, s_g & = & s_0\cup s_1 \\
 & & \text{where } s_1 = \mathcal{FV}[\![t_1]\!]\, s_g,\ \ s_t = \mathcal{FV}[\![t_0]\!]\,(s_g\cup\{v\}) \\
\mathcal{FV}[\![\text{case } t_0 \text{ of } p_1\to t_1 & = & s_0\cup\cdots\cup s_n \\
;\cdots;p_n\to t_n]\!]\, s_g & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\,(s_g\cup Pat(p_i))\ (i=0,\ldots,n),\ \ p_0=() \\
\mathcal{FV}[\![v\ t_1\cdots t_n]\!]\, s_g & = & s_0\cup\cdots\cup s_n \\
 & & \text{where } s_0 = \mathcal{FV}[\![v]\!]\, s_g,\ \ s_i = \mathcal{FV}[\![t_i]\!]\, s_g\ (i=1,\ldots,n) \\
\mathcal{FV}[\![C\ t_1\cdots t_n]\!]\, s_g & = & s_1\cup\cdots\cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\, s_g\ (i=1,\ldots,n) \\
\mathcal{FV}[\![t_0\ t_1]\!]\, s_g & = & s_0\cup s_1 \\
 & & \text{where } s_0 = \mathcal{FV}[\![t_0]\!]\, s_g,\ \ s_1 = \mathcal{FV}[\![t_1]\!]\, s_g \\
\mathcal{FV}[\![[\![t_0,t_1,t_2]\!]_{F_0,F_1}]\!]\, s_g & = & s_0\cup s_1\cup s_2 \\
 & & \text{where } s_0 = \mathcal{FV}[\![t_0]\!]\, s_g,\ \ s_1 = \mathcal{FV}[\![t_1]\!]\, s_g,\ \ s_2 = \mathcal{FV}[\![t_2]\!]\, s_g \\
\mathcal{FV}[\![(\mathsf{n},t)]\!]\, s_g & = & \mathcal{FV}[\![t]\!]\, s_g
\end{array}
$$

図 3.5. Restructuring Hylomorphisms — ψ part

この定理は，一般的ではあるが，構成的な手法で (すなわちアルゴリズムとして) どのように ϕ' や ψ' を見つけるかが明らかには示されていない．従って，自動計算融合変換システムを実装するためには，この非構成的な変換法則を実現するための，構成的なアルゴリズムを開発する必要がある．本節では，このための基本的アルゴリズムを提案し，τ の導出について詳細に議論する．なお双対の概念である σ の導出についても少し触れることにする．

3.5 τ 導出のためのアルゴリズム

2 つの Hylo の関数合成 $[\![\_,\_,out_F]\!]_{\_,F}\circ[\![\phi,\_,\_]\!]_{F',\_}$ が酸性雨定理 (定理 1 (a)) によって融合されるとき，ϕ は，データ構成子を抽象化するための多様型関数 τ を用いて，$\tau\, in_F = \phi$ をみたすように記述されなければならない．型付けについて，ϕ は $F'\mu F \to \mu F$ の型をもつことがわかるが，これは ϕ は F' 構造で再帰要素が

型 μF であるようなデータを受け取り，型 μF のデータを結果とすることを示す．ϕ から τ を導出するためには，型 μF のデータ構成子を検出しそれを抽象化することが必要である．

より正確には，$C_1, \cdots, C_n$ を型 μF のデータ構成子とし (このとき $in_F = C_1 \triangledown \cdots \triangledown C_n$)，変数 $c_1, \cdots, c_n$ を $C_1, \cdots, C_n$ を抽象化するためのラムダ変数とする．ϕ に現われる $C_1, \cdots, C_n$ をラムダ抽象することによって τ を導出し，その後 ϕ を τ を in_F に適用した式で置換する．すなわち $\phi = \tau\, in_F$ で $\tau = \lambda(c_1 \triangledown \cdots \triangledown c_n).\ \phi'$ となり，これは定理 2 をみたす．

説明を簡潔にするため，これから τ を導出しようとする関数 ϕ を以下のような特定な形をするものと仮定するが，これは一般性を損ねてはいない．

$$\phi_1 \triangledown \cdots \triangledown \phi_k : (F'_1 + \cdots + F'_k)\mu F \to \mu F,$$

そして各 $\phi_i : F'_i \mu F \to \mu F$ は以下のラムダ抽象とする:

$$\phi_i = \lambda(v_{i_1}, \cdots, v_{i_{m_i}}\, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i \tag{3.1}$$

ここで $F'_i = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{m_i}}) \times I_1 \times \cdots \times I_{l_i}$，すなわち定数関手の後に恒等関手が続くようにする．ここで恒等関手 I に対応するラムダ変数は，通常**再帰変数** と呼ぶことにし，変数に明示的にプライム ($'$) を付けることにする．明らかに，各 ϕ_i に対して τ_i を導出することができたなら，ϕ に対する τ は $\tau = \lambda cs.\ (\tau_1\ cs\ \triangledown \cdots \triangledown\ \tau_k\ cs)$ となることがわかる．従って，我々の関心は ϕ_i から等式 (3.1) によって定義される τ_i を導出すればよいことがわかる．すべての関数 ϕ に対して τ の存在が保証されているわけではないことが知られているので，ϕ に対するいくつかの制限が必要になってくる．適切な制限を選択するために 2 つの必要条件がある．1 つは，与えられた ϕ がその制限をみたしているかを自動的に判定可能である必要がある．もう 1 つは，その記述能力を大きく制限してはいけないということである．ここで [SF93] における**基準項** *(canonical term)* の概念を借りて，その項 t_i が以下の形式から構成されるものとして ϕ_i を制限する．

1. **変数**: v, ラムダ抽象において束縛される
2. **構成子適用**: $C\ t'_1\ \cdots\ t'_n$, C はデータ構成子，各 t'_i は項
3. *hylo* **適用**: $[\![\phi'_1 \triangledown \cdots \triangledown \phi'_{n'}, \eta, out]\!]\ v'$, ϕ'_i は制限された形式，v' は再帰変数
4. **関数適用**: $f\ t_1\ \cdots\ t_n$, f は大域関数，各項 t_i は再帰変数への参照を含まない

ここで ϕ_i は制限が厳しいようにみえるが，より一般的な ϕ_i はこの形式に変換することができることに注意されたい．まず第一に，3.3節の再構成アルゴリズムがとても役立つ．なぜなら ϕ の中で，結果となるべき型 μF のデータの形成に関係しない多くの計算を移動することが ϕ の簡潔化に大いに貢献するからである．実際，再構成の後は，制限された項は最後の形式，すなわち関数適用を含まないことがわかる．これはこの形式の項はすべて抽出され η の要素へ移動されるからである．第 2 に，$let, case$ などいくつかの構成要素や非再帰の関数適用は考慮する必要はない．なぜならこれらは前処理で除去することが可能だからである．局所定義は大域領域へ移動 (lift) 可能であるし，非再帰関数の適用は unfold 処理によって展開できる．第 3 に，これは驚くことではなく説得できることであるが，すべての潜在的に正規化されたプログラムは，十分に実用的な記述力をもったものであったとしても，自動的にこの形式へ変換することが可能であることが示されている [SF93].

この制限の助けにより，我々のアルゴリズムはより簡潔になり，アルゴリズムの正しさの証明が簡単になる．これから，我々のアルゴリズム $\mathcal{F}'$ が ϕ_i から τ_i を導出することを示す:

$$\phi_i = \tau_i\, in_F, \quad \text{where } \tau_i = \lambda(c_1 \triangledown \cdots \triangledown c_n).\ \lambda(v_{i_1}, \cdots, v_{i_{m_i}}\, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ \mathcal{F}'[\![t_i]\!]$$

このラムダ抽象は，t_i における μF に現われた適切な各構成子を対応するラムダ変数で置き換える．アルゴリズム $\mathcal{F}'$ を図 3.6に示す．これは t_i の種類に応じて，以下のいずれかの処理を行なうことによって進められる．なお $\mathcal{F}'$ が適用されるすべての式は，型 μF でなければならないことに注意したい．

Assume: $F = F_1 + \cdots + F_n$, $in_F = C_1 \triangledown \cdots \triangledown C_n$ and $c_1, \cdots, c_n$ are fresh variables.

$\phi \quad : \quad F'\mu F \to \mu F$

$\phi \quad \Rightarrow \quad \tau in_F \quad$ where $\tau = \mathcal{F}[\![\phi]\!]$

where

$$
\begin{aligned}
\mathcal{F}[\![\phi_1 \triangledown \cdots \triangledown \phi_k]\!] &= \lambda cs.\ (\mathcal{F}[\![\phi_1]\!]\ cs \triangledown \cdots \triangledown \mathcal{F}[\![\phi_k]\!]\ cs) \\
\mathcal{F}[\![\lambda(v_{i_1}, \cdots, v_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\, t_i]\!] & \\
&= \lambda(c_1 \triangledown \cdots \triangledown c_n).\lambda(v_{i_1}, \cdots, v_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\mathcal{F}'[\![t_i]\!] \\
\\
\mathcal{F}'[\![v]\!] &= (\!|c_1 \triangledown \cdots \triangledown c_n|\!)_F\ v \quad (*\ v \text{ is not a recursive variable } *) \\
\mathcal{F}'[\![v']\!] &= v' \qquad\qquad\qquad\qquad (*\ v' \text{ is a recursive variable } *) \\
\mathcal{F}'[\![C_i\ t'_1\ \cdots\ t'_{n'}]\!] &= c_i(F_i\ \mathcal{F}'\ (t'_1, \cdots, t'_{n'})) \\
\mathcal{F}'[\![[\![\phi', \eta, out_F]\!]_{F',F}\ v]\!] &= [\![\mathcal{F}[\![\phi']\!]\ (c_1 \triangledown \cdots \triangledown c_n), \eta, out_F]\!]_{F',F}\ v \\
\mathcal{F}'[\![f\ t'_1\ \cdots\ t'_{n'}]\!] &= (\!|c_1 \triangledown \cdots \triangledown c_n|\!)\ (f\ t'_1\ \cdots\ t'_{n'})
\end{aligned}
$$

図 3.6. Deriving τ

1. t_i が変数のとき: それが再帰変数なら何もしない．その他の場合，その変数が μF の構成子を含む式に束縛されている可能性があるので，その式に Cata$(\!|c_1 \triangledown \cdots \triangledown c_n|\!)_F$ を適用し，μF の各構成子をラムダ変数 c_i によって置換する処理を行なう必要がある．
2. t_i が構成子適用 $C_i\ t'_1 \cdots t'_{n'}$ のとき: C_i は対応するラムダ変数 c_I で置換されなければならない．その引数に対して，アルゴリズム $\mathcal{F}'$ が再帰的に適用される．
3. t_i が hylo 適用 $[\![\phi', \eta, out_F]\!]_{F',F}\ v'$ のとき: $\mathcal{F}'$ ではなく $\mathcal{F}$ を ϕ' へ再帰的に適用する．
4. t_i が関数適用 $f\ t_1\ \cdots\ t_n$ のとき: $(\!|c_1 \triangledown \cdots \triangledown c_n|\!)$ を全体に適用する．

3.5.1 アルゴリズム $\mathcal{F}$ の正当性

ϕ における構成子を抽象化するとき，なぜ単に ϕ における μF の構成子のすべての出現を，対応するラムダ変数で置換してしまわないのか，不思議に思われるかもしれない．実際，特別な注意が払われなければならないことを示そう．まず第一に，ϕ における μF の構成子には 2 種類ある．1 つは置換されるべき構成子で，もう 1 つはそうでない構成子である．この 2 つの違いをみるために，以下の例を考えることにしよう．

$$\phi = \lambda().\ Nil \triangledown \lambda(v, v').\ Cons\ (Cons\ (v, Nil), v').$$

この ϕ に対する正しい τ は以下のように導かれる．

$$\tau = \lambda(c_1 \triangledown c_2).\ (\lambda().\ c_1 \triangledown \lambda(v, v').\ c_2\ (\underline{Cons\ (v, Nil)}, v')).$$

これより，リストの要素の中に含まれる構成子 (下線部) は置換してはいけないことがわかる．図 3.6 に示した構成子適用に対する規則において，関手 F_i に指定された適切な引数にのみ $\mathcal{F}'$ を再帰的に適用することによって，この事実を考慮していたことになる．もう 1 点，ϕ の非再帰引数の構成を考慮しなければならない点にも注意しなければならない．なぜならこれらの引数は，最終的な結果を生成するために実際に用いられるからである．このような場合，Cata を適用することによって，この段階で構成子を置換する必要がある．

***Theorem 3* (アルゴリズム $\mathcal{F}$ の正当性)** 図 3.6 のアルゴリズムにおいて，以下が成り立つ．

$$\begin{array}{ll}\text{(i)} & \phi = \mathcal{F}[\![\phi]\!]\ in_F \\ \text{(ii)} & \mathcal{F}[\![\phi]\!] : \forall A.(FA \to A) \to (F'A \to A)\end{array}$$

証明の概略． (i) の証明は直接的である．$c_1, \cdots, c_n$ をそれぞれ $C_1, \cdots, C_n$ で回復させ，$(\![C_1 \triangledown \cdots \triangledown C_n]\!)_F$ は型 μF における恒等関数である事実を用いると，$\mathcal{F}[\![\phi]\!]\ in_F$ から ϕ を得ることができる．

(ii) に関しては，もし任意の $\phi : F'(\mu F) \to \mu F$ に対して以下が成り立つことがいえれば，定理 2 により正しいことがわかる．

$$\forall \alpha.\ (\![\alpha]\!] \circ \phi = (\mathcal{F}[\![\phi]\!]\ \alpha) \circ F'\,(\![\alpha]\!].$$

これは ϕ の構造における帰納法により，以下のように証明することが可能である．

Case $\phi = \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).v_i$:

$$\begin{array}{ll} & \forall \alpha.\ (\![\alpha]\!] \circ \phi = (\mathcal{F}[\![\phi]\!]\ \alpha) \circ F'\,(\![\alpha]\!] \\ \equiv & \{\text{Note } F' = !\Gamma(v_1) \times \cdots \times !\Gamma(v_m) \times I_1 \times \cdots \times I_n \text{ and by Def. of } \mathcal{F} \text{ and } \phi\} \\ & \forall \alpha.\ (\![\alpha]\!] \circ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).v_i = \\ & \quad ((\lambda(c_1 \triangledown \cdots \triangledown c_n).\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).(\![c_1 \triangledown \cdots \triangledown c_n]\!)\ v_i)\ \alpha) \circ \\ & \quad (id_1 \times \cdots \times id_m \times (\![\alpha]\!]_1 \times \cdots \times (\![\alpha]\!]_n) \\ \equiv & \{\text{Simplification}\} \\ & \forall \alpha.\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).(\![\alpha]\!]\ v_i = (\lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).(\![\alpha]\!]\ v_i) \circ \\ & \quad (id_1 \times \cdots \times id_m \times (\![\alpha]\!]_1 \times \cdots \times (\![\alpha]\!]_n) \\ \equiv & \{\text{Obvious}\} \\ & \text{True}\end{array}$$

Case $\phi = \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).v'_i$:

$$\begin{array}{ll} & \forall \alpha.\ (\![\alpha]\!] \circ \phi = (\mathcal{F}[\![\phi]\!]\ \alpha) \circ F'\,(\![\alpha]\!] \\ \equiv & \{\text{Note } F' = !\Gamma(v_1) \times \cdots \times !\Gamma(v_m) \times I_1 \times \cdots \times I_n \text{ and by Def. of } \mathcal{F} \text{ and } \phi\} \\ & \forall \alpha.\ (\![\alpha]\!] \circ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).v'_i = \\ & \quad ((\lambda(c_1 \triangledown \cdots \triangledown c_n).\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).v'_i)\ \alpha) \circ \\ & \quad (id_1 \times \cdots \times id_m \times (\![\alpha]\!]_1 \times \cdots \times (\![\alpha]\!]_n) \\ \equiv & \{\text{Simplification}\} \\ & \forall \alpha.\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).(\![\alpha]\!]\ v'_i = (\lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).v'_i) \circ \\ & \quad (id_1 \times \cdots \times id_m \times (\![\alpha]\!]_1 \times \cdots \times (\![\alpha]\!]_n) \\ \equiv & \{\text{Simplification}\} \\ & \forall \alpha.\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).(\![\alpha]\!]\ v'_i = \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n.\ (\![\alpha]\!]_i\ v'_i) \\ \equiv & \{\text{Obvious}\} \\ & \text{True}\end{array}$$

Case $\phi = \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).\ C_i\ t'_1 \cdots t'_n$:

$$\begin{array}{ll} & \forall \alpha.\ (\![\alpha]\!] \circ \phi = (\mathcal{F}[\![\phi]\!]\ \alpha) \circ F'\,(\![\alpha]\!] \\ \equiv & \{\text{Note } F' = !\Gamma(v_1) \times \cdots \times !\Gamma(v_m) \times I_1 \times \cdots \times I_n \text{ and by Def. of } \mathcal{F} \text{ and } \phi\} \\ & \forall \alpha.\ (\![\alpha]\!] \circ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).\ C_i\ t'_1 \cdots t'_n = \\ & \quad ((\lambda(c_1 \triangledown \cdots \triangledown c_n).\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).c_i\,(F_i\ \mathcal{F}'(t_1, \cdots, t'_n))\ \alpha) \circ F'(\![\alpha]\!] \\ \equiv & \{\text{We can assum that } \alpha = \alpha_1 \triangledown \cdots \triangledown \alpha_n \text{ for its type}\} \\ & \forall \alpha.\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).\ \alpha_i\,(F_i(\![\alpha]\!](t'_1 \cdots, t'_n)) = \\ & \quad (\lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).\alpha_i\ (F_i\ \mathcal{F}'(t'_1, \cdots, t'_n))) \circ F'(\![\alpha]\!] \\ \equiv & \{\lambda \text{ related transformation}\} \\ & \forall \alpha.\ \lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).\ \alpha_i\,(F_i(\![\alpha]\!](t'_1 \cdots, t'_n)) = \\ & \quad (\lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).\ \alpha_i \\ & \quad (F_i\ \mathcal{F}'((\lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).t'_1) \circ F'(\![\alpha]\!], \cdots, (\lambda(v_1, \cdots, v_m, v'_1, \cdots, v'_n).t'_n) \circ F'(\![\alpha]\!])))) \\ \equiv & \{\text{By induction}\} \\ & \text{True}\end{array}$$

□

この点を明白にするため，リストの代わりに木の生成の抽象化に関する例をみてみることにする．関数 $mkTree$ はリストを受け取り平衡木を生成するものと仮定する:

$$\begin{array}{rcl} mkTree & = & \lambda xs.\ \text{case}\ xs\ \text{of} \\ & & \quad Nil \rightarrow Leaf \\ & & \quad Cons\ (a, as) \rightarrow Node\ (double\ a, squareNodes\ (mkTree\ as), mkTree\ as) \end{array}$$

ここで $squareNodes$ は以下の Hylo であるものとする:

$$\begin{array}{c} squareNodes = [\![\lambda().Leaf\ \triangledown\ \ \lambda(v, v_1', v_2').\ Node(v, v_1', v_2'), \\ id + \lambda(v, v_1', v_2').(square\ v, v_1', v_2'),\ \ out_{F_T}]\!]_{F_T, F_T} \end{array}$$

F_T は 2.4 節の木の型を定義する関手である．Hylo 導出のアルゴリズムを適用すると，以下の式を得る:

$$\begin{array}{l} mkTree = [\![\phi, id, out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ \quad \text{where}\ \phi = \lambda().\ Leaf\ \triangledown\ \lambda(v, v').\ Node\ (double\ v, squareNodes\ v', v') \end{array}$$

この式は以下ように再構成される:

$$\begin{array}{l} mkTree = [\![\phi, id + \lambda(v, v').(double\ v, v'), out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ \quad \text{where}\ \phi = \lambda().\ Leaf\ \triangledown\ \lambda(v, v').\ Node\ (v, squareNodes\ v', v'). \end{array}$$

ここでアルゴリズム $\mathcal{F}$ を適用し，$\phi : F_L(\mu F_T) \rightarrow \mu F_T$ に対する τ を導出すると，以下の結果が得られる:

$$\begin{array}{rcl} mkTree & = & [\![\tau\ in_{F_T}, id + \lambda(v, v').(double\ v, v'), out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ & & \text{where}\ \tau = \lambda(c_1 \triangledown c_2).\ (\lambda().\ c_1 \triangledown \\ & & \qquad \lambda(v, v').\ c_2\ (v, [\![\lambda().c_1\ \ \triangledown\ \ \lambda(v, v_1', v_2').c_2\ (v, v_1', v_2'), \\ & & \qquad\ id + \lambda(v, v_1', v_2').(square\ v, v_1', v_2'), out_{F_T}]\!]_{F_T, F_T}\ v', v')) \end{array}$$

3.5.2 σ 導出のアルゴリズム

酸性雨定理を適用するもう一つの場合は，$[\![\_, \_, \psi]\!]_{\_, F'} \circ [\![in_F, \_, \_]\!]_{F, \_}$ のような 2 つの Hylo の結合を融合することがある．ここでは，ψ から σ が $\sigma out_F = \psi$ をみたすように導出されなければならない．σ の導出は τ の導出過程と**双対** *(dual)* の関係になっているが，技術的観点からはいくつかの注意すべき重要な点が存在する．

図 3.7 に示したアルゴリズムは，以下の制限された形式である $\psi : \mu F \rightarrow F'\mu F$ から σ を導出する．

$$\psi = \lambda v_s.\ \text{case}\ v_s\ \text{of}\ p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n$$

ここで case 式の判定式は変数に簡略化されなければならない．我々のアルゴリズムの重要な点は，入力の際にはパターンマッチによって行なわれるため非明示的であった分離の作業を，μF を $F\mu F$ に分離する関数 out_F を用いることによって明示的にすることである．ここで，$d_1, \ldots, d_n$ を型 μF を out_F によって連続して分離 (unfold) したデータとして定義する．

$$\begin{array}{rcl} d_1 & = & out_F\ v_s \\ d_2 & = & F\ out_F\ d_1 \\ & \vdots & \\ d_m & = & F^{m-1}\ out_F\ d_{m-1} \end{array}$$

ここで F^m は $F^m = F^{m-1} \circ F$ によって帰納的に定義される．今 ψ における $v_s : \mu F$ を $(d_1, \ldots, d_m) : F\mu F \times F^2\mu F \times \cdots \times F^m\mu F$ で置換し，図 3.7 に示すようにアルゴリズム $\mathcal{G}'$ を適用し，元の v_s に対応する case 式のパターン $p_1, \ldots, p_n$ を，$(d_1, \ldots, d_m)$ に対応するパターン $p_1', \ldots, p_n'$ に入れ替える．最後に，d_i に含まれるすべての out_F をラムダ抽象すれば結果として σ が得られる．

ここで図 3.7 のアルゴリズムについてさらに説明を加える．アルゴリズム $\mathcal{G}$ は，ψ の case 式における各パターン p_i に対してアルゴリズム $\mathcal{G}'$ を適用し，結果としてペア (v_i, q_i) を得る．最初の要素である v_i は，パ

$$
\begin{array}{lcl}
\psi & : & \mu F \to F'\mu F \\
\psi & = & \sigma out_F, \quad \text{where } \sigma = \lambda\beta.\mathcal{G}[\![\psi]\!]
\end{array}
$$

where

$$
\begin{array}{l}
\mathcal{G}[\![\lambda v_s.\text{case } v_s \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!] = \\
\quad \lambda v_s.\text{let } d_1 = \beta\ t_0 \\
\qquad\quad d_2 = F\ \beta\ d_1 \\
\qquad\qquad \vdots \\
\qquad\quad d_m = F^{m-1}\ \beta\ d_{m-1} \\
\qquad \text{in} \\
\qquad \text{case } (d_1, \ldots, d_m) \text{ of } p'_1 \to t_1 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad p'_n \to t_n \\
\qquad \text{where } (v_i, q_i) = \mathcal{G}'[\![p_i]\!] \quad (i = 1, \ldots, n) \\
\qquad\quad [q_{i_1}, \ldots, q_{i_{l_i}}] = q_i \\
\qquad\quad m = max\{|q_1|, \ldots, |q_n|\} \\
\qquad\quad p'_i = (q_{i_1}, \ldots, q_{i_{l_i}}, \_, \ldots, \_) \quad \text{(m-tuple)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G}'[\![v]\!] & = & (v,\ []) \\
\mathcal{G}'[\![C_j\ p_1 \cdots p_n]\!] & = & (\_,\ q) \\
 & & \text{where } (v_i, q_i) = \mathcal{G}'[\![p_i]\!] \quad (i = 1, \ldots, n) \\
 & & \qquad q = (\mathrm{j},\ (v_1, \ldots, v_n)) : map\ (\lambda x.(\mathrm{j}, x))\ (zip_n\ q_1 \cdots q_n) \\
\mathcal{G}'[\![(p_1, \ldots, p_n)]\!] & = & (\_,\ q) \\
 & & \text{where } (v_i, q_i) = \mathcal{G}'[\![p_i]\!] \quad (i = 1, \ldots, n) \\
 & & \qquad q = zip_n\ q_1 \cdots q_n \\
 & & /^*\ \text{This } zip_n \text{ is different from the standard in regard to } ^*/ \\
 & & /^*\ \text{supplying wildcard \_ at the tail for shorter lists } ^*/
\end{array}
$$

図 3.7. Deriving σ

ターンに現われる変数である．もし変数が現われなければ，v_i として _ (ワイルドカード) が返される．二番目の要素 q_i は，入れ子になってタグ付けされた項を示す．このリストの各要素は，out_F によって分離されたパターンに対応する．このリストの長さは，すべてのパターン $p_1, \cdots, p_i$ を得るような v_s を展開するために，何回 out_F を適用しなければならないかを示している．

ψ から σ を導出するこのアルゴリズムの正当性の証明は省略する．その代わりに，空でない整数のリストの最大値を求める関数を例として，ここから Hylo を求めてみよう．

$$
\begin{array}{l}
maximum = \lambda xs.\ \text{case } xs \text{ of } Nil \to error \\
\qquad\qquad\qquad\qquad\qquad\quad Cons\ (a, Nil) \to a \\
\qquad\qquad\qquad\qquad\qquad\quad Cons\ (a, as) \to max(a, maximum\ as) \\
\qquad = [\![\phi, id, \psi]\!]_{F', F'} \\
\qquad\quad \text{where } \phi = error \triangledown id \triangledown max \\
\qquad\qquad\quad \psi = \lambda xs.\ \text{case } xs \text{ of } Nil \to (1, ()) \\
\qquad\qquad\qquad\qquad\qquad\quad Cons\ (a, Nil) \to (2, (a)) \\
\qquad\qquad\qquad\qquad\qquad\quad Cons\ (a, as) \to (3, (a, as))
\end{array}
$$

$$F' = !\mathbf{1} + !Int + !Int \times I.$$

We apply Algorithm $\mathcal{G}'$ to the pattern part of the case expression in ψ:

$$\begin{aligned}
\mathcal{G}'[\![Nil]\!] &= (_,\ [(1,())]) \\
\mathcal{G}'[\![Cons\ (a,\ Nil)]\!] &= (_,\ q) \\
&\quad \text{where } (a,\ []) = \mathcal{G}'[\![a]\!] \quad (_,\ [(1,())]) = \mathcal{G}'[\![Nil]\!] \\
&\qquad q = (2,\ (a,\ _)) : map\ (\lambda x.\ (2, x))\ (zip\ []\ [(1,())]) \\
&\qquad\ = [(2,\ (a,\ _)), (2, (_, (1, ())))] \\
\mathcal{G}'[\![Cons\ (a,\ as)]\!] &= (_,\ q) \\
&\quad \text{where } (a,\ []) = \mathcal{G}'[\![a]\!] \quad (as,\ []) = \mathcal{G}'[\![as]\!] \\
&\qquad q = (2,\ (a, as)) : [] \\
&\qquad\ = [(2,\ (a, as))]
\end{aligned}$$

この中では $\mathcal{G}'[\![Cons\ (a, Nil)]\!]$ の結果リストの長さが最長の 2 になったので，この case 文が分岐を行なうには out_F を 2 回適用すればよいことがわかる．そこで d_1, d_2 に xs を分解したバインドするよう let 式を作ると，以下のように σ を導出することができる．

$$\begin{aligned}
\sigma &= \lambda\beta.\ \mathcal{G}[\![\lambda xs.\ \text{case } xs \text{ of } Nil \to (1,()) \\
&\qquad\qquad\qquad\qquad Cons\ (a, Nil) \to (2, (a)) \\
&\qquad\qquad\qquad\qquad Cons\ (a, as) \to (3, (a, as))]\!] \\
&= \lambda\beta.\ \lambda xs.\ \text{let } d_1 = \beta\ xs \\
&\qquad\qquad\qquad\quad d_2 = F'\ \beta\ d_1 \text{ in} \\
&\qquad\qquad\quad \text{case } (d_1, d_2) \text{ of } ((1,()), _) \to (1,()) \\
&\qquad\qquad\qquad\qquad ((2, (a, _)), (2, (_, (1, ())))) \to (2, (a)) \\
&\qquad\qquad\qquad\qquad ((2, (a, as)), _) \to (3, (a, as)).
\end{aligned}$$

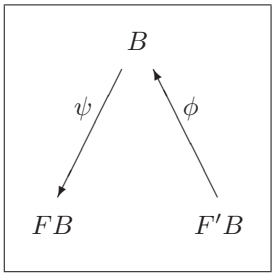
3.6 酸性雨定理の適用

本節では，融合変換を行なうための酸性雨定理をどのように適用するかについて述べる．定理 1 が示すように，この定理を適用するためには，$[\![\_, \_, \tau in_F]\!] \circ [\![out_F, \_, \_]\!]$ または $[\![\_, \_, in_F]\!] \circ [\![\sigma out_F, \_, \_]\!]$ のような特定の形式になっている必要がある．ここで，いくつかの技術的問題に直面する．例えば，プログラムにおいて融合変換可能な個所をどのように見つければよいであろうか．さらに見つけた関数合成が，本当に酸性雨定理によって融合可能であるかを，どのようにして自動的に判定すればよいであろうか．

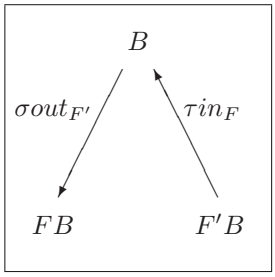
ここで，潜在的に融合可能な合成を以下のように定義することにする．この合成では，中間的なデータ構造が実在していることになる (これは機械によって直接サポートされている例えば *float* や *int* などのデータ型ではない)．

Definition 3 (*潜在的に融合可能な関数合成*) 項が**潜在的に融合可能な関数合成**であるとは，(i) $t_1 \circ t_2$ または $(t_1\ (t_2\ t))$ の形式のどちらかであり，(ii) t_1 のドメインは内関手によって定義されるデータ構造であることである． □

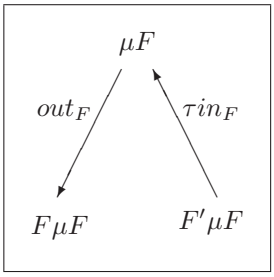
融合変換を適用するには，まずは main 関数から開始する．その後，main 関数の定義本体から潜在的融合可能合成 $(t_1 \circ t_2)$ を探索する．そして t_1, t_2 の両方に対して適切な Hylo を得ることによってそれを酸性雨定



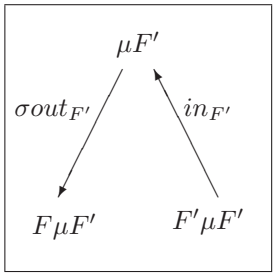
(a) ϕ and ψ in general case



(b) Fusion Impossible



(c) $B = \mu F$



(d) $B = \mu F'$

図 3.8. Relation between two hylomorphisms

理によって融合を試みる．ここでは 2 つの Hylo の合成 $h_1 \circ h_2$ を考えてみよう:

$$\begin{array}{ll} h_1 : B \to C & h_2 : A \to B \\ h_1 = [\![\_, \_, \psi]\!]_{\_,F} & h_2 = [\![\phi, \_, \_]\!]_{F',\_} \end{array}$$

ここでは読み易くするために，本質的でない個所にはワイルドカードを用いることにする．図 3.8 (a) は ψ, ϕ と，それによって導出されるデータ型の関係を示している．定理を適用するために，以下の手続きが行なわれる．括弧で囲まれた項は，同時に検査されるべき双対の条件を示している．

1. B **が** μF $(\mu F')$ **かどうかの検査**．明らかに B が μF や $\mu F'$ のどちらでもない場合は，図 3.8 (b) に示したように，酸性雨定理が適用できる形式を得ることはできない．
2. B **が** μF $(\mu F')$ **であるときは，** ψ (ϕ) **が** out_F $(in_{F'})$ **かどうかの検査**．一般的に，2 つの関数が等しいかどうかを判定することは不可能である．しかし ϕ が in_F であるかどうかは，ϕ や in_F に対してすべての (有限な) μF のデータ型を適用し，その結果を比較することによって，同一性の判定を行なうことができる．
   一度 out_F (in_F) と判定されたなら，それを σout_F (τin_F) の形式に変更する必要はない．従って，2 つの Hylo のうち片方については，3.5 節や 3.5.2 節の過程を省略することができる．
3. **もし他方の** ψ (ϕ) **が** out_F $(in_{F'})$ **であるとき，もう片方の** *Hylo* **における** *phi* (ψ) **が** τin_F $(\sigma out_{F'})$ **に変換できるかの検査**．図 3.6 や図 3.7 のアルゴリズムを用いて，図 3.8 (c) や (d) に示した状態へと到達する．

このテストを通過したものだけが，酸性雨定理を適用することが可能な関数合成である．

さきほどの ssf の例に戻ると，導出された Hylo に対し連続的に融合することが可能であり，関数適用を関数合成を用いた形式に書き換えると

$$[\![\phi_1, \eta_1, out_F]\!]_{F,F} \circ [\![in_F, \eta_2, \psi_2]\!]_{F,F} \Rightarrow [\![\phi_1, \eta_1 \circ \eta_2, \psi_2]\!]_{F,F}$$

これより元の関数 ssf は以下のようになる．

$$ssf = sum \circ map\ square \circ upto$$

$$
\begin{aligned}
&= [\![0 \triangledown (+), id, out_F]\!]_{F,F} \circ [\![in_F, id + \lambda(v, v').\ (square\ v, v'), out_F]\!]_{F,F} \circ \\
&\qquad [\![in_F, id, \psi]\!]_{F,F} \\
&= [\![0 \triangledown (+), id + \lambda(v, v').\ (square\ v, v'), out_F]\!]_{F,F} \circ [\![in_F, id, \psi]\!]_{F,F} \\
&= [\![0 \triangledown (+), id + \lambda(v, v').\ (square\ v, v'), \psi]\!]_{F,F} \\
&= \lambda(m, n).\ \text{case}\ (m > n)\ \text{of}\ True \to 0 \\
&\qquad\qquad\qquad\qquad False \to square\ m + ssf\ (m + 1, n)
\end{aligned}
$$

3.7 Hylo の内部における融合

これまでに，2 つの Hylo の合成をどのように融合するかをみてきた．しかし融合変換後に生成された Hylo の内部に対する融合変換については触れてこなかった．直観的には，この融合変換は非常に単純に思えるかもしれない．Hylo の各要素に対して融合変換を行なえばよさそうであるからである．しかし，この単純な戦略では，3 つの要素の間にある中間データ構造を効率的には除去できない．簡単な例として，以下のような Hylo の式 ($concat \circ map\ f$ から導出される) を考えてみよう:

$$[\![\lambda().() \ \triangledown\ \lambda(u, v).u +\!\!+ v,\ id \triangledown \lambda(a, b).(f\ a, b),\ out]\!]$$

このままでは，各要素はこれ以上融合できない．しかし，η 部を ψ 部へと移動すると，

$$[\![\lambda().() \ \triangledown\ \lambda(u, v).f\ u +\!\!+ v,\ id,\ out]\!]$$

f によって生成されるデータ構造が $(+\!\!+\ v)$ によって消費されるので除去可能になる．

そこで Hylo$[\![\phi, \eta, \psi]\!]$ に対しては，以下のように融合変換を行なう:

(1) $[\![\phi \circ \eta, id, \psi]\!]$ の各要素に対して処理を行なう
(2) ステップ (1) によって得られた Hylo を再構成し，$[\![\phi', \eta', \psi']\!]$ が得られる
(3) $[\![\phi', id, \eta' \circ \psi']\!]$ の各要素に対して処理を行なう
(4) ステップ (3) で得られた Hylo を再構成する

3.8 Hylo のインライン展開

Hylo の展開は，3.2 節における hylo 導出の逆の過程に相当する．すなわち，プログラム中に fusion されないで残ったすべての Hylo を以下のようにして再帰的な関数定義に戻すことによって，融合変換のために便宜的に導入した hylo 構造をプログラムから消すことができる．

第 4 章

融合変換の実装

本章では，これまで融合変換をプロトタイプ上で実験してきた経験 [OHIT97] を活かし，これを実際の広く用いられている関数型言語のコンパイラに対して，どのように実装したかについて説明する．対象とした言語処理系は，Haskell 言語の処理系としてはもっとも一般的に用いられている Glasgow Haskell Compiler (GHC) である．我々の最適化技術を実在するコンパイラに組み入れることによって，実用上の問題点を調べることが可能になり，さらに実用的な規模の例題に対して融合変換の有効性を検証することが可能になる．まず GHC について説明することから始め，次に内部で用いられる中間言語について触れる．この中間言語上で，融合変換の処理を行なうことになる．その後，融合変換を GHC に加える方法について解説する．

次に，実際に実装し出力されるコードを調べることによって得られた実装のある局面に関する詳細について触れる．融合変換を多くの式に対して行なうようにするには，いくつかの注意すべき点についても述べる．

4.1 Glasgow Haskell Compiler

Glasgow Haskell Compiler (以後 GHC と略) は，Glasgow 大学の GRASP, AQUA プロジェクトによって生みだされた実用的な規模のコンパイラである．このコンパイラは，それ自身 Haskell で記述されており，そのモジュラー性やクラス構造などから新しい最適化技術を追加実装しやすくなっている特徴がある．

GHC は，その様々なコンパイルステージを，正当性を保持するプログラム変換として表現している特徴がある．このプログラム変換を用いたコンパイルというパラダイムは，関数型言語のコンパイラの世界においては共通の認識となっている [FlM91]．このような変換は，ある文法を同様の文法へ変換することも可能であるし，ある文法上において最適化変換を行ないプログラムの効率を高めることも可能である．

Glasgow 大学で開発された Haskell のコンパイラである．現在の最新版は 5.02 であるが，5.0 系列はまだできて間もないため，OS 毎に用意されているコンパイル済バイナリ (以降パッケージと呼ぶ) が少ないことがある．そのような場合は，1 つ前の版である 4.08.2 を利用するとよい．また 5.0 系列 において新たに追加された機能については，4.1.4 節で解説する．

4.1.1 GHC の構成

図 4.1 は，コンパイラの主な構成要素を表わしている．パーザー部とコンパイラ部を別にすると，主な構成要素はすべて Haskell そのもので記述されている．これらの構成要素は以下の関数から組み合わされている．

1. Yacc パーザーが Haskell プログラムを読み込み，抽象構文木をコンパイラに渡す
2. Renamer が，名前の有効範囲問題やモジュール境界を越えた情報伝達などの変数名に関する問題を解決する
3. 型推論部が，Hindley-Milner の型推論アルゴリズムを拡張したもの [WB89] を用いて，型情報をプログラムに注釈付けする

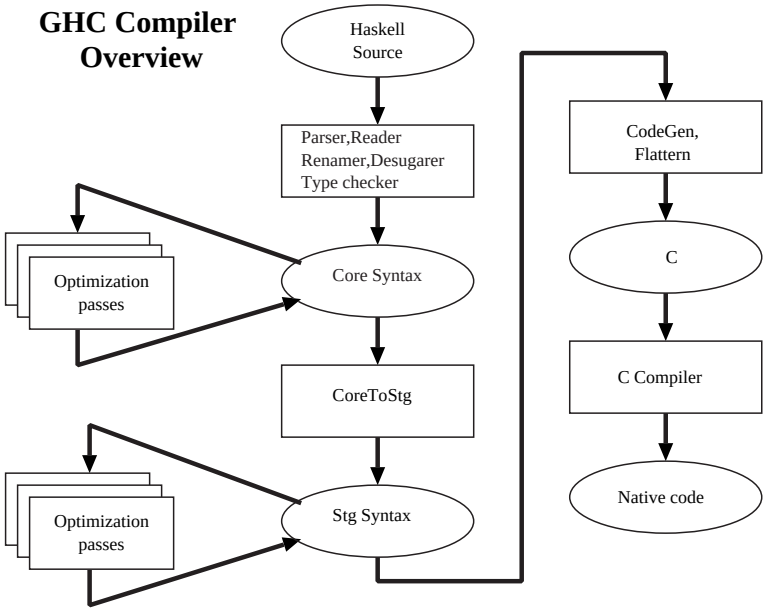


図 4.1. GHC の構造

4. Desugaring が，元の Haskell の構文全体から，Core 言語と呼ばれるより直接的な関数型言語に変換する
5. 各種の異なる Core-to-Core の最適化処理が行なわれる．この構成部分に，我々の融合変換処理を追加するもっとも自然な個所である．
6. Core 言語を，さらに簡略化された言語である STG (Shared Term Glaph) 言語に変換する
7. この簡略化された言語の上で，さらにいくつかの変換を行なう
8. 最終的に，コード生成器が STG 言語を C 言語に変換する．この C 言語で記述されたプログラムは，外部で用意されている任意の C コンパイラ (通常は gcc) によりコンパイルされ，実行ファイルが生成される．

この手順は，GHC によるコンパイル時に最適化オプション `-O` を付けた場合の流れであり，通常はこのように処理が行なわれることになる．また GHC は，内部にネイティヴコード生成器 (Native Code Generator, NCG) を備えていて，最適化オプションを付けずにコンパイルした場合，この NCG が STG 言語から直接アセンブル言語 (x86,sparc) を生成することも可能である．最適化オプションを付けた場合の方が，外部のコンパイラをを呼び出すこともありコンパイルに要する時間はかかることが多いが，gcc における強力な最適化を利用できるため，実行モジュールの効率は良いことが多い．

4.1.2 GHC における Core 言語

GHC コンパイラの中で，我々の融合変換を最適化処理の一部として実装するのに適している箇所は，図 4.1 の左中部に位置する Core 言語上での最適化変換フェーズである．GHC における Core 言語は，二階のラムダ計算を拡張させたもので，その文法は図 4.2 のようになっている．

Core 言語は最適化変換に用いられるように設計されたものである．このため，Core 言語は必要最小限から構成されるように，意図的に設計されている．これにより，Haskell のような大きな言語の文法の複雑さからおこる混乱を意識することなく，最適化処理を行なうことが可能になっている．図 4.2 の式に含まれる型適用式 (Type application) や型ラムダ式 (Type abstraction) が示すように，第二階のラムダ計算を基本としていて，型を抽象化した多様型 (polymorphic) の式を用いることが可能である．また `let` 式や `case` 式，明示的な構

成子やプリミティブの適用が追加されたことにより，近代的な関数型言語を効率よく操作することができる．

- Core 言語において，`let` 式はメモリ割り当てを表わす

```
let
    h = <exp>
in
    ...
```

  このコードは `<exp>` を計算する実体がヒープ上に割り当てられたことを意味する．
- `case` 式によって，評価が行なわれる

```
case f x of
    True  -> <exp1>
    False -> <exp2>
```

  これは `f x` が評価され，その結果に応じて `<exp1>` または`<exp2>` が評価される．
- GHC の内部では，コード生成器に至るまで型情報を維持しつづける．型付き Core 言語において任意の変換を行なうために，言語は型を扱える第二階のラムダ計算に基づいて設計されている．このため，任意のオブジェクトに対し，その型をいつでも参照することが可能である．融合変換を行なう際にも，この型情報を用いている．
- 関数適用の引数は，アトム (変数やリテラル) である．これにより，最適化システムが，変換のルールを最小限に保つことが可能になる．例えば，`f (g 2)` のような Haskell の式は，Core 言語上では

```
let
    t = g 2
in
    f t
```

  のように表わされる．

この Core 言語は，前章までにアルゴリズムの提示する際に用いていた言語 (第 2.2 章の図 3.2) と比べて，以下に示すような差異はあるものの，大きな違いではなくこれまで示したアルゴリズムを GHC 上で実装することが可能となっている．

- 関数引数が変数やリテラルなどのアトムに制限される．
  アトムは式の一種なので前章までのアルゴリズムを適用するのに文法上は問題がないが，この制限により関数引数に対する let 束縛が増えることになり，適宜その処理をアルゴリズムに追加する必要がある．
- 組式 (tuple) には，組型の構成子を用いて構成子式として対応させる．
- GHC Core 言語には型適用式や型抽象式などが新たに現われるが，これらを用いた多様性を維持するように注意すれば，Hylo のアルゴリズムに影響を与えることはない．
- GHC 内で Hylo 式を記述するための手法については，後の 4.2.1 節で詳しく解説する．

4.1.3 Haskell から Core 言語への変換 (desugaring)

desugaring の過程によって，Haskell のような大きな言語から文法上の糖衣構文 (syntax sugar) が取り除かれ，Core 言語のような簡潔で限定された言語へ変換される．GHC 内部におけるこの実装は，[Jon87] で行なわれている方式に基づいている．とくに desugaring では以下の各変換が行なわれることになる．

Program	*Prog*	::=	$Binding_1$; ... ; $Binding_n$	$n \geq 1$
Bindings	*Binding*	::=	`nonrec` *Bind*	
		|	`rec` *Binds*	
	Binds	::=	$Bind_1$; ... ; $Bind_n$	$n \geq 1$
	Bind	::=	*v* `=` *Expr*	
Expressions	*Expr*	::=	*Expr Atom*	Application
		|	*Expr ty*	Type application
		|	`\` $v_1 \ldots v_n$ `->` *Expr*	Lambda abstraction, $n \geq 0$
		|	`/\` *ty* `->` *Expr*	Type abstraction
		|	`case` *Expr* `of` *Alts*	Case expression
		|	`let` *Binding* `in` *Expr*	Local definition
		|	*Binding* `in` *Expr*	Local definition
		|	$Con\ Atom_1 \ldots Atom_n$	Constructor, $n \geq 0$
		|	$prim\ Atom_1 \ldots Atom_n$	Primitive, $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	::=	*v*	Variable
		|	*Literal*	Unboxed object
Literal values	*Literal*	::=	$integer \mid float \mid \ldots$	
Alternatives	*Alts*	::=	$Calt_1$; ... ; $Calt_n$; $Default$	$n \geq 0$
		|	$Lalt_1$; ... ; $Lalt_n$; $Default$	$n \geq 0$
Constr. alt	*Calt*	::=	$Con\ v_1 \ldots v_n$ `->` *Expr*	$n \geq 0$
Literal alt	*Lalt*	::=	*Literal* `->` *Expr*	
Default alt	$Default$	::=	*v* `->` *Expr*	
		|	ϵ	

図 4.2. GHC Core Syntax

- パターンマッチングは，単一のレベルの `case` 文に置き換えられる．
- Haskell における `let` 文や `where` 節は，それと等価である単一レベルの `let` 文に置き換えられる．
- 構成子やプリミティブの引数は，必要なら余計にラムダ変数を追加することによって満たされる．
- 関数や構成子，プリミティブの引数はアトムにするために，アトムでない引数に対しては新たな `let` 束縛が追加される．
- 型推論システムによって付けられた型情報を用いて，型ラムダ式や型適用式が加えられる．
- リストの包括表記 (list comprehension) は，再帰的なリストの生成関数と消費関数を用いて表わされる．この変換には，[Wad87] などの伝統的なアルゴリズムが用いられている．

Core 言語の例として，以下の Haskell の関数を考える:

```
reverse xs = rev xs []
  where
    rev (x:xs) ys = rev xs (x:ys)
    rev []     ys = ys
```

desugaring の後は，この関数は以下のようになる．

```
reverse :: \/ a . [a] -> [a]
reverse = /\ ty ->
            \ t ->
                let
                  rev = \ t ys ->
                          case t of
                            x : xs -> let
                                        t1 = (:) ty x ys
                                      in
                                        rev xs t1
                            []     -> ys
                in
                  rev t []
```

ここで，`\/ a.` は任意の型 `a` に対して，という意味になる．

4.1.4 GHCi の新機能

GHC は 5.0 系列から，従来のコンパイラに加え，インタプリタ (GHCi) としても動作するようになっている．起動するには，シェルスクリプトである `ghci` コマンドを実行すればよく，これは通常の `ghc` コマンドに`--interactive` オプションを付けても同じ動作をする．この機能追加により，これまでの 4 系列に比べ約 2 ～ 3 K バイト実行時に多くヒープを消費するようになったが，これは後の実験で用いている NoFib ベンチマークのようにサイズの大きいベンチマークで消費されるヒープ使用量においては，わずかな増加にとどまっているといえよう．

またモジュール間の依存関係を自動的に解決する機能も追加された．この機能を用いると，従来は Haskell プログラム間の依存関係を解決するために Makefile を用いていたところが，ルートモジュール (通常は Main) と `--make` オプションを指定するだけで，再コンパイルが必要なものを自動的に検出できるようになった．これは複数のモジュールに分割して書かれたプログラムをコンパイルする際には，大きな利点となるであろう．

4.2 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 [OHT98]．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 の処理系のひとつで，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 GHC[GHC] 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–5.02 patchlevel 1 である．

図 4.1 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 4.3 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 [Iwa98] 参照)．

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

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

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

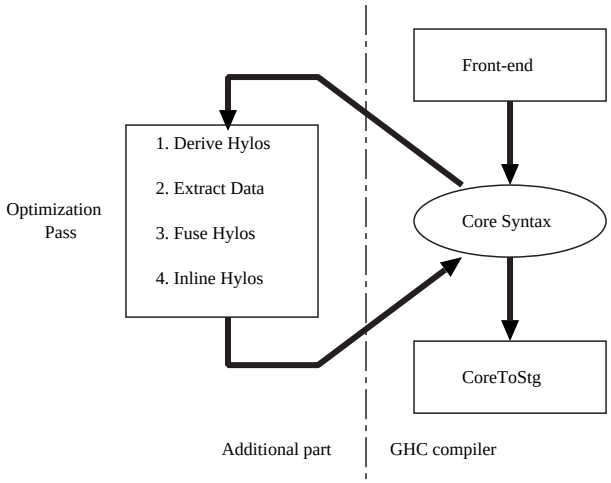


図 4.3. 追加した変換部

$$x : xs \rightarrow p\ x \mid\mid f\ xs$$

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．

4.2.1 Core 言語における Hylo の表現

GHC の Core 言語では，式は以下のようなデータ構造 `Expr` を用いて表わされる．

```
data Expr b
    = Var  Id
    | Lit  Literal
    | App  (Expr b) (Arg b)
    | Lam  b (Expr b)
    | Let  (Bind b) (Expr b)
    | Case (Expr b) b [Alt b]
    | Note Note (Expr b)
    | Type Type
```

ここで融合変換を Core 言語上で行なうためには，Hylo を Core 言語で表わすための枠組を用意する必要がある．その手法としては，一番自然だと思われるのは，上記の `Expr` データ型に対して，新たに `Hylo` などの構成子を追加するものである．

```
data Expr b
    = ..
    | Hylo (..some arguments..)
```

しかしこのアプローチでは，以下のような短所があることがわかった．

- Core 言語は，コンパイラの内部の多くの場所から参照されている．これを変更することは，コンパイラ自身のコンパイルの際に多くのモジュールを再コンパイルする必要があることを示している．
- 本来 Hylo 式は，融合変換が行なわれる際に一時的に必要になるもので，Core 言語上での各種のプログラム変換の際に必要となることはない．そのようなものを Expr データ型の一部として独立して用意するのは大仰である．

そこで本実装では，Expr データ型に含まれる Note 構成子に注目し，これを活用することで Hylo を実現することにした．Note 構成子は，以下のように示すようにある式に対する注釈を付けるものとして Core 言語に用意されている．その利用目的としては，C 言語におけるキャストのような型変換のための情報であったり，その式をインライン展開するか否かの情報であったり，プロファイリングのための情報保持であったりする．

```
data Expr b
    = ..
    | Note  Note (Expr b)
    | ..

data Note
    = SCC CostCentre

    | Coerce
          Type            -- The to-type:   type of whole coerce expression
          Type            -- The from-type: type of enclosed expression

    | InlineCall          -- Instructs simplifier to inline
                          -- the enclosed call

    | InlineMe            -- Instructs simplifer to treat the enclosed expression
                          -- as very small, and inline it at its call sites
```

Hylo 式を Core 言語上であらわすために，この Note の情報を用いるよう，以下のように Note 型の構成子 HyloDef，HyloFnct，HyloNote を追加した．

```
data Note
    = ..
    | HyloDef
          (TyVar,Type)      -- function's returned type
          CoreBndr          -- recursive function's name
          HyloKind          -- kind of hylo
    | HyloFnct
          Fnct              -- recursive flag
    | HyloNote

data Fnct     = FnCon | FnId
data HyloKind = Cata | Ana | Hylo
```

上記の Core 言語を用いて Hylo 形式を表現すると以下のようになる．

```
Note (HyloDef (TyVar,Type) b HyloKind) $
Lam b $
    Note HyloNote $
    Case (Expr b) b
        [
          (DataAlt DataCon, [b],
           Rec [(b, Expr b)] $  -- binds
             Let (b, Expr b) $  -- phi_rhs
               Expr b           -- phi_body
          ),
          ⋮,
          (DataAlt DataCon, [b],
           Rec [(b, Expr b)] $
             Let (b, Expr b) $
               Expr b
          ),
        ]
  :: Expr b
```

例えば，関数 *map* に対する Core 言語上での Hylo 表現は以下のようになる．

```
Note (HyloDef (tv, [Int]) map_f Cata) $
 Lam ds $
    Note HyloNote $
     Case ds wild [
        (DataAlt (:), [y, ys],
            Rec [(y',  Note (HyloFnct FnCon) y),
                 (ys', Note (HyloFnct FnId) ys)] $
                Let (y'', f y') $
                    (:) y'' ys'),
        (DataAlt [], [],
                    []),
     ]
```

4.2.2 hylo fusion のタイミング

GHC は，内部で数多くの最適化処理を行なうことによって，効率のよいコードを生成することが知られている．その最適化処理は Core 言語から Core 言語へのプログラム変換として実現されているため [JS98]，状態遷移を用いている通常のコンパイラと比べて，ユーザが任意のプログラム変換をコンパイラ内に追加実装しやすくなっている．

この融合変換の処理を最適化の流れの中のどこに加えるかが，大きな問題となってくる．この問題への一つの解決策として，[Gil96] によると shortcut 融合変換 (6.2.1) を行なうタイミングとしては，完全遅延変換の後，正格性解析の前がよいと示している．これは以下のような理由による．

- 完全遅延変換を先に行なう
  例として以下のプログラムを考える．

```
foo f = map f (map g xs)
```

  ここで関数 f の内部に現われる g と xs は自由変数である．この例に対し，完全遅延変換を行なうと

```
v = map g xs
foo f = map f v
```

  のようになる．もしこれが，完全遅延変換の前に融合変換を行なったとすると

```
foo f =
  let
      h [] = n
      h (a:as) = f (g a) : as
  in
      h xs
```

  のように 2 つの map 関数は融合されることになる．この状態ではもう中間リスト構造は生成されないため，ここから中間リストを一旦保存するような式はもう導けない．ここで完全遅延変換と融合変換のどちらを優先して行なうべきであろうか．この問題を調べるために，さきほどの例題で自由変数 xs, g が以下の値をもつものと仮定してみる．

```
xs = [1..n]
g x = if x > 2 then g (x-1) + g (x-2) else 1
```

  ここで完全遅延変換を先に行なうと，g 1 : g 2 : g 3 : ... というリストは一回だけ生成される．一方融合変換を先に行なうと，関数 g を適用した結果は，可変である関数 f の引数として与えられるため，式が評価されるたびに関数 g の計算を毎回実行することになる．したがってこの例から判断すると，完全遅延変換を先に行なうほうが望ましいことになる．
  この振舞いは極めて直感的である．融合変換は生成側と消費側を一つにまとめ上げる効果を持つが，その生成側や消費側がその評価結果を共有するが可能であるなら，それらを間で受け渡しされる中間データ構造を除去するよりも，より大きな節約に繋がることがあるからである．このことからわかるように，一般的に中間データ構造の生成よりもリスト等の各要素の評価にコストがかかる場合の方が多いといえよう．
- 正格性解析を後から行なう
  融合変換を行なった後に正格性解析を行なえば，関数 foldr が unfold されることによって，より正格性解析に適した関数が生成されることになる．具体例として，

```
foo n = sum [1..n]
```

  を考えてみよう．このプログラムに融合変換やいくつかの最適化を行なうことによって，最終的に以下の形式へと変換される．

```
foo n =
  let
      h x a = if x < n then h (x+1) (a+x)
                       else a
```

```
in
    h 1 0
```

この状態では，正格性解析によって h の両引数が調べられ，非常に効率のよいコードが生成されることになる．これを逆順で行なってしまうと，この重要な最適化の機会を失ってしまうことになるのである．

一方 Hylo 変換を行なうタイミングとしては，第一の条件として shortcut 融合変換を行なった後に行なうことが望ましい．これは，組み込み関数を処理するためにはその関数の定義をどこかで知る必要があるのだが，shortcut 融合変換を行なう際に行なわれる RULES プラグマの処理によって，ライブラリ内にある関数定義が変換対象のプログラム内に取り込まれるからである．これを逆順にしてしまうと Hylo 変換を行なう時点では組み込み関数の定義がわからないため，これらの関数に対する融合処理は行なえない．

また Hylo 変換はなるべく Core 言語として簡略化されたものを受け付けるほうが，Hylo の導出や融合の処理が行ないやすいことが経験的にわかっている．これは Hylo 変換プログラムの実装が，ある程度 Core 言語として自然な形のものを想定しているためで，冗長な let 束縛や不自然な形の case 式などが現われると，そこで融合変換を諦めてしまうことも多く存在してしまう．したがって最適化変換のなるべく後の段階に，Hylo 変換の処理を追加することにした．

GHC コンパイラの中で最適化の処理の流れを示している箇所をコンパイラのソースから一部抜粋したものを図 4.4, 4.5 に示す．元となるファイルは，ghc/complier/main/DriverState.hs で，この関数 buildCoreToDo が返す CoreToDo 型構成子のリストによって，最適化オプション -O をつけたときの最適化変換の流れが決定することになる．

具体的には，この中の ★★★ (B) ★★★ の部分に Hylo 変換を行なわせるよう以下の 8 行を追加した．

```
if hyloFusion then
  CoreDoSimplify (isAmongSimpl [ MaxSimplifierIterations max_iter ])
else
  CoreDoNothing,
if hyloFusion then
  CoreDoHyloFusion
else
  CoreDoNothing,
```

これは -O2 オプションを行なわない限りは最適化変換列のほぼ最終段階に位置するものである．なお ★★★ (A) ★★★ は，shortcut 融合変換を行なう前に相当し，後の節で試験的にこの箇所で Hylo 変換を行なった場合と比べ，効率がどのように変化するかを調べることにする．

4.2.3 現在の実装の問題点

[m..n] から Hylo が導出不可

例えば，[m..n] のような列挙により生成されたリストは，関数プログラミングの世界では非常に多用されているにもかかわらず，現在の我々の実装では融合変換を行なうことができない．そこで，ghc-5.02.1 でのライブラリの実装例を挙げてみることにする．Haskell における [m..n] という記法は簡略化された記法で，コンパイラ内部では enumFromTo m n のような関数適用として表現されている．この関数 enumFromTo は，ghc/lib/std/PrelEnum.lhs の中に定義されている多様型関数で，Int 型に関する定義部分を図 4.6 に示す．l.3 にある enumFromTo は一行上にある INLINE プラグマの指定によって，引数が unbox 化された後右辺の式にインライン展開される．その後，関数 eftInt は，以下のように shortcut 融合変換の有無によって異なる処理がなされる．

```
buildCoreToDo :: IO [CoreToDo]
buildCoreToDo = do
   opt_level  <- readIORef v_OptLevel
   max_iter   <- readIORef v_MaxSimplifierIterations
   usageSP    <- readIORef v_UsageSPInf
   strictness <- readIORef v_Strictness
   cpr        <- readIORef v_CPR
   cse        <- readIORef v_CSE

   if opt_level == 0 then return
      [
        CoreDoSimplify (isAmongSimpl [
            MaxSimplifierIterations max_iter
        ])
      ]

    else {- opt_level >= 1 -} return [

        -- initial simplify: mk specialiser happy: minimum effort please
        CoreDoSimplify (isAmongSimpl [
            SimplInlinePhase 0,
                        -- Don't inline anything till full laziness has bitten
                        -- In particular, inlining wrappers inhibits floating
                        -- e.g. ...(case f x of ...)...
                        --  ==> ...(case (case x of I# x# -> fw x#) of ...)...
                        --  ==> ...(case x of I# x# -> case fw x# of ...)...
                        -- and now the redex (f x) isn't floatable any more
            DontApplyRules,
                        -- Similarly, don't apply any rules until after full
                        -- laziness.  Notably, list fusion can prevent floating.
            NoCaseOfCase,
                        -- Don't do case-of-case transformations.
                        -- This makes full laziness work better
            MaxSimplifierIterations max_iter
        ]),

        -- Specialisation is best done before full laziness
        -- so that overloaded functions have all their dictionary lambdas manifest
        CoreDoSpecialising,

        CoreDoFloatOutwards False{-not full-},
        CoreDoFloatInwards,

        ★★★ (A) ★★★

        CoreDoSimplify (isAmongSimpl [
           SimplInlinePhase 1,
                -- Want to run with inline phase 1 after the specialiser to give
                -- maximum chance for fusion to work before we inline build/augment
                -- in phase 2.  This made a difference in 'ansi' where an
                -- overloaded function wasn't inlined till too late.
           MaxSimplifierIterations max_iter
        ]),

        -- infer usage information here in case we need it later.
        -- (add more of these where you need them --KSW 1999-04)
        if usageSP then CoreDoUSPInf else CoreDoNothing,

        CoreDoSimplify (isAmongSimpl [
                -- Need inline-phase2 here so that build/augment get
                -- inlined.  I found that spectral/hartel/genfft lost some useful
                -- strictness in the function sumcode' if augment is not inlined
                -- before strictness analysis runs
           SimplInlinePhase 2,
           MaxSimplifierIterations max_iter
        ]),
```

図 4.4. コンパイラ内部における最適化の流れ (ソースより抜粋)

```
        CoreDoSimplify (isAmongSimpl [
           MaxSimplifierIterations 3
                -- No -finline-phase: allow all Ids to be inlined now
                -- This gets foldr inlined before strictness analysis
                --
                -- At least 3 iterations because otherwise we land up with
                -- huge dead expressions because of an infelicity in the
                -- simpifier.
                --        let k = BIG in foldr k z xs
                -- ==>  let k = BIG in letrec go = \xs -> ...(k x).... in go xs
                -- ==>  let k = BIG in letrec go = \xs -> ...(BIG x).... in go xs
                -- Don't stop now!
        ]),

        if cpr        then CoreDoCPResult   else CoreDoNothing,
        if strictness then CoreDoStrictness else CoreDoNothing,
        CoreDoWorkerWrapper,
        CoreDoGlomBinds,

        CoreDoSimplify (isAmongSimpl [
           MaxSimplifierIterations max_iter
                -- No -finline-phase: allow all Ids to be inlined now
        ]),

        CoreDoFloatOutwards False{-not full-},
                -- nofib/spectral/hartel/wang doubles in speed if you
                -- do full laziness late in the day.  It only happens
                -- after fusion and other stuff, so the early pass doesn't
                -- catch it.  For the record, the redex is
                --          f_el22 (f_el21 r_midblock)


-- Leave out lambda lifting for now
--          "-fsimplify",        -- Tidy up results of full laziness
--            "[",
--                 "-fmax-simplifier-iterations2",
--            "]",
--          "-ffloat-outwards-full",

        -- We want CSE to follow the final full-laziness pass, because it may
        -- succeed in commoning up things floated out by full laziness.
        -- CSE used to rely on the no-shadowing invariant, but it doesn't any more

        if cse then CoreCSE else CoreDoNothing,

        CoreDoFloatInwards,

        ★★★ (B) ★★★

-- Case-liberation for -O2.  This should be after
-- strictness analysis and the simplification which follows it.

        if opt_level >= 2 then
           CoreLiberateCase
        else
           CoreDoNothing,
        if opt_level >= 2 then
           CoreDoSpecConstr
        else
           CoreDoNothing,

        -- Final clean-up simplification:
        CoreDoSimplify (isAmongSimpl [
          MaxSimplifierIterations max_iter
                -- No -finline-phase: allow all Ids to be inlined now
        ])
    ]
```

図 4.5. コンパイラ内部における最適化の流れ (ソースより抜粋) (cont.)

```
1  instance  Enum Int  where
2      {-# INLINE enumFromTo #-}
3      enumFromTo (I# x) (I# y) = eftInt x y
4
5  {-# RULES
6  "eftInt"        forall x y.     eftInt x y
7         = build (\ c n -> eftIntFB c n x y)
8
9  "eftIntList"    eftIntFB  (:) [] = eftIntList
10  #-}
11
12 {-# INLINE eftIntFB #-}
13 eftIntFB c n x y | x ># y    = n
14                  | otherwise = go x
15                  where
16                    go x = I# x `c` if x ==# y then n else go (x +# 1#)
17                         -- Watch out for y=maxBound; hence ==, not >
18         -- Be very careful not to have more than one "c"
19         -- so that when eftInfFB is inlined we can inline
20         -- whatver is bound to "c"
21
22 eftIntList x y | x ># y    = []
23                | otherwise = go x
24                where
25                  go x = I# x : if x ==# y then [] else go (x +# 1#)
```

図 4.6. enumFromTo の定義 (PrelEnum.hs より一部抜粋)

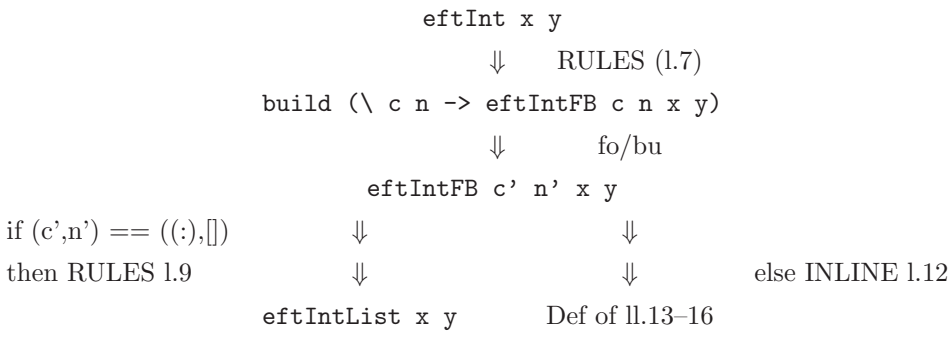


図 4.7. shortcut 融合変換が行なう場合の `[m..n]` の処理

ちなみにコンパイルの際に最適化オプション `-O` を指定しないと，`enumFromTo` などの関数は `Enum` 辞書型の要素として含まれており，その中から必要な処理を抽出する形で実行されている．したがって，shortcut 融合変換は行なわれない．ll.13–16 の定義は，ll.22–26 の定義から最終結果となるリストを構成するための構成子を抽象化したものに相当する．

ここで ll.22–26 の関数定義に注目すると，この中の関数 `go` は Core 言語上で以下のような式となっており，これは $\lambda xs.\text{case } xs \text{ of } \ldots$ の形をしておらず，case 式が let 式の右辺に現われていることがわかる．

```
go = \ x ->
        let a = I# x in
        let b = case x of
                  y -> []
                  _ -> go (x +# 1#) in
        (:) a b
```

この形を図 3.3 のアルゴリズムに照らし合わせてみると，$\mathcal{A}[\![\lambda v_{s_1} \cdots \lambda v_{s_m}.\text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!]\, f$ のパターンにマッチしないため Hylo の導出は行なわれず，したがってこの関数に対する融合変換はできなくなる．

```
eftIntList x y | x >=# y    = []
               | otherwise = go x
               where
--                  go x = I# x : if x ==# y then [] else go (x +# 1#)
                  go x = if x ==# y then I# x : [] else I# x : go (x +# 1#)
```

ここで関数 `go` の定義において，`(I# x :)` の部分を then 部と else 部に分配するよう書き換えた上の定義を用いると，我々のアルゴリズムでも `[m..n]` から Hylo の導出と，それに伴なう融合変換が行なえることを確認した．この例は，図 3.3 のアルゴリズムでは Hylo を導出できないような再帰関数の例がある，ということを示している．これは図のアルゴリズムを開発した当初は let 式を含まない言語を対象としていた事に起因するもので，let 式を含む複雑な式にも対応できるように，さらに変換アルゴリズムを見直すべきであろう．

この他にも，理論的には変換が可能な場合であっても，前章で示したアルゴリズムでは対応が不十分でこの実装では融合変換が行なえない可能性があることは注意しなければならない．したがって 4.2.2 節で述べたように，なるべく Core 言語が簡略化された状態を変換の入力とすることによって，等価なプログラムでも形が複雑であるために変換できないという事態の多くを避けることができるようになる．

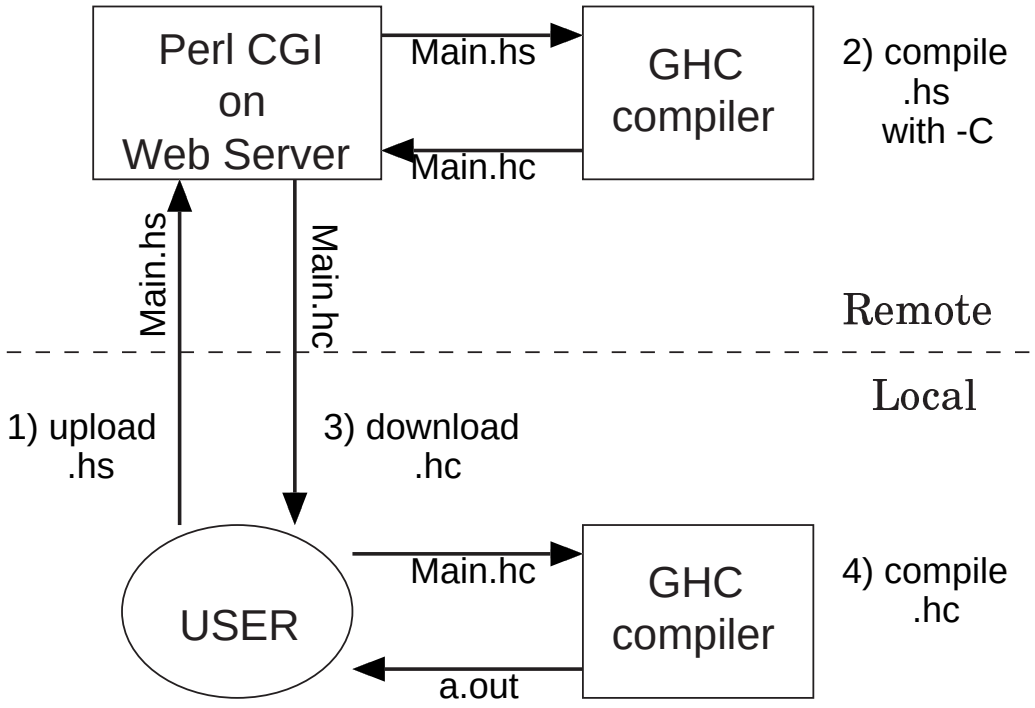


図 4.8. WWW インターフェースの流れ図

4.3 GHC コンパイラに対する WWW インターフェース

GHC コンパイラは，その機能や内部で行なわれている最適化も豊富なため，われわれが埋め込んだ融合変換の効果を外部から直感的に理解するのは困難である．またコンパイラに対して埋め込むことによって，幅広い場面での利用されることを想定しており，現時点での我々の実装を多くのユーザに試してもらうことが重要と考えている．そこで GHC コンパイラに対し，ネットワークを経由した WWW におけるインターフェースを作成することによって，任意のユーザが我々のシステムを試用できるようにした．その構成を図 4.8 に示す．使い方は以下のようになっている．

1. まずコンパイルする対象となる Haskell のプログラムを指定する．これはユーザが自分で作った任意のプログラムを，我々の改造したコンパイラで自由にコンパイルできるように，ファイル名をテキストフィールドで指定可能なようになっている．また，融合変換が行なわれる様子を手軽に確認したいときには，あらかじめこちらで用意した 2 つのサンプルプログラム (`List.hs`,`Tree.hs`) を指定し，そのコンパイル結果を確認することも可能である．(図 4.9)
2. 'Compile' ボタンを押すと，我々の Web サーバ上にインストールされている GHC コンパイラが指定されたオプションを用いて起動され，対象となるプログラムがコンパイルされる．オプションの中には `-C` が含まれているため，コンパイラは入力の Haskell プログラムから対応する C プログラム (.hc ファイル) を出力した段階で終了する．ユーザは，ここで生成された `Main.hc` プログラムをダウンロードし，これをローカルな環境でコンパイルすることによって，融合変換の処理を行なった実行ファイルを生成することができる．(図 4.10) このようにユーザが .hc ファイルをダウンロードする形式にすることによって，各ユーザ側でもコンパイルの作業が必要になるという欠点はあるが，.hc ファイルはプラットフォーム独立なため，サーバ側で用いている OS (FreeBSD) 以外の環境にも適合した実行ファイルが生成可能であるという利点がある．
3. また，オプションの中には`-ddump-simpl` が含まれているため，融合変換などの最適化が行なわれた後

の Core プログラムが画面に表示される．図 4.11 の例では，*go¡+¿go¡+¿upto'* というような斜体で示される関数が融合変換によって新たに生成された関数を表わし，これは元の式が `go . go . upto'` という関数合成式であったことを示すものである．

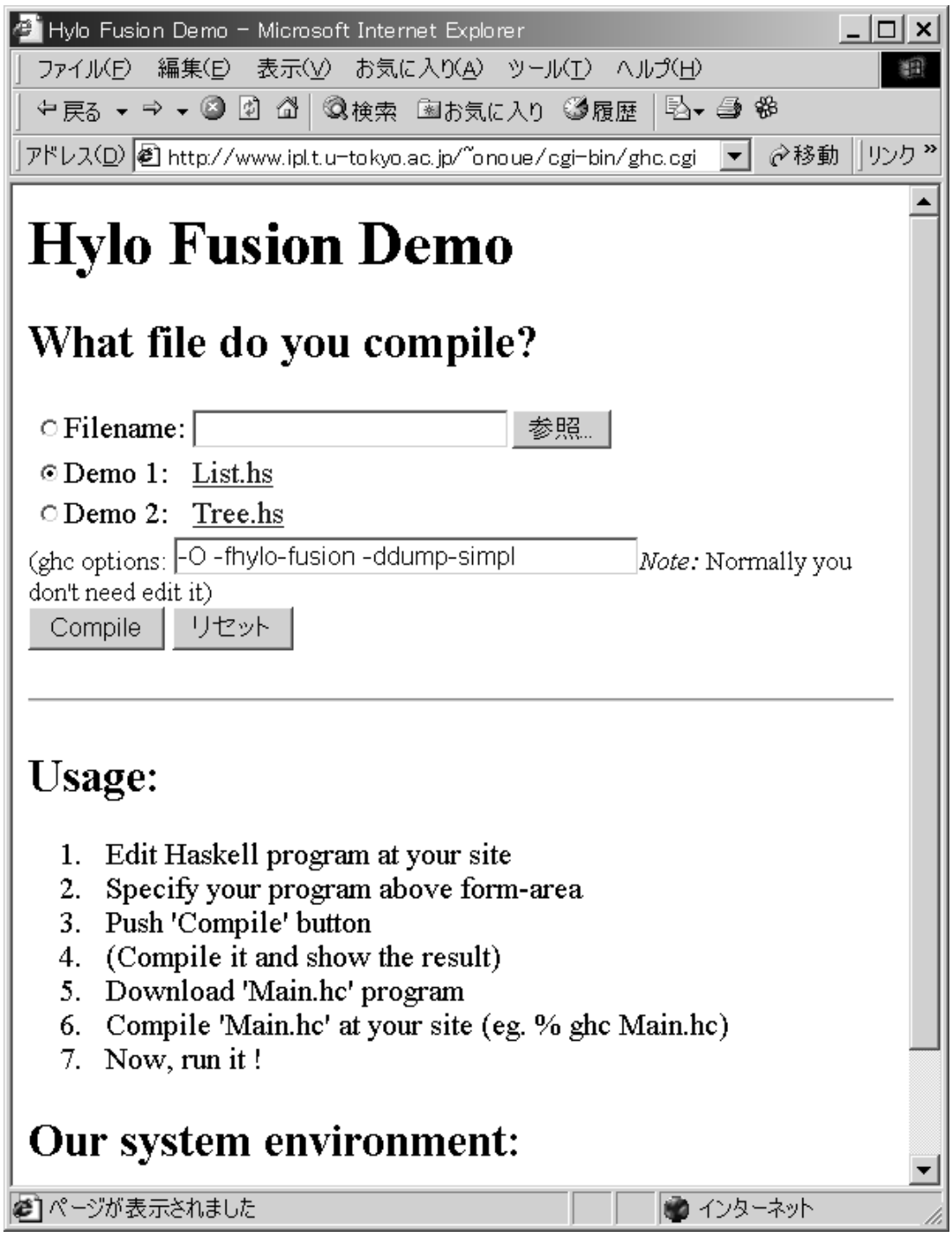


図 4.9. 変換対象プログラムの指定画面

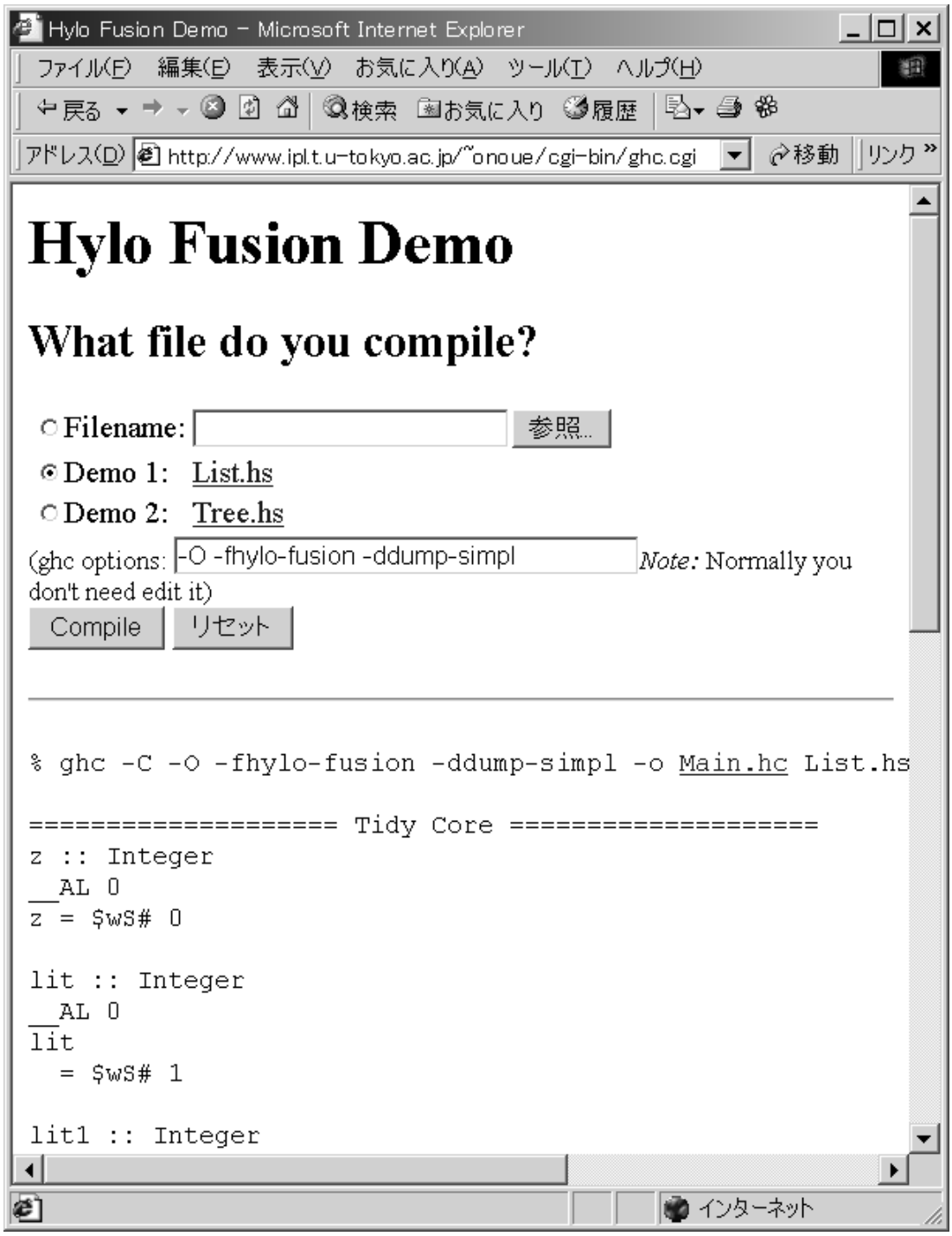
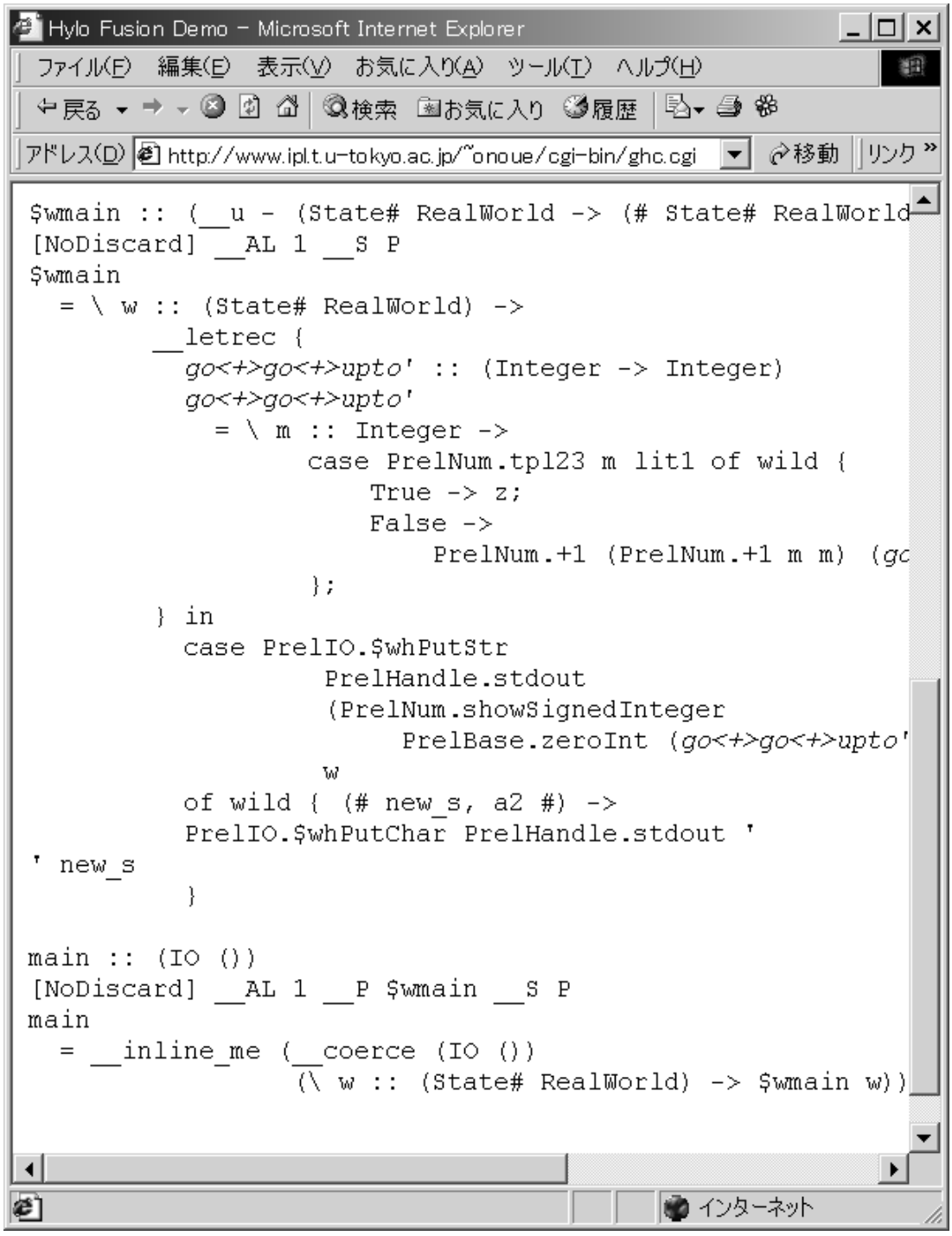


図 4.10. 変換されたプログラムの表示画面




```
$wmain :: (__u - (State# RealWorld -> (# State# RealWorld
[NoDiscard] __AL 1 __S P
$wmain
  = \ w :: (State# RealWorld) ->
        __letrec {
          go<+>go<+>upto' :: (Integer -> Integer)
          go<+>go<+>upto'
            = \ m :: Integer ->
                  case PrelNum.tpl23 m lit1 of wild {
                      True -> z;
                      False ->
                          PrelNum.+1 (PrelNum.+1 m m) (gc
                  };
        } in
          case PrelIO.$whPutStr
                   PrelHandle.stdout
                   (PrelNum.showSignedInteger
                        PrelBase.zeroInt (go<+>go<+>upto'
                   w
          of wild { (# new_s, a2 #) ->
          PrelIO.$whPutChar PrelHandle.stdout '
' new_s
          }

main :: (IO ())
[NoDiscard] __AL 1 __P $wmain __S P
main
  = __inline_me (__coerce (IO ())
                 (\ w :: (State# RealWorld) -> $wmain w))
```

図 4.11. 変換されたプログラムの表示画面 (2)

第 5 章

融合変換の性能評価

本章では，まずはじめにインタープリタを基にしたプロトタイプ上で，n クィーン問題を解くプログラムを手で融合変換したものの実行効率を比較する実験を行ない，融合変換の有効性を検証する．次に，既存のコンパイラである Glasgow Haskell Compiler (GHC) に対して，融合変換を行なう処理を追加することによって，任意の Haskell プログラムを変換することが可能となる汎用的なコンパイラを作成し，実用的な規模のベンチマークを行なうことによって，その長所短所を広く調べることにする．

5.1 性能評価 [IFIP97]

本論文で示したアルゴリズムはすべて実装済である．HYLO システムの有効性を評価し，我々の手法によってどれくらいの効率改善が得られたかを調べるために，3 つのプログラム ssf, $unlines$, $queens$ に対して融合変換の前後における性能の差を調査した．ssf はこれまでみてきた例である，$unlines$ は Gofer 処理系のプレリュードにあるプログラムで，文字列のリストを受け取りすべての文字列を CR 文字 (“`[]`”) を狭んで結合した 1 つの新しい文字列を返す．$queens$ は有名な n クィーン問題を解くプログラムで，$n \times n$ のチェス盤上に n 個のクィーンを互いに取り合わない位置に並べる配置を求める．

今回の実験では，Gofer インタープリタ (2.30a 版) と Glasgow Haskell コンパイラ (GHC と略, 0.29 版) の 2 つの処理系を用いた．まず第一に，これらのプログラムを Gofer 処理系を用いて評価した．実験の結果を表 5.1 にある通り，時間 (簡約段数) と空間 (使用ヒープセル数) の両面において改善がみられた．例えば，ssf と $unlines$ の例に関して，HYLO システムを用いると簡約段数とヒープセルの両方とも約半分に減少されているのがわかる．より具体的にみるために，$queen$ の問題に対して，変換前と変換後のプログラムを図 5.1,5.2 に示す．これより HYLO システムの変換能力が，それほど単純なものではないことがわかるであろう．

次に，$queen$ プログラムを元のものと変換後のものの両方をコンパイルし，我々の融合変換を shortcut 融合変換 [GLJ93] と比較することにする． shortcut 融合変換は，既に GHC に組み込まれている hylo 融合変換と同様の効果をもたらす変換の一つであり，そのアルゴリズムは関連研究 (6.2.1節) に示す．GHC において shortcut 融合変換を有効/無効にするためには，コンパイル時のオプションである `-O` と `-fno-foldr-build` を指定すればよい．表 5.2 にある実験結果は，我々の融合変換が shortcut 融合変換より速度では約 20%，ヒー

表 5.1. Experimental results using Gofer († applies to a text with 100 words)

Program	Reduction Steps			Heap Cells		
	before fusion	after fusion	ratio	before fusion	after fusion	ratio
$ssf(1000)$	11,015	6,005	0.54	17,030	10,019	0.59
$unlines^{\dagger}$	4,151	1,759	0.42	8,079	4,393	0.54
$queens(10)$	33,776,593	26,419,252	0.89	65,428,706	50,839,988	0.78

```
nqueen = 10

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

図 5.1. 変換前の *queens* プログラム

表 5.2. Experimental results of *queens* using GHC

	Total Time (secs)		**Heap Cells (mbytes)**	
	before fusion	after fusion	before fusion	after fusion
by cheap fusion	9.41	5.13	61.27	15.55
by hylo fusion	9.41	4.10	61.27	9.38

プセルでは 60% ほど，よい結果を得られることを示している．なお GHC を用いたときの改善度が Gofer を用いたときと比べてより大きいことは，興味のある結果である．実行時間が 9.41 秒から 4.10 秒へ，ヒープセルが 61.27 MB から 9.38 MB へそれぞれ減少している．

5.2 プロトタイプ上での実験 (n-queens プログラム) [JSSST98]

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 [OHIT97] 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 [GLJ93] 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．

5.2.1 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．

```
nqueen = 10

queens = print (
  let
    x681 [] = 0
    x681 ((:) x342 x343)
     = let
        x680 [] = x681 x343
        x680 ((:) x351 x352) = (+) x351 (x680 x352)
       in x680 x342
  in x681
      (let
        x687 x354
         = case ((==) x354 0) of
            True -> (:) [] []
            False ->
             let
              x686 [] = []
              x686 ((:) x375 x376)
               = let
                  x685 x528
                   = case ((<=) x528 nqueen) of
                      True ->
                       case (let
                              x404 = (+) 1 (length x375)
                             in let
                                 x682 ((:) x446 x447,(:) x448 x449)
                                  = (&&) (not ((||) ((==) x448 x528)
                                              ((||) ((==) ((+) x446 x448)
                                                          ((+) x404 x528))
                                                    ((==) ((-) x446 x448)
                                                          ((-) x404 x528)))))
                                        (x682 (x447,x449))
                                 x682 x450 = True
                                in x682 (let
                                           x683 x458
                                             = (:) x458 (x683 ((+) 1 x458))
                                          in x683 1,x375)) of
                        True -> (:) (let
                                       x684 [] = (:) x528 []
                                       x684 ((:) x508 x509)
                                                = (:) x508 (x684 x509)
                                      in x684 x375) (x685 ((+) 1 x528))
                        False -> x685 ((+) 1 x528)
                      False -> x686 x376
                 in x685 1
             in x686 (x687 ((-) x354 1))
       in x687 nqueen))
```

図 5.2. 変換後の *queens* プログラム

表 5.3. 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

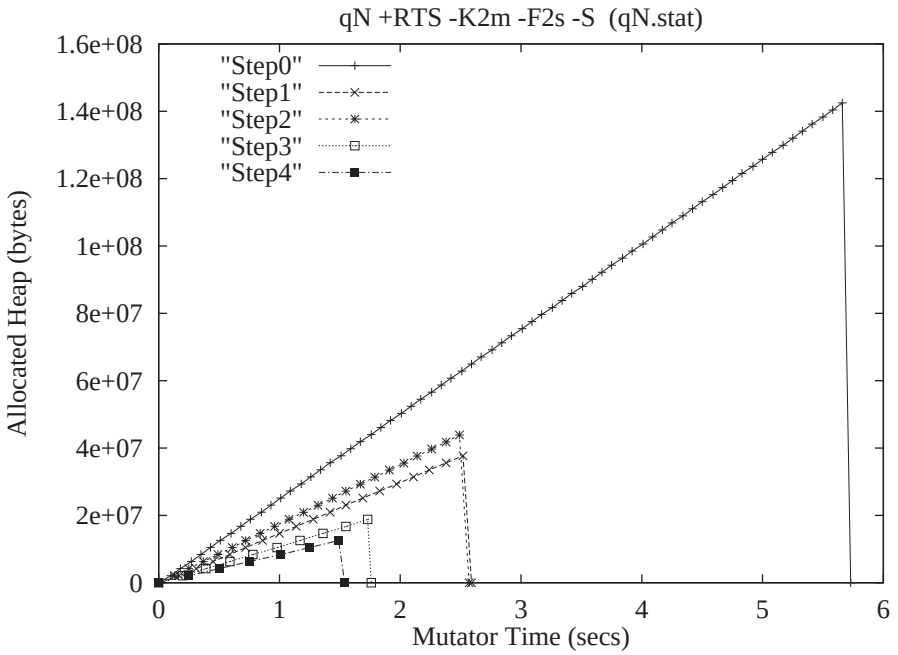


図 5.3. ヒープ割当量の時間変化

表 5.3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 5.3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 5.3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 5.3の数値よりも大きくなっているのは，図 5.3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 5.4∼5.8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 5.3の Step 0 と図 5.4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 5.4が低い水平なグラフになっているのは，図 5.4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．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様子がわかる．

5.2.2 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．

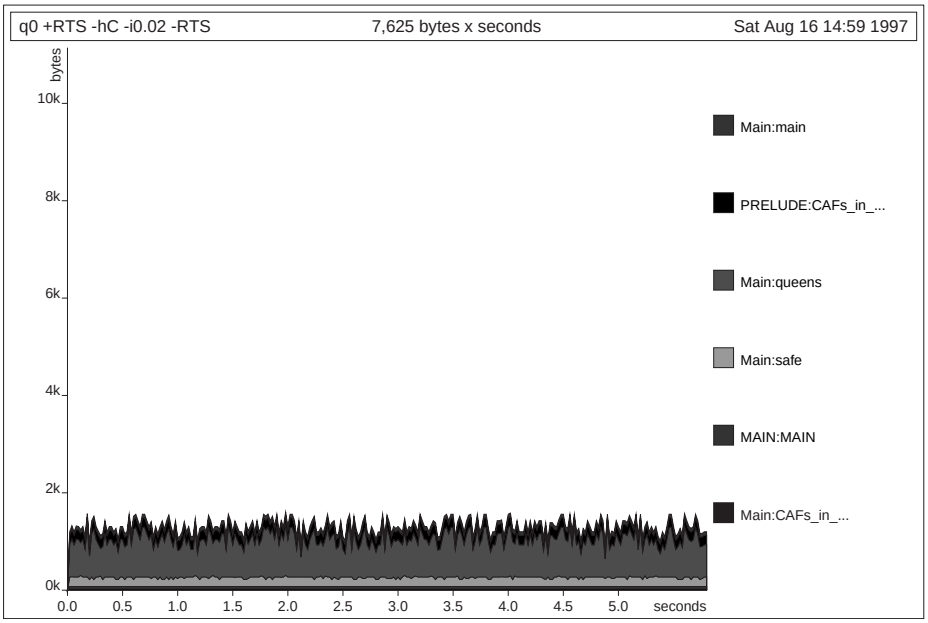


図 5.4. Step 0: 変換前のプログラム

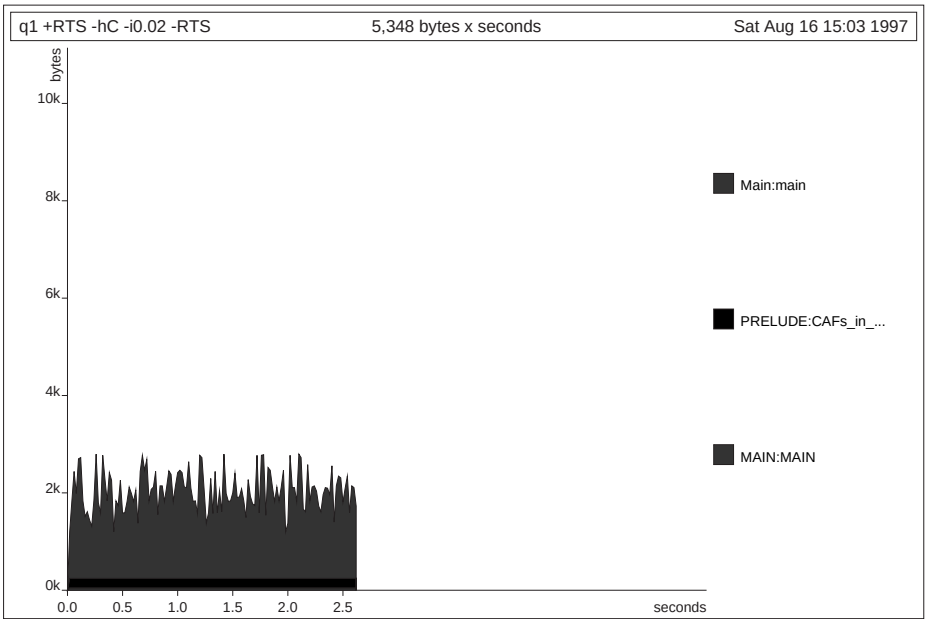


図 5.5. Step 1: 関数を局所定義にする

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局所定義による最適化のメリットを受けることになる．

5.2.3 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．

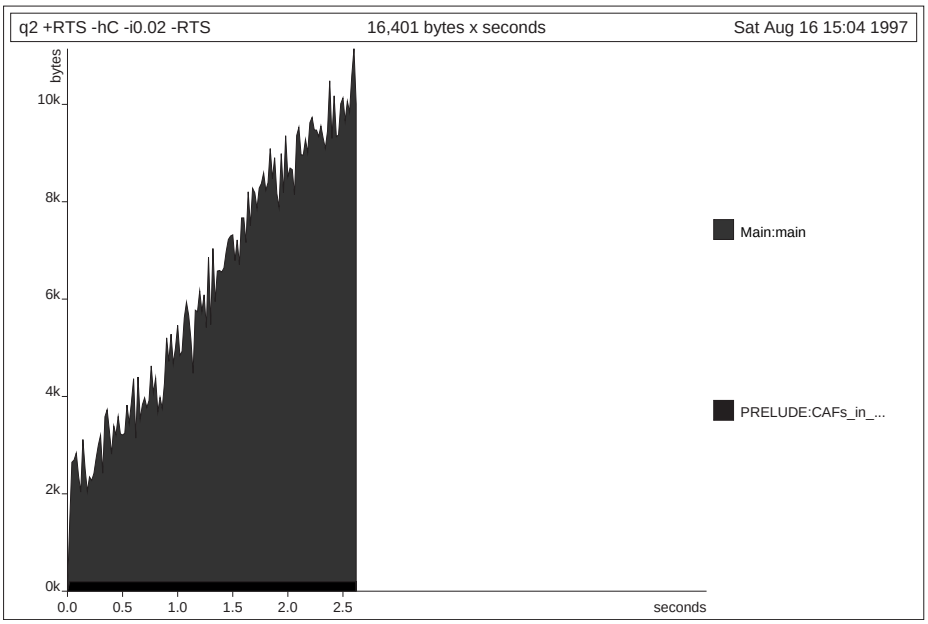


図 5.6. Step 2: 式 (`safe p n`) を除く融合変換

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 5.3)．これは，Step 1 におけるリストの内包表記 (list comprehension) が Step 2 では展開され無くなった結果，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 5.5と図 5.6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．

5.2.4 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 5.8を一回り大きくしたものに近い形状をしている．

5.2.5 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ので，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 [OHIT97] 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 5.3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．

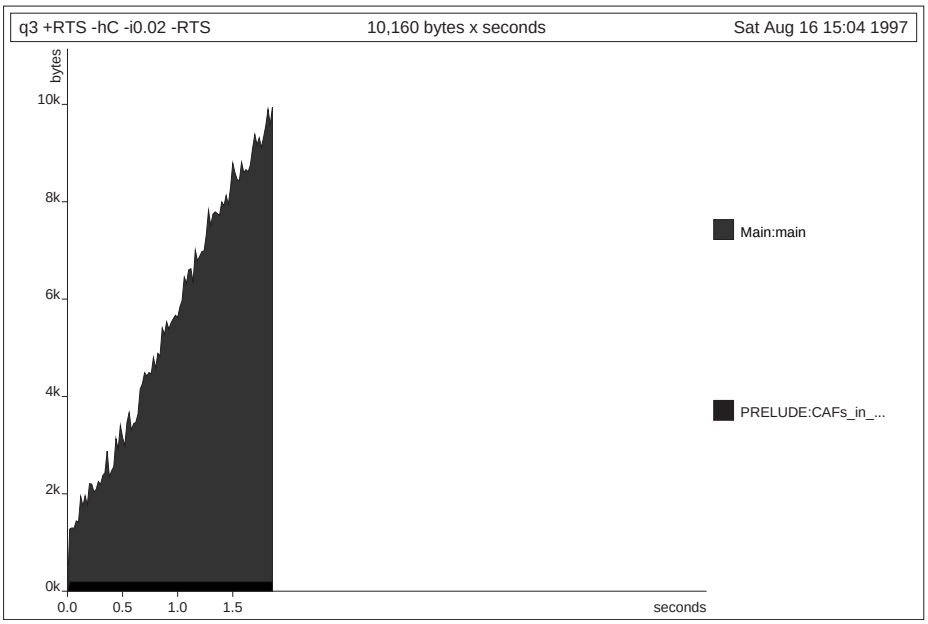


図 5.7. Step 3: システムによる全体の融合変換

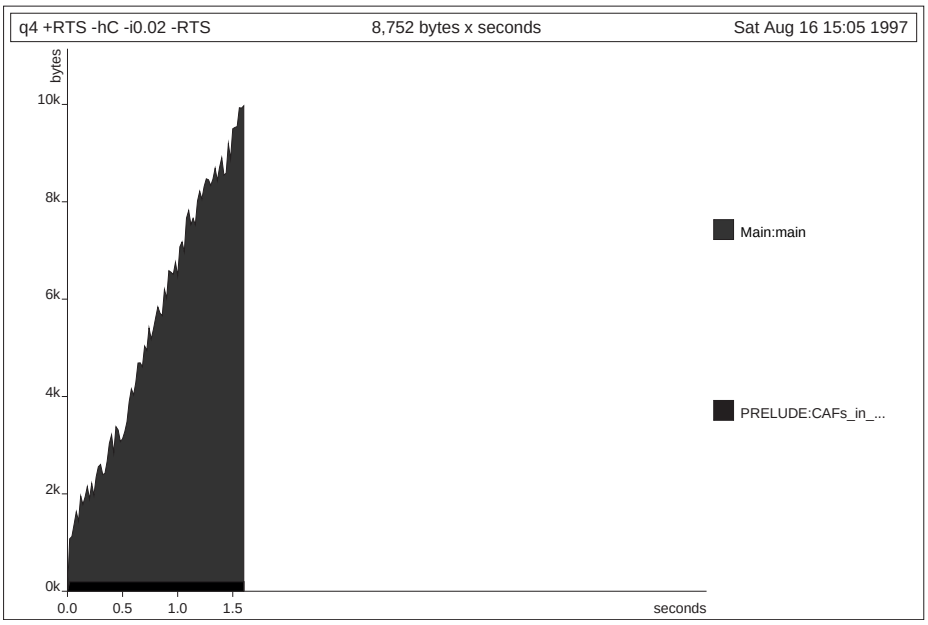


図 5.8. Step 4: 再帰関数の構成子式への適用

5.2.6 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．

今後の課題としては，まず第一により多くの多様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．更に，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．

5.3 NoFib ベンチマーク

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．この実験は当初 ghc の古い版に対して行なっていたが [OHIT00]，今回は実験環境を新たにして再度その最適化の効果を測定した．

5.3.1 実験

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[NoF] 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．小規模に相当する imaginary ベンチマークは 11 種のプログラムからなり，n クィーンやフィボナッチ関数などの簡単な数行からなるプログラムである．図 5.6 において行数が多いものが存在するのは，コメント行も含まれていることを注意しなければならない．中規模に相当する spectral ベンチマークは，3 群の中でもっとも多い 53 種類のベンチマークからなる．また，この中の 15 種類は Hartel ベンチマーク [Har91, HL93] という SASL,Hope 言語用に記述されたベンチマークを Haskell 言語に移植したものを含んでいる．大規模にあたる real ベンチマークは，行数の多いプログラムの集まりで，ある規模の問題を解決するために実用されているものばかりである．

対象としたプログラムに関しては，表 5.6,5.4, 5.5,5.7 にそれぞれ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. hylo 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．このとき GHC に標準で実装されている shortcut 融合変換の有効無効と合わせて，4 通りの最適化を用いてコンパイルを行なう．
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．

また，実験に用いた計算機は PC/AT 互換機 (PentiumIII 1.26GHz デュアル CPU, メモリ 1,024MB) 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，CPU やメモリ搭載量が異なる機種でも常に同じ値となることが知られている．

なお NoFib ベンチマークで実験を行なうには，単に make コマンドを起動するだけでよい．すると，予め用意された Makefile 内に記述された規則に従って，各種のベンチマークがコンパイルされるとともに，コンパイルに関する情報が出力される．また生成された実行ファイルは，起動されたスクリプト runstdtest によって，実行時間などの情報を得るとともに，予め用意してある正しい結果と比較しコンパイルが正常に行なえたかも確認することができる．今回の実験で採取したデータは，Makefile の規則やスクリプト runstdtest を用いることによって，以下の方法により計測されたものであることを明記しておく．

コンパイル時間 OS の time コマンドによる user 時間出力
バイナリサイズ OS の size コマンド出力における text と data の和
実 行 時 間 `+RTS -S` オプションによる統計情報 INIT, MUT, GC の和
メモリ割り当て `+RTS -S` オプションによる統計情報のヒープ割り当て (allocated in the heap)

また，これらを集計するために，GHC の配布ソースの中に含まれているユーティリティツールである nofib-analysis コマンドを利用した．このコマンドは，NoFib ベンチマークを実行した際に得られた出力結果 (これにはコンパイルに関する情報や実行時に関する情報などがすべて含まれる) を入力として受け取り，これを人間が読みやすいように表形式へと変換するものである．なお時間の計測はばらつくことが予想されるため，各ベンチマークはそれぞれ 3 回ずつ実行し，その平均を結果としてのせている．

表 5.4. spectral ベンチマーク

プログラム	内容	行数
ansi	文字端末制御	125
atom	原子の運動	183
awards	受賞者決定手法	111
banner	簡単なバナープログラム	108
boyer	Gabriel スィーツ boyer ベンチマーク	1014
boyer2	Gabriel スィーツ boyer ベンチマーク	723
calendar	Unix 'cal' コマンド	140
cichelli	完全 hash 関数	244
circsim	電子回路シミュレータ	668
clausify	節形式の命題	179
constraints	制約解消システム	267
cryptarithm1	覆面算ソルバ	164
cryptarithm2	覆面算ソルバ	326
cse	共通部分式削除	464
eliza	偽精神科医プログラム	266
expert	エキスパートシステム	525
fft2	フーリエ変換	216
fibheaps	フィボナッチヒープ	294
fish	図形描画関数	125
gcd	最大公約数	56
integer	整数 (Integer) による演算	68
knights	knight's ツアー	887
life	ライフゲーム	53
mandel	マンデルブロー集合	498
mandel2	マンデルブロー集合	222
minimax	三目並べ	238
multiplier	二進積算器	498
para	段落整形プログラム	1781
power	べき級数展開	138
pretty	プリティプリンタ	265
primetest	素数判定	292
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
simple	標準的 Id ベンチマーク	1129
sorting	ソーティングアルゴリズム	162
sphere	球体に対するレイトレーシング	472
treejoin	木の追加	121

表 5.5. spectral ベンチマーク (hartel)

プログラム	内容	行数
comp_lab_zift	画像処理アプリケーション	884
event	フリップフロップのイベント駆動	451
fft	2 つの高速フーリエ変換	412
genfft	合成の FFT プログラムの生成	502
ida	n パズルのある配置の解決	490
listcompr	リスト内包表記のコンパイル	522
listcopy	コピーを伴う内包表記のコンパイル	527
nucleic2	原子の 3 次元構造	3389
parstof	Wadler の手法に基づく字句解析と構文解析	1280
sched	並列ジョブの最適スケジュール問題	555
solid	計算幾何学における固定モデル問題	1244
transform	同期プロセス型からマスタースレーブ型への変換	1142
typecheck	関数定義に対する多様型型チェック	658
wang	線型方程式に対する wang アルゴリズム	357
wave4main	8x8 面における水位計算	599

表 5.6. imaginary ベンチマーク

プログラム	内容	行数
exp3_8	3 の 8 乗を計算	89
gen_regexps	簡略化された正規表現の展開	32
integrate	8 分法による積分計算	38
paraffins	パラフィンの構造解析	88
primes	エラトステネスの篩による素数生成	13
queens	10 クィーン問題	14
rfib	素朴なフィボナッチ関数	9
tak	Macarthy の tak 関数	10
wheel-sieve1	遅延 wheel sieve 法による素数生成	35
wheel-sieve2	遅延 wheel sieve 法による素数生成	43
x2n1	n 乗すると 1 になる複素数の生成	32

今後 2 種類の融合変換の組み合わせた場合を，以下のように簡略化した 2 文字の名称で呼ぶことにする．例えば $\mathcal{HY}$ は hylo-fusion のみを有効にしている状態のことを表わす．なおここでは shortcut 融合変換を foldr/build で表わすものとする．

$\mathcal{NA}$ No fusion
$\mathcal{HY}$ hylo-fusion
$\mathcal{FB}$ foldr/build
$\mathcal{FH}$ foldr/build + hylo-fusion

また shortcut 融合変換の無効化については，以下のような手順で行なった． shortcut 融合変換は GHC が Haskell 言語に対して拡張したプラグマの機構を用いて実装されている．プラグマとは，本来プログラムの持つ意味を変えることなく，その実行に関する諸情報をコンパイラ等に指定する命令をコメント中に埋め込む

表 5.7. real ベンチマーク

プログラム	内容	行数
anna	正格性解析 (strictness analysis)	9561
bspt	BSP (binary space partition) 木モデラー	2141
cacheprof	キャッシュミスのプロファイラ	2151
compress	テキスト圧縮	736
compress2	テキスト圧縮	199
fem	有限要素法を用いたトラス構造の解析	1286
fluid	流体力学プログラム	2401
fulsom	固形モデリング	1397
gamteb	モンテカルロ法による光子輸送計算	701
gg	GRIP 統計からのグラフ	812
grep	正規表現を用いた文字列検索	356
hidden	隠線除去	521
hpg	Haskell プログラムの生成	2067
infer	Hindley-Milner の型推論	594
lift	完全遅延ラムダ式の持ち上げ	2033
maillist	メーリングリストの生成	175
mkhprog	コマンドラインパーザ生成器	803
parser	Haskell のサブセットに対応したパーザ	3139
pic	セルの中の粒子	527
prolog	ミニ Prolog インタープリタ	641
reptile	Escher のタイル問題	1522
rsa	RSA 暗号	74
symalg	算術演算インタープリタ	1146
veritas	定理証明器	11124

機能のことで，通常の Haskell のコメント行を拡張した `{-#` *word* ... `#-}` のような記述方法を用いる．ここで *word* がプラグマの種類を示し，通常その後に各プラグマに必要なオプション情報を付加することが多い．GHC で利用可能なプラグマの種類としては，RULES の他に INLINE, NOINLINE, SPECIALIZE, LINE, DEPRECATED などがある．

さて， shortcut 融合変換は現在 RULE プラグマを用いて，書き換え規則をプログラム中に指定することによって実装されている．しかし我々の融合変換との効果の比較を行なうためには，類似の効果を持つ shortcut 融合変換を一時的に無効にした方が，その効果を比較しやすいものと思われる．そこで我々の実装では，コンパイル時にオプションを指定すると RULES プラグマによる書き換え規則の適用を抑制できるように変更した．しかし RULES プラグマは shortcut 融合変換以外にも，汎用関数の引数の型が判別したときの特殊化や，簡単な不等号計算などの他の処理も行なっているため，RULES の適用を一斉に止めてしまうことは効率を大きく損ねてしまい比較の対象とする意味がなくなる．これを解決するために，各 RULES プラグマに付けられた名前を用いて， shortcut 融合変換に関係する RULES であると判明した場合にはプラグマの適用を行なわないように実装を工夫した．具体的には，GHC のソースツリーにおける ghc/specialise/Rules.lhs モジュールにおける matchRules 関数において，以下のような条件をもつ規則は shortcut 融合変換に関するものとし，適用しないようにした．各条件は自然数 n と文字列 s のペア (n, s) から構成され，比較対象の規則名の先頭 n 文字を取得して文字列 s と等しい場合にはそれが shortcut 融合変換の規則であるものとみなしている．

```
rules_deny
     = [
        -- PrelBase.lhs
        (5, "fold/"), (6, "foldr/"), (7, "augment"),
        (5, "mapFB"), (7, "mapList"),
        (11, "unpack-list"), (13, "unpack-append"),
        -- PrelEnum.lhs
        (11, "eftCharList"), (11, "efdCharList"), (12, "efdtCharList"),
        (10, "eftIntList"), (10, "efdIntList"), (11, "efdtIntList"),
        -- PrelList.lhs
        (4, "head"), (8, "filterFB"), (10, "filterList"),
        (9, "iterateFB"), (8, "repeatFB"),
        (9, "and/build"), (8, "or/build"),
        (9, "any/build"), (9, "all/build"),
        (7, "foldr2/"), (7, "zipList"), (11, "zipWithList"),
        -- PrelNum.lhs
        (16, "enumDeltaInteger"), (18, "enumDeltaToInteger")
       ]
```

なお注意点として build 形式を作るだけの規則に関しては上記禁止リストに含めていない．例えば "map" プラグマは forall f xs. map f xs = build (\c n -> foldr (mapFB c f) n xs) のように定義されているが，この規則まで無効にしてしまうと build 形式やその先の再帰関数の定義がプログラム中に含まれる状態にならないので，その後 hylo 変換を行なうことができなくなるためである．

図 5.8,5.9 は，各ベンチマークプログラムを融合変換を用いないでコンパイルした際にかかった所要時間とバイナリの大きさ，ならびに実行時間とヒープ使用量をアルファベット順に示してある．また時間やサイズとの比較のため，元プログラムの行数を再掲してある．この図より，行数にはコメント等を含んではいるものの，行数とコンパイル時間の間には一次的な関係が成り立っているものと思われる．一方行数とオブジェクトサイズの間には，それほど強い相関関係があるようには見えない．例えば，lift や nucleic2, parser などは 2,000 行以上からなる大きなプログラムであるが，コンパイル後のサイズは 300KB 近くとそれほど大きくもない．これは，コンパイラ内部で行なわれる各種の最適化によって，結果的に簡略化されたプログラムが生成されたものと思われる．

5.3.2 実験結果の考察

図 5.10∼5.21 までに，各融合変換の有無に応じたプログラムのオブジェクトサイズ，コンパイル時間，実行時のメモリ割り当て量，実行時間を，融合変換無の状態 NA を基のした相対量で示したものである．例えば hylo 変換を用いた際の，実行時ヒープ使用量が 0.80 であったとすると，hylo 変換を用いることによって，融合変換を用いなかった場合に比べヒープ使用量が 20% 減少したことを意味する．

表 5.8. 最適化を行なわない状態

ベンチマーク	ソース		オブジェクト		実行	
	モジュール	行数	時間 [sec]	サイズ [KB]	時間 [sec]	サイズ [KB]
anna	32	9561	119.78	1207	0.46	32338
ansi	1	125	2.65	212	0.00	51
atom	1	183	2.04	230	1.60	203143
awards	2	111	2.54	191	0.00	141
banner	1	108	8.46	273	0.00	111
boyer	1	1014	4.07	196	0.17	26613
boyer2	5	723	8.32	242	0.02	1848
bspt	17	2141	29.43	374	0.01	5515
cacheprof	3	2151	50.14	639	2.33	257888
calendar	1	140	2.94	185	0.00	384
cichelli	5	244	5.52	233	0.43	36220
circsim	1	668	9.37	281	5.52	705354
clausify	1	179	3.18	188	0.14	19086
comp_lab_zift	1	884	8.69	227	1.19	157496
compress	10	736	8.12	242	0.84	137842
compress2	3	199	3.68	225	0.77	63653
constraints	1	267	4.16	196	13.21	1008018
cryptarithm1	1	164	1.22	170	7.75	997236
cryptarithm2	3	326	4.24	200	0.09	15918
cse	2	464	4.61	208	0.00	557
eliza	1	266	8.05	268	0.00	316
event	1	451	3.11	179	0.89	60513
exp3_8	1	89	1.49	173	0.76	158308
expert	6	525	9.27	275	0.00	150
fem	17	1286	29.07	457	0.13	24846
fft	1	412	5.51	250	0.10	13900
fft2	3	216	6.38	338	0.35	46324
fibheaps	1	294	3.50	228	0.20	33179
fish	1	125	3.52	181	0.02	7087
fluid	18	2401	42.74	641	0.03	3607
fulsom	13	1397	24.35	477	2.42	231058
gamteb	13	701	15.52	344	0.48	52140
gcd	1	56	1.27	177	0.20	29692
gen_regexps	1	32	2.04	210	0.00	1065
genfft	1	502	5.32	198	0.12	26148
gg	9	812	30.45	541	0.03	4568
grep	3	356	7.42	252	0.00	28
hidden	15	521	18.05	458	4.30	553364
hpg	8	2067	20.85	445	0.24	44248
ida	1	490	4.49	189	0.38	61343
infer	16	594	16.58	341	0.27	19723
integer	1	68	1.28	173	12.94	1656637
integrate	1	38	1.79	281	3.28	441926
knights	6	887	10.40	235	0.01	1020

表 5.9. 最適化を行なわない状態 (cont.)

ベンチマーク	ソース		オブジェクト		実行	
	モジュール	行数	時間 [sec]	サイズ [KB]	時間 [sec]	サイズ [KB]
life	1	53	2.17	183	1.59	220497
lift	5	2033	10.70	245	0.00	398
listcompr	1	522	5.06	196	0.73	130634
listcopy	1	527	5.09	197	0.78	145322
maillist	1	175	2.50	224	0.08	21137
mandel	4	498	4.38	382	1.38	143551
mandel2	1	222	2.46	181	0.02	4194
minimax	6	238	6.52	235	0.02	2812
mkhprog	1	803	6.54	253	0.00	1674
multiplier	1	498	3.76	194	0.71	106449
nucleic2	6	3389	19.68	318	0.40	46688
para	1	1781	3.51	237	2.99	348955
paraffins	1	88	3.37	188	0.44	24960
parser	2	3139	21.06	360	0.10	14502
parstof	1	1280	59.79	522	0.42	36023
pic	9	527	14.44	394	0.02	3602
power	1	138	7.86	253	4.52	226571
pretty	3	265	3.72	241	0.00	61
primes	1	13	1.08	169	0.18	27817
primetest	4	292	3.99	253	1.06	67612
prolog	9	641	9.21	268	0.00	960
puzzle	1	170	3.88	194	0.94	93590
queens	1	14	1.01	165	0.10	11860
reptile	13	1522	30.17	472	0.03	7733
rewrite	1	631	8.35	232	0.13	18757
rfib	1	9	0.90	218	0.13	25
rsa	2	74	2.54	249	0.24	16934
scc	2	100	1.30	169	0.00	25
sched	1	555	3.48	179	0.10	17846
simple	1	1129	31.01	503	2.53	201138
solid	1	1244	39.90	442	0.81	84782
sorting	2	162	3.33	213	0.00	579
sphere	1	472	7.31	321	0.68	57475
symalg	11	1146	22.67	518	0.55	69004
tak	1	10	1.07	172	0.09	9760
transform	1	1142	27.26	391	2.18	316824
treejoin	1	121	2.20	213	1.04	62643
typecheck	1	658	7.13	214	1.38	149308
veritas	32	11124	106.62	1053	0.00	608
wang	1	357	4.04	239	0.44	43193
wave4main	1	599	5.29	199	2.60	313811
wheel-sieve1	1	35	1.55	173	2.07	27762
wheel-sieve1	1	43	1.57	174	1.41	37353
x2n1	1	32	2.30	308	0.12	18623

表 5.10. バイナリサイズの変化 (hylo-fusion)

ベンチマーク	比率
solid	0.82
parstof	0.87
reptile	0.94
transform	0.94
infer	0.95
simple	0.96
lift	0.97
fem	0.98
grep	0.98
mkhprog	0.98
pic	0.98
prolog	0.98
anna	0.99
bspt	0.99
cacheprof	0.99
comp_lab_zift	0.99
cryptarithm2	0.99
event	0.99
gcd	0.99
hpg	0.99
listcopy	0.99
parser	0.99
sphere	0.99
symalg	0.99
⋮	
constraints	1.01
exp3_8	1.01
fulsom	1.01
life	1.01
veritas	1.01
eliza	1.02
genfft	1.02
gg	1.02
fluid	1.03
gamteb	1.03
paraffins	1.03
puzzle	1.03
compress	1.05
nucleic2	1.06
cse	1.07
wave4main	1.38
48 others	1.00
Min	0.82
Max	1.38
Geom ave.	1.00

表 5.11. バイナリサイズの変化 (foldr/build)

ベンチマーク	比率
parstof	0.61
solid	0.64
transform	0.75
reptile	0.77
anna	0.78
cacheprof	0.78
veritas	0.78
banner	0.79
gg	0.85
eliza	0.86
fem	0.89
symalg	0.89
boyer2	0.90
bspt	0.91
mkhprog	0.91
parser	0.91
simple	0.91
comp_lab_zift	0.92
typecheck	0.92
expert	0.93
genfft	0.93
listcompr	0.93
listcopy	0.93
rewrite	0.93
circsim	0.94
fluid	0.94
hpg	0.94
knights	0.94
minimax	0.94
pic	0.94
prolog	0.94
cse	0.95
infer	0.95
lift	0.95
multiplier	0.95
ansi	0.96
cichelli	0.96
clausify	0.96
⋮	
fulsom	0.96
gamteb	0.96
hidden	0.96
calendar	0.97
constraints	0.97
fft	0.97
power	0.97
puzzle	0.97
wave4main	0.97
compress	0.98
cryptarithm2	0.98
fft2	0.98
grep	0.98
ida	0.98
life	0.98
maillist	0.98
mandel	0.98
paraffins	0.98
sched	0.98
sorting	0.98
sphere	0.98
wang	0.98
atom	0.99
awards	0.99
boyer	0.99
event	0.99
fibheaps	0.99
fish	0.99
gcd	0.99
mandel2	0.99
para	0.99
pretty	0.99
primes	0.99
primetest	0.99
treejoin	0.99
integer	1.01
14 others	1.00
Min	0.61
Max	1.01
Geom ave.	0.94

表 5.12. バイナリサイズの変化 (fo/bu + hylo)

ベンチマーク	比率
parstof	0.62
solid	0.64
transform	0.74
reptile	0.77
anna	0.78
cacheprof	0.78
veritas	0.78
banner	0.79
gg	0.86
eliza	0.89
fem	0.89
symalg	0.89
boyer2	0.90
bspt	0.91
comp_lab_zift	0.91
mkhprog	0.91
parser	0.91
simple	0.91
infer	0.92
typecheck	0.92
expert	0.93
listcompr	0.93
listcopy	0.93
circsim	0.94
fluid	0.94
genfft	0.94
⋮	
hpg	0.94
knights	0.94
lift	0.94
minimax	0.94
pic	0.94
prolog	0.94
rewrite	0.94
cse	0.95
multiplier	0.95
ansi	0.96
cichelli	0.96
clausify	0.96
gamteb	0.96
hidden	0.96
calendar	0.97
constraints	0.97
cryptarithm2	0.97
fft	0.97
fulsom	0.97
grep	0.97
power	0.97
puzzle	0.97
fft2	0.98
ida	0.98
maillist	0.98
mandel	0.98
⋮	
paraffins	0.98
sched	0.98
sorting	0.98
sphere	0.98
wang	0.98
atom	0.99
awards	0.99
boyer	0.99
event	0.99
fibheaps	0.99
fish	0.99
gcd	0.99
life	0.99
mandel2	0.99
para	0.99
pretty	0.99
primes	0.99
primetest	0.99
treejoin	0.99
wheel–sieve1	0.99
integer	1.01
nucleic2	1.01
compress	1.04
wave4main	1.08
12 others	1.00
Min	0.62
Max	1.08
Geom ave.	0.94

表 5.13. コンパイル時間の変化 (hylo-fusion)

ベンチマーク	比率
solid	0.84
infer	0.95
lift	0.96
cryptarithm2	0.97
parstof	0.97
reptile	0.97
tak	0.97
wheel–sieve1	0.97
awards	0.98
gcd	0.98
pic	0.98
primes	0.98
calendar	0.99
exp3_8	0.99
fft2	0.99
grep	0.99
minimax	0.99
mkhprog	0.99
prolog	0.99
comp_lab_zift	1.01
event	1.01
hidden	1.01
integrate	1.01
mandel	1.01
transform	1.01
cichelli	1.02
circsim	1.02
compress2	1.02
⋮	
expert	1.02
fft	1.02
fibheaps	1.02
gen_regexps	1.02
ida	1.02
listcopy	1.02
mandel2	1.02
pretty	1.02
primetest	1.02
rsa	1.02
sched	1.02
sorting	1.02
anna	1.03
boyer	1.03
boyer2	1.03
bspt	1.03
fish	1.03
fulsom	1.03
integer	1.03
knights	1.03
multiplier	1.03
power	1.03
treejoin	1.03
wang	1.03
wheel–sieve2	1.03
clausify	1.04
listcompr	1.04
maillist	1.04
⋮	
⋮	
parser	1.04
queens	1.04
veritas	1.04
atom	1.05
banner	1.05
typecheck	1.05
ansi	1.06
constraints	1.06
fluid	1.06
gg	1.06
life	1.06
cacheprof	1.07
genfft	1.07
cryptarithm1	1.08
gamteb	1.08
rewrite	1.08
scc	1.09
nucleic2	1.14
puzzle	1.14
eliza	1.23
paraffins	1.29
cse	1.30
wave4main	2.39
compress	13.84
8 others	1.00
Min	0.84
Max	13.84
Geom ave.	1.07

表 5.14. コンパイル時間の変化 (foldr/build)

ベンチマーク	比率
banner	0.36
solid	0.38
parstof	0.46
eliza	0.49
transform	0.61
mkhprog	0.62
reptile	0.65
cacheprof	0.67
ansi	0.68
boyer2	0.68
gg	0.71
genfft	0.72
symalg	0.74
listcompr	0.75
listcopy	0.75
multiplier	0.75
typecheck	0.75
anna	0.77
comp_lab_zift	0.77
veritas	0.77
cse	0.78
life	0.78
calendar	0.79
clausify	0.79
minimax	0.79
maillist	0.80
rewrite	0.80
circsim	0.82
expert	0.82
⋮	
pic	0.83
simple	0.83
cichelli	0.84
fem	0.84
fft	0.84
parser	0.84
puzzle	0.84
mandel	0.85
prolog	0.85
bspt	0.87
cryptarithm2	0.87
gen_regexps	0.87
sorting	0.87
fft2	0.88
hpg	0.88
lift	0.88
atom	0.89
constraints	0.89
infer	0.89
knights	0.89
paraffins	0.89
power	0.89
wang	0.90
awards	0.91
fluid	0.91
hidden	0.91
treejoin	0.91
fulsom	0.92
gamteb	0.92
⋮	
⋮	
grep	0.92
ida	0.92
para	0.92
pretty	0.92
sched	0.92
wave4main	0.92
gcd	0.93
integer	0.93
wheel–sieve1	0.93
compress	0.94
compress2	0.94
exp3_8	0.94
primes	0.94
sphere	0.94
tak	0.95
fibheaps	0.96
integrate	0.96
mandel2	0.96
primetest	0.97
boyer	0.98
event	0.98
fish	0.98
rsa	0.98
x2n1	0.99
rfib	1.03
queens	1.04
scc	1.05
3 others	1.00
Min	0.36
Max	1.05
Geom ave.	0.83

表 5.15. コンパイル時間の変化 (fo/bu + hylo)

ベンチマーク	比率
banner	0.37
solid	0.38
parstof	0.47
eliza	0.56
mkhprog	0.64
transform	0.65
reptile	0.66
ansi	0.69
boyer2	0.69
cacheprof	0.69
gg	0.74
symalg	0.75
listcopy	0.76
comp_lab_zift	0.77
listcompr	0.77
multiplier	0.77
genfft	0.78
minimax	0.78
typecheck	0.78
anna	0.79
calendar	0.79
veritas	0.79
clausify	0.80
cse	0.81
maillist	0.81
expert	0.83
cichelli	0.84
circsim	0.84
pic	0.84
⋮	
fem	0.85
infer	0.86
prolog	0.86
cryptarithm2	0.87
fft	0.87
life	0.87
rewrite	0.87
simple	0.87
gen_regexps	0.88
lift	0.88
mandel	0.88
parser	0.88
puzzle	0.88
bspt	0.89
fft2	0.89
hpg	0.89
sorting	0.89
atom	0.90
wang	0.90
hidden	0.91
knights	0.91
constraints	0.92
gamteb	0.92
gcd	0.92
power	0.92
wheel–sieve1	0.92
awards	0.93
sched	0.93
compress2	0.94
fluid	0.94
⋮	
fulsom	0.94
grep	0.94
ida	0.94
para	0.94
exp3_8	0.95
integer	0.95
mandel2	0.95
paraffins	0.95
pretty	0.95
primes	0.95
sphere	0.95
treejoin	0.96
tak	0.97
boyer	0.98
cryptarithm1	0.98
fibheaps	0.98
integrate	0.98
primetest	0.98
queens	0.98
rfib	0.98
fish	0.99
event	1.01
wheel–sieve2	1.01
x2n1	1.02
compress	1.07
nucleic2	1.07
wave4main	1.25
2 others	1.00
Min	0.37
Max	1.25
Geom ave.	0.85

表 5.16. メモリ割り当ての変化 (hylo-fusion)

ベンチマーク	比率
wheel–sieve1	0.37
infer	0.70
rewrite	0.77
cryptarithm2	0.78
pic	0.82
awards	0.84
gcd	0.84
puzzle	0.87
ansi	0.88
mkhprog	0.88
transform	0.88
wave4main	0.88
lift	0.89
listcopy	0.89
circsim	0.91
eliza	0.91
typecheck	0.92
gg	0.93
gamteb	0.94
prolog	0.94
fft	0.95
genfft	0.95
simple	0.95
cacheprof	0.96
wang	0.96
⋮	
anna	0.97
fem	0.97
fibheaps	0.97
nucleic2	0.97
bspt	0.98
knights	0.98
minimax	0.98
sphere	0.98
comp_lab_zift	0.99
expert	0.99
fluid	0.99
ida	0.99
life	0.99
listcompr	0.99
mandel	0.99
multiplier	0.99
parstof	0.99
reptile	0.99
solid	0.99
44 others	1.00
Min	0.70
Max	1.00
Geom ave.	0.96

表 5.17. メモリ割り当ての変化 (foldr/build)

ベンチマーク	比率
parstof	0.04
⋮	
wave4main	0.60
mkhprog	0.69
infer	0.70
pic	0.70
rewrite	0.76
fibheaps	0.78
simple	0.79
fft	0.82
gcd	0.86
puzzle	0.87
ansi	0.88
awards	0.88
circsim	0.90
multiplier	0.91
wheel–sieve1	0.92
banner	0.93
constraints	0.93
prolog	0.93
calendar	0.94
gg	0.94
cacheprof	0.95
cichelli	0.95
cryptarithm1	0.95
expert	0.95
fem	0.95
minimax	0.95
reptile	0.95
sched	0.95
anna	0.96
bspt	0.96
event	0.96
transform	0.96
⋮	
eliza	0.97
hidden	0.97
knights	0.97
lift	0.97
sphere	0.97
comp_lab_zift	0.98
cse	0.98
fluid	0.98
gamteb	0.98
listcompr	0.98
listcopy	0.98
nucleic2	0.98
atom	0.99
cryptarithm2	0.99
fft2	0.99
genfft	0.99
hpg	0.99
ida	0.99
integrate	0.99
life	0.99
mandel	0.99
primes	0.99
solid	0.99
sorting	0.99
typecheck	0.99
veritas	0.99
x2n1	0.99
compress2	1.01
27 others	1.00
Min	0.04
Max	1.01
Geom ave.	0.91

表 5.18. メモリ割り当ての変化 (fo/bu + hylo)

ベンチマーク	比率
parstof	0.04
wheel–sieve1	0.43
wave4main	0.60
infer	0.69
mkhprog	0.69
pic	0.70
rewrite	0.72
cryptarithm2	0.78
fibheaps	0.78
simple	0.79
fft	0.82
gcd	0.86
transform	0.86
puzzle	0.87
ansi	0.88
awards	0.88
lift	0.88
listcopy	0.88
circsim	0.89
multiplier	0.91
typecheck	0.91
banner	0.93
constraints	0.93
prolog	0.93
calendar	0.94
gg	0.94
bspt	0.95
cacheprof	0.95
cichelli	0.95
cryptarithm1	0.95
expert	0.95
fem	0.95
genfft	0.95
minimax	0.95
reptile	0.95
sched	0.95
⋮	
anna	0.96
eliza	0.96
event	0.96
wang	0.96
comp_lab_zift	0.97
hidden	0.97
knights	0.97
listcompr	0.97
nucleic2	0.97
sphere	0.97
cse	0.98
fluid	0.98
gamteb	0.98
atom	0.99
fft2	0.99
hpg	0.99
ida	0.99
integrate	0.99
life	0.99
mandel	0.99
paraffins	0.99
primes	0.99
solid	0.99
sorting	0.99
veritas	0.99
x2n1	0.99
compress2	1.01
25 others	1.00
Min	0.04
Max	1.01
Geom ave.	0.90

表 5.19. 実行時間の変化 (hylo-fusion)

ベンチマーク	比率
cryptarithm2	0.65
gcd	0.77
genfft	0.86
infer	0.86
hpg	0.89
puzzle	0.89
wave4main	0.89
rewrite	0.90
circsim	0.92
fibheaps	0.92
listcopy	0.92
transform	0.92
anna	0.93
sched	0.93
boyer	0.94
gamteb	0.94
parser	0.94
clausify	0.95
listcompr	0.95
rfib	0.95
typecheck	0.95
cichelli	0.96
ida	0.96
maillist	0.96
comp_lab_zift	0.97
fem	0.97
hidden	0.97
nucleic2	0.97
wheel–sieve1	0.97
⋮	
event	0.98
fulsom	0.98
para	0.98
primes	0.98
constraints	0.99
life	0.99
treejoin	0.99
wheel–sieve2	0.99
atom	1.01
integer	1.01
mandel	1.01
primetest	1.01
symalg	1.01
compress	1.02
paraffins	1.02
simple	1.02
rsa	1.03
solid	1.03
fft2	1.04
parstof	1.06
queens	1.06
tak	1.07
x2n1	1.14
36 others	1.00
Min	0.65
Max	1.14
Geom ave.	0.98

表 5.20. 実行時間の変化 (foldr/build)

ベンチマーク	比率
parstof	0.02
fibheaps	0.65
gcd	0.66
wave4main	0.67
infer	0.79
primes	0.81
cryptarithm1	0.85
rewrite	0.85
tak	0.86
cacheprof	0.89
puzzle	0.90
circsim	0.92
event	0.92
fem	0.92
maillist	0.92
rfib	0.92
simple	0.92
anna	0.93
ida	0.93
listcompr	0.93
clausify	0.95
listcopy	0.95
transform	0.95
atom	0.96
boyer	0.96
cryptarithm2	0.96
hidden	0.96
⋮	
compress	0.97
genfft	0.97
hpg	0.97
nucleic2	0.97
sched	0.97
treejoin	0.97
constraints	0.98
wheel–sieve1	0.98
gamteb	0.99
integrate	0.99
life	0.99
multiplier	0.99
para	0.99
power	0.99
sphere	0.99
symalg	0.99
wheel–sieve2	0.99
fulsom	1.01
integer	1.01
solid	1.01
typecheck	1.01
cichelli	1.02
compress2	1.03
fft2	1.03
rsa	1.03
paraffins	1.05
wang	1.05
x2n1	1.06
33 others	1.00
Min	0.02
Max	1.06
Geom ave.	0.92

表 5.21. 実行時間の変化 (fo/bu + hylo)

ベンチマーク	比率
parstof	0.04
cryptarithm2	0.65
wave4main	0.66
fibheaps	0.70
rewrite	0.70
fem	0.77
gcd	0.80
infer	0.84
primes	0.85
cryptarithm1	0.86
genfft	0.89
cacheprof	0.90
listcopy	0.90
puzzle	0.90
queens	0.90
transform	0.90
anna	0.92
circsim	0.92
event	0.92
listcompr	0.92
maillist	0.92
simple	0.92
fft	0.93
sched	0.93
tak	0.93
boyer	0.94
hpg	0.94
multiplier	0.94
parser	0.94
⋮	
sphere	0.95
typecheck	0.95
compress	0.96
hidden	0.96
nucleic2	0.96
paraffins	0.96
wheel–sieve2	0.96
comp_lab_zift	0.97
rfib	0.97
atom	0.98
cichelli	0.98
clausify	0.98
gamteb	0.98
wang	0.98
constraints	0.99
ida	0.99
integrate	0.99
life	0.99
para	0.99
power	0.99
rsa	0.99
treejoin	0.99
wheel–sieve1	0.99
fft2	1.01
integer	1.01
mandel	1.01
primetest	1.01
compress2	1.02
solid	1.03
x2n1	1.08
29 others	1.00
Min	0.04
Max	1.08
Geom ave.	0.91

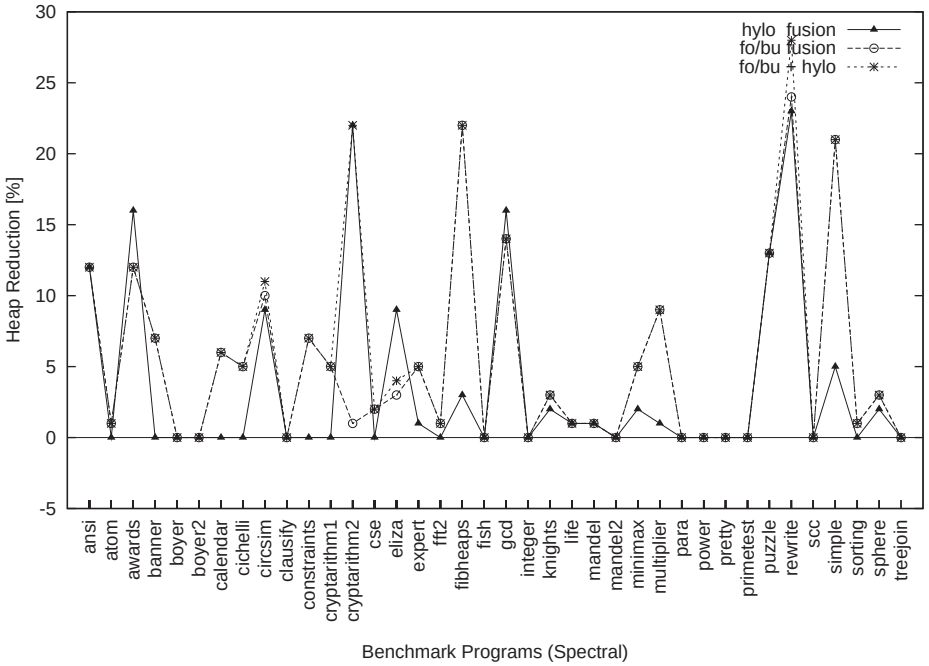


図 5.9. 各融合変換によるヒープ減少量の比較

コンパイル時

表 5.10~5.15 は，ベンチマークの各対象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，適用した融合変換の種類毎に分類して示したものである．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現わし，相対的な減少度の大きい順に並べてある．

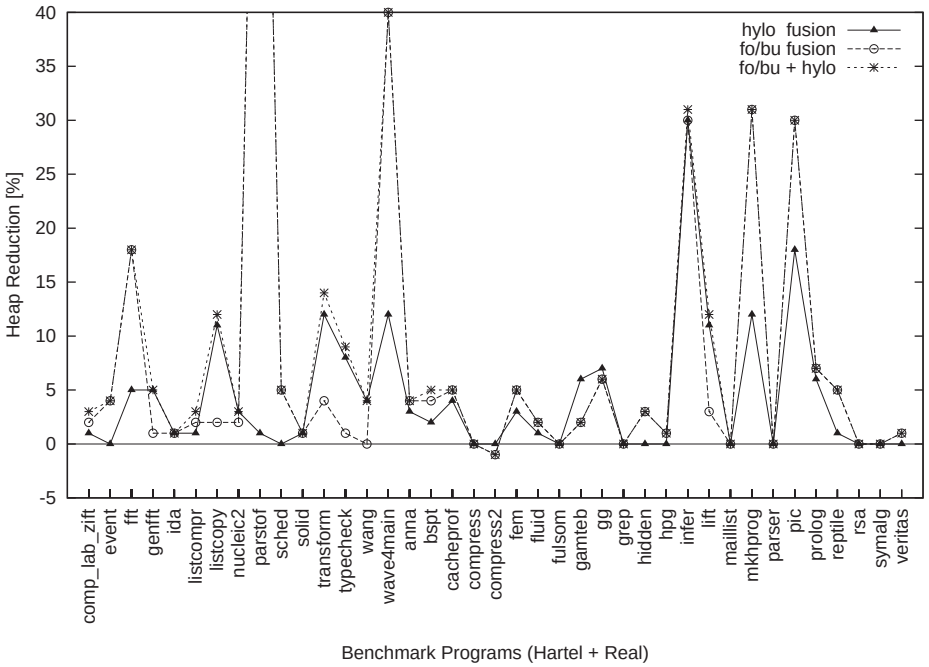


図 5.10. 各融合変換によるヒープ減少量の比較 (parstof: foldr/build, fo/bu+hylo 96%)

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．

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．

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と実行時間を調べるためのオプションを指定して実行し，それを比率に応じて並べたものが表 5.16~5.21 である．また図 5.9,5.9 は，この中でもとくにヒープ使用量に注目して，これを適用した融合変換毎に折線グラフで表示したものであり，図 5.9 が spectral ベンチマークでのヒープ使用量の変化，図 5.10 が spectral(hartel)+real ベンチマークでの変化を表わしている．図の縦軸は，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，上に伸びるほど融合変換の処理によりヒープ使用量が減少したことを示している．

図 5.10 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な中間データが生成されなくなった効果が現われている．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．

表 5.13 の中で，real の compress ベンチマークだけが最適化オプションを付けなかった場合と比べて，13.84 倍と非常に遅いコンパイル時間を示しているが，この原因については 5.3.3 節で解説する．

Hartel で勝てる理由

spectral ベンチマークの一部として含まれている Hartel ベンチマーク [HL93] は，本来関数型言語 SASL/Hope 用に記述されたものを Haskell 用に書き直したもので，本来 Haskell 言語の組み込み関数に含まれている関数をユーザ定義関数として再定義している．このため， shortcut 融合変換では，プラグマを用いて RULES が定義されていないユーザ定義の関数を融合することはできないため，融合変換後の効率が我々の実装と比べてそれほど効率が上がっていないことがわかる．

5.3.3 hylo の構成子式への適用と大域変数の展開

第 5.2.5 節でもふれたが，$[\![\phi, \eta, \psi]\!]\ (C_i\ t_1 \ldots t_n)$ のような hylo 関数を構成子式に適用した形は，融合変換の途中でよく出現する．この式は構成子 C_i の種類が判明しているため，これに応じて引数部分 $t_1 \ldots t_n$ にさらに hylo 式を適用しさらなる融合変換を行なうことが可能となる．このような特殊な場合の動作である，再帰関数の構成子式への展開を行なうか否かもコンパイラへのオプションで指定可能にしてある．そこでこの変換の有効性を調べることにした．なお hylo 変換の効果を明確にするため， shortcut 融合変換を行なわない状態，すなわち NA と CataHY で比較した．

さらに，図 3.6 における hylo 式の τ 関数の導出の際，規則 $\mathcal{F}'$ を適用する対象の式が再帰変数 v' や非再帰変数 v 以外の変数の場合，この τ 導出を続けるためには該当変数のインライン展開を行なうことによって，さらに融合変換可能な箇所を増やす処理を行なっている．このとき対象となる変数が大域変数のものまでインライン展開してしまうと，その先融合変換がさらに進むことによってより多くの不要な中間データを生成しないようにすることもあり得るが，本来複数回参照されていた大域変数の右辺式が複数回評価されることに繋が

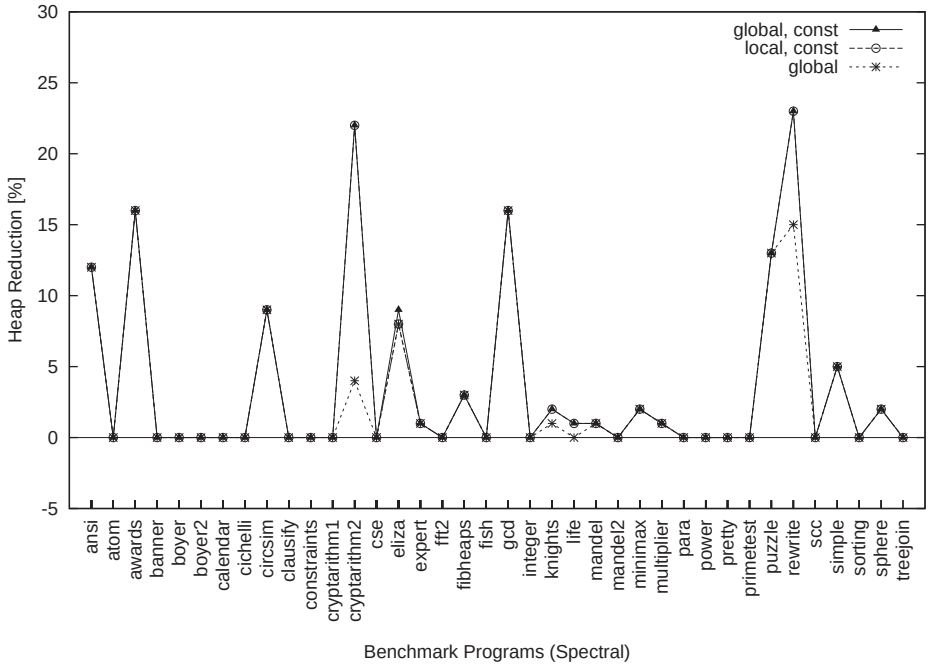


図 5.11. 構成子式への適用と大域変数の展開 (spectral)

る可能性も否定できない．従って，このインライン展開は実行効率の面からみると両刃の剣であり，この有効性も検証しておく必要がある．

そこで，a) hylo 式の構成子式への適用を展開する場合を const，b) τ 導出時の大域変数の展開を行なう場合を global，として global+const, global, const の 3 通りの場合について，ベンチマークプログラムをコンパイル実行しその効果を比較した．このヒープ使用量の減少度を図にしたものが図 5.11,5.12 である．なおこれまでに行なってきた hylo 変換は global+const の設定で行なってきた．この図からわかることは，それほど大きな差は生じなかったものの，これまで用いてきた global+const の設定が一番良い結果を導きだしているといえよう．これより global や const の処理を行なうことが不要なインライン展開に繋がり効率が悪化する場合がないことが確認された．

ただし表 5.13 の中で，real の compress ベンチマークだけが最適化オプションを付けなかった場合と比べて，13.84 倍と非常に遅いコンパイル時間を示しているが，この原因は，compress に含まれる Encode.hs ファイルの中に長さ 256 の整数の即値が入ったリストが存在し，このリストに対して後の 5.3.3 節で述べる hylo 関数の構成子式への適用処理を行なった結果，大量のインライン処理とそれに伴なうごみ集めの時間がかかったものである．その証拠に，このオプションを無効にして再度コンパイルしてみたところ，所要時間はほぼ同様の時間で収まった．したがって，このように具体的に値の定まった大規模な再帰データがプログラム中に存在したとき，そのコンパイル時間が通常よりも大幅にかかる場合があるという問題があるようである．しかし，実行時間やヒープ使用量という実行時の性質については，ほとんど悪影響を及ぼしてないことも明記しておく．

5.3.4 hylo-fusion と foldr/build の順序入れ換え

第 4.2.2 節で見たように，GHC の内部では一連の最適化処理の流れが各プログラム変換をフラグによって組み合わせる形で行なわれている．そこで我々の融合変換もオプションとして '-fhylo-fusion-on' をコンパイラに指定すると，内部で対応するフラグが立ち，融合変換の処理が行なわれることになっている．一方 shortcut 融合変換は，コンパイラ内において既定の動作として組み込まれて実装されているため，その挙動を明示的に無効にする手法は用意されていない．また shortcut 融合変換は，単一のプログラム変換としてではなく，最

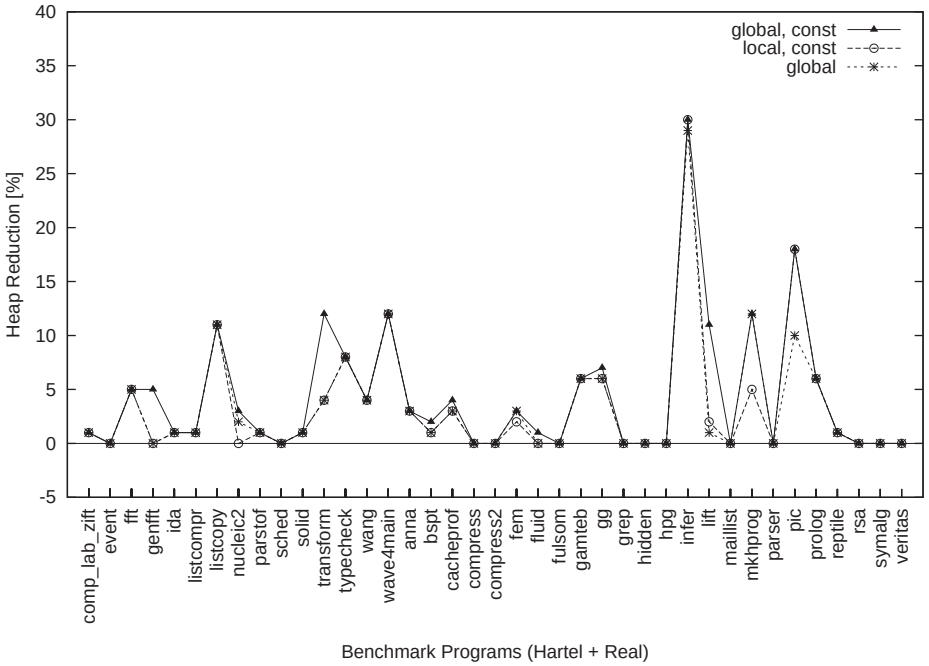


図 5.12. 構成子式への適用と大域変数の展開 (hartel + real)

適化の中で何度も呼び出される簡略化変換 (simplifier) の中で段階的に行なわれるように実装されている．

そこで我々の融合変換はと shortcut 融合変換の順序を入れ換えた上で両方の最適化を有効にすることによって，プログラムの実行効率がどのように変化するかを調べてみた．GHC に対して最適化オプション `-O` を付けた場合に行なわれる最適化処理の流れは，図 5.13 に示すようになっているが，この中で本来 (B) の位置で行なっていた hylo 変換の処理を，新たに (A) の位置で行なうように変更した．これにより，`Simplify1` の位置で行なわれる shortcut 融合変換より前に hylo 変換を行なうことができる．

図 5.14,5.15 に，hylo 変換を shortcut 融合変換 より先に行なった場合のヒープ使用量の減少度を示す．いずれの図も，各融合変換処理を行なわなかった場合を基にしている．これらの図からわかることは，多くの例において shortcut 融合変換を先に行なったほうが効率の良くなるケースが多いことがわかった．これは hylo 融合変換を shortcut 融合変換より先に行なう場合，組み込み関数に対する融合変換が行なわれないことが原因である．

組み込み関数の融合変換は以下のような流れで行なわれる．

```
map f . map g
        ⇓
build (\ c n -> ... mapFB (mapFB c f) g ...)
        ⇓
build (\ c n -> ... mapFB c (f.g) ...)
        ⇓
mapList (f.g)
        ⇓
let { go = \ xs -> ... go ... } in go
```

この一連の流れの中で shortcut 融合変換を適用することにより，関数 `map` の関数合成が 2 つの `mapFB` で表わされる．foldr/build の書き換え規則によって単一の `mapFB` に置き換えられた後，最終的に `map` の実体をもつ関数で再帰的に定義された `mapList` に帰着される．この `mapList` も，foldr/build 後のいくつかの変換

```
Simplify0
Specialising
Float outwards
Float inwards
(A)
Simplify1        <foldr/build>
Simplify2
Simplify3
DemandAnalysis
WorkerWrapper
GlomBinds
Simplify
Float out
Common sub-expression
Float inwards
(B)                <Simplify + HyloFusion>
Simplify
Tidy Core
```

図 5.13. Core 言語上の最適化処理の一覧

によりインライン展開され，その再帰的定義 go がプログラム中に含まれるようになる．一方我々の hylo 変換は，組み込み関数に対して特別な処理を行なわない．すなわちライブラリとして用意されている関数には手を加えなくてよい反面，その関数定義の実体がないと hylo 式が導出できないため融合変換を行なうことはできない．従って，組み込み関数に対する融合変換は foldr/build 変換を適用することによって関数定義がプログラム中に現われた後でないと，hylo 変換を適用することができないことになる．

また図 5.15 の pic のように hylo 変換を先に行なったことにより，何もしなかった場合に比べ効率が悪くなる場合も見受けられた．これは，hylo 変換の実装する際に，foldr/build 変換終了後の Core 言語プログラムを入力として想定し，変換時にどの程度まで変数のインライン展開を行なうかなどのチューニングをしたことが原因の一つであると考えられる．

5.3.5 実装上の注意点

今回の実験を行なったところ，いくつかのベンチマークプログラムにおいて融合変換を行なったところヒープ使用量が増える例が見受けられた．これは融合変換の目的に反するものであり，我々の実装に問題があるものと考えられるためその原因を追及したところ，以下の二点に関して問題が見つかった．以下の節では，これらの問題が起こった背景と，それに対する対応策を説明することにする．

頻度解析の必要性

あるベンチマークの例で

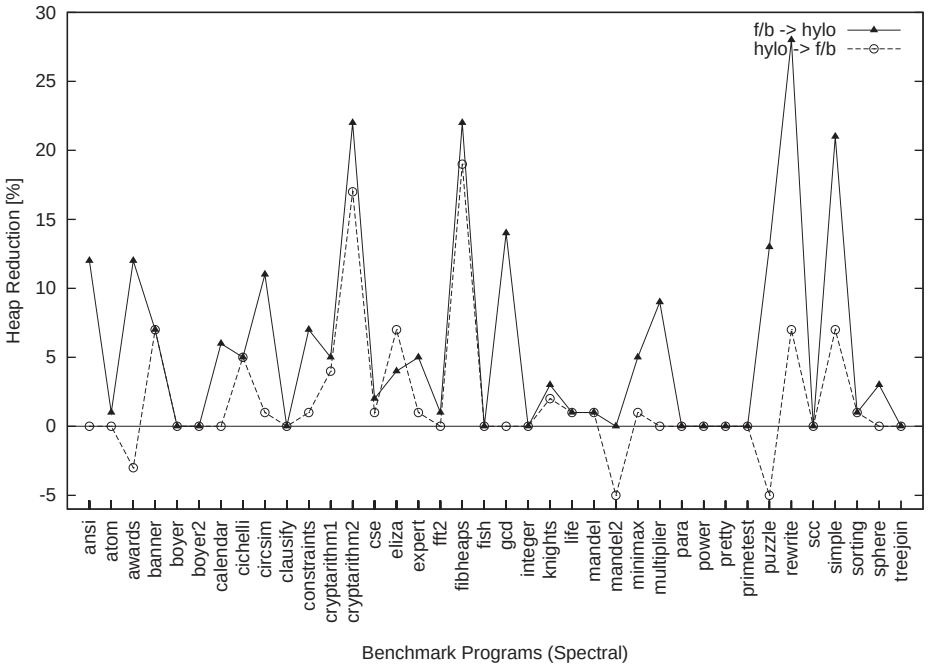


図 5.14. 融合変換の順序の入れ替え

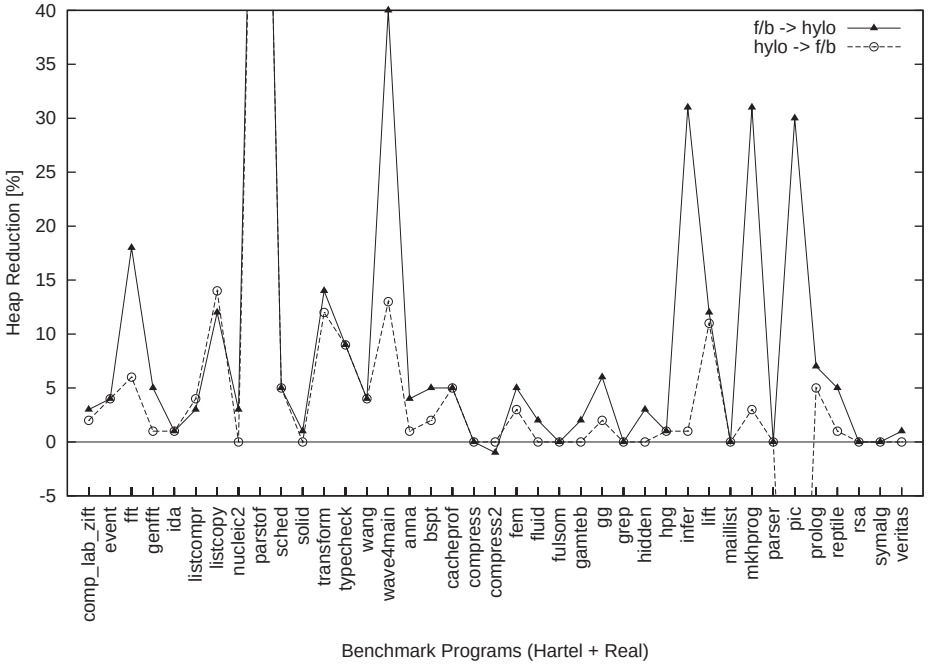


図 5.15. 融合変換の順序の入れ替え (parstof: fb->hy 96%, hy->fb 95%, pic: hy->fb -43%)

不用意な let 束縛の外部への移動

当初我々の実装では，再帰関数から Hylo を導出する際に，let 式に対して以下のような最適化を行なうことによって，無駄な変数間の let 束縛を作らないようにしていた．

$$
\begin{array}{lcl}
\mathcal{D}[\![\text{let } v = t_1 \text{ in } t_0]\!]\, s_l\, f & = & (s_0 \cup s_1, c_0 \cup c_1, \text{let } v = t_1' \text{ in } t_0') \\
 & & \text{where } (s_1, c_1, t_1') = \mathcal{D}[\![t_1]\!]\, s_l\, f, (s_0, c_0, t_0') = \mathcal{D}[\![t_0]\!]\, (s_l \cup \{v\})\, f \\
 & \Downarrow & \\
\mathcal{D}[\![\text{let } v = t_1 \text{ in } t_0]\!]\, s_l\, f & = & \text{if } c_1 = \{(u, e)\} \ \wedge \ t_1' = u \\
 & & \quad \text{then } (s_0 \cup s_1, c_0 \cup c_1, t_0'[u/v]) \\
 & & \quad \text{else } \ (s_0 \cup s_1, c_0 \cup c_1, \text{let } v = t_1' \text{ in } t_0') \\
 & & \text{where } (s_1, c_1, t_1') = \mathcal{D}[\![t_1]\!]\, s_l\, f, (s_0, c_0, t_0') = \mathcal{D}[\![t_0]\!]\, (s_l \cup \{v\})\, f
\end{array}
$$

このように変換規則を変更することで，let 束縛の右辺式が変数であった場合，変数に変数を割り当てるような束縛を作ることを防いでいた．これは変数に変数を割り当てることは無駄だと思えたからであるが，実際にベンチマークプログラムに対して評価実験を行なったところ，この変更後の規則では問題が発生することがわかった．例えば，以下の例である．

第 6 章

関連研究

中間データとして受け渡しされるデータ構造を除去するための手法は，古くは [BD77] から現在に至るまで，様々な手法が提案されている．この中には全自動で除去する試みのほか，プログラマが何らかの指示を与えることによって，不要なデータ構造の削除を行なうものも存在する．

ここではそれらをサーベイし，とくに現在もっとも注目を浴びている手法を中心に説明する．さらに，この手法と我々の採用した融合変換の手法との関係を明確にし，そのいくつかは型推論に基づく解析によって向上することを示す．

一般的にいって，融合変換の手法は大きくわけて二種類に分類される．一つは unfold/fold に基づいた手法で，もう一つは shortcut 融合変換のように一般的な再帰定義関数を扱うことを諦め，その中である” 定まった“再帰形式で表わせるものを対象に融合変換を行なおうとするものである．この後者の手法では，パラメータ性定理 (parametricity theorem) に基づいた融合規則を用いることで融合変換を行なう．(olaf) 筆者らは，再帰の” 定まった” 形式を得る方法に関して異なる観点に着目する．これは，プログラマによって提供される必要があったり，他のプログラム変換によって任意の再帰プログラムから自動的に推論される必要があったりしていた．属性文法と関連する定式化とを結合する手法は，上記二つの分類の中間に位置するものといえよう．しかし，これらは定式化のための制限された再帰表現に依存しているため，unfold/fold のような任意の再帰形式を扱えるものよりも，” 定まった” 形式を扱う shortcut 融合変換の方により近いということができよう．

6.1 Unfold/Fold 融合変換 (探索型)

6.1.1 Unfold/Fold 変換

unfold/fold による変換手法は [BD77] によって開発されたもので，関数プログラムを最適化するために開発された多くのプログラム変換の技術に基づいている．

この手法は，等式集合に適用される 6 つの基本規則から巧みに適用することによって構成される．最初の等式の集合は変換の対象となるプログラムで，変換が発生するようにいくつかの等式が追加されているかもしれない．各等式は，左辺に $v(e_1, \ldots, e_n)$ の形式をもち，右辺に任意の式をもつ．これは多くの関数型言語にみられるパターンマッチに相当する．規則はプログラムが以下に示す条件に適合するとき，それに適用されることによって意味を変えない新しいプログラムを生成する．

1. *Definition.* 新しい等式が導入され，左辺は集合にすでに存在する他の等式のいずれかとも重なることはない．
2. *Instantation.* 既にある等式に対して置換を行ない，新しい等式を導入する．
3. *Unfolding.* 等式 $e = e'$ と $f = f'$ が存在し，f' が e の置換インスタンス $S\ e$ を持つ場合，それを $S\ e'$ で置換する．
4. *Folding.* 等式 $e = e'$ と $f = f'$ が存在し，f' が e の置換インスタンス $S\ e'$ を持つ場合，それを $S\ e$

で置換する．

5. *Abstraction.* 任意の等式 $e = e'$ に対して **where** 節を導入し，その右辺を以下で置換する．

$$e'[x_1/f_1 \ldots x_n/f_n]\text{where } (x_1, \ldots, x_n) = (f_1, \ldots, f_n)$$

6. *Laws.* 式で用いられているプリミティブに関する法則を用いることができる．例えば，連結性，可換性，など

ここで実際に unfold/fold 変換の例をみることにしよう．よく利用されるようなプログラムの断片として以下の例を選んだ．ある述語をみたすような列の初期セグメントの長さである．以下の式がこれを求めるための関数となっている．

$$length\ (takewhile\ p\ (iterate\ f\ 0))$$

これは問題を，三つの部分問題に分割し，それらを組み合わせることによって解決している．関数 *length* はリストを受け取り，その長さを返す．関数 *takewhile* はリストを受け取り，その値がすべて述語 p をみたすような先頭からのリストを返す．関数 *iterate* は初期値に対して関数を繰り返し適用することによって得られるリストを生成する．ここで組み合わせ関数の定義を以下に示す．

$$iterate\ f\ x = x : iterate\ f\ (f\ x) \tag{6.1}$$
$$takewhile\ p\ (x : xs) = \text{if } (p\ x) \text{ then } (x : takewhile\ p\ xs) \text{ else } [] \tag{6.2}$$
$$length\ [] = 0 \tag{6.3}$$
$$length\ (x : xs) = 1 + length\ xs \tag{6.4}$$
$$\tag{6.5}$$

上のプログラムはプログラマの意思を明白に表わしている一方，問題点として，関数合成の際に二つの中間的なデータ構造を生成してしまう．*iterate* により生成されるリストは *takewhile* によって消費され，次に *takewhile* の結果は *length* によって消費される．我々の変換の目的は，このような中間的なリストを用いることなく，同様の計算を実行することである．

図 6.1 は変換と望みの結果を得るまでの過程が示されている．まず元の式に等しい関数 g を定義し，以下の各ステップを順に適用する．

1. 新しい関数 h の定義: g と等価だが，*iterate* の第二引数が抽象化したものとする．これが，この変換過程における**ひらめき** *(eureka)* である．
2. 元の式を h に関して fold する．これでプログラムは新しい関数 h への単一呼び出しによって表わされるようになった．
3. h の右辺式における *iterate* 呼び出しを unfold する
4. h の右辺式における *takewhile* 呼び出しを unfold する
5. 以下の法則を用いる

$$f(\text{if } e \text{ then } e_1 \text{ else } e_2) \rightarrow \text{if } e \text{ then } fe_1 \text{ else } fe_2$$

   この法則は関数 f が正格 (strict)，すなわち $f\top = \top$ のとき成り立つ．関数 *length* は正格なのでこの法則を用いて，*length* の呼び出しを if の分岐の内部に移動することができる．
6. *length* の各呼び出しを unfold する
7. これで，関数 h の元の右辺のインスタンスが現われたことになり，fold 規則を適用することができる．

このプログラムの断片は，元の式では関数結合で表現されていたものと同等の再帰関数へと変換された．一般的な関数結合を用いるのに比べ，このプログラムはその目的を行なうために特殊化され，その結果として不要な中間データ構造を作らずに以前よりメモリ使用量が少なくなった．このことは実際にプログラムを実行させたときにその効率の改善となってみることができる．

```
                g f p   = length (takewhile p (iterate f 0))
{新しい h の定義}
                h f p x = length (takewhile p (iterate f x))
{h の fold}
                g f p   = h f p 0
{iterate の unfold}
                h f p x = length (takewhile p (x : iterate f (f x)))
{takewhile の unfold}
                        = length (if (p x) then (x : iterate f (f x))
                                           else [])
{if の法則}
                        = if (p x) then length (x : takewhile p (iterate f (f x)))
                                   else length []
{length の unfold}
                        = if (p x) then 1 + length (takewhile p (iterate f (f x)))
                                   else 0
{h の fold}
                        = if (p x) then 1 + h f p (f x)
                                   else 0
```

図 6.1. unfold/fold による変換過程

これらの 6 つの法則は任意のプログラムを他のプログラムへ変換するのに十分一般的といえるが，しかし任意のプログラム変換を行なえるわけではない．なぜなら以下のような関数を

$$f([]) = 0 \tag{6.6}$$
$$f(x : xs) = f(xs) \tag{6.7}$$
$$\tag{6.8}$$

以下のように変換できないからである．

$$f(xs) = 0 \tag{6.9}$$

しかし**再定義**規則を追加すればこのような変換も可能になる，という報告もある．

ここで注意しなければならないのは，unfold/fold 変換は部分的にのみ正しいということである．これにより，変換の結果が結果を生成するようなプログラムなら，この結果は元のプログラムが生成した結果と同じであるだろう．不幸なことに，変換によって元のプログラムより less defined なプログラムが生成されてしまうこともあり得る．これは，任意回 fold のステップを適用してしまった場合に起こる．例えば，等式 $f(n) = e$ が集合中に存在した場合，fold 規則の適用によってこれが $f(n) = f(n)$ に置換されてしまう．これは明らかに停止しない関数になっている．何人かの人がこれの解決策を提案した．Kott は，変換中のある条件下で，unfold のステップ数が常に fold のステップ数以上なら正当性が保証されることを示した．Scherlis は全体の正当性を保証するため fold のステップを制限する案を示した．

Sands は一般的な改良のための定理を提案した．この定理は，高階の unfold/fold に基づいたプログラム変換手法に関して正当性を保証している．彼の定理は特定の変換が厳密にプログラムを改善していることを検証するために用いることができる．

この手法の一般性は，以下のことを意味する．それは，プログラム変換によって，より効率的なプログラムが得られることがある反面，効率が落ちてしまうプログラムが生成されることもあり得るということである．

$$
\begin{aligned}
tt = {} & x \\
| \ & f\ x_1 \ldots\ x_n \\
| \ & C\ tt_1 \ldots\ tt_n \\
| \ & \text{case } tt \text{ of } \{p_1 \rightarrow tt_1\ ; \ldots ;\ p_n \rightarrow tt_n\}
\end{aligned}
$$

図 6.2. 森林伐採のツリーレス形式

これは，有利なプログラムを得るには，変換ステップがユーザやよく定義された戦略集合 (**戦術** *(tactics)*) によって導かれる必要がある．

Burstall と Darlington によって示された例は，ある意味ユーザによって導かれたものであり，望みの結果を得るための変換過程を示すような中心となる *eureka* ステップを含んでいた．いくつかの eureka ステップは達成する最適化の種類に応じて分類され，この事実が変換戦略を特定し自動化するものであった．これは多くの提案された変換技術 [Chi90] に基づいている．Feather は，大きなシステムの変換を容易にするためにどのように unfold/fold 変換手法を半自動化するかを示している．[Fea79,Fea82]

Burstall と Darlington は多くの変換が同様の形式に従うと認識し，これを規則適用の**戦略**と提案した．これは以下のようになる．必要な定義を用意し，実体化 (instantiate) し，法則や抽象化を適用している間または folding している間，繰り返し unfold する．この戦略によると例えば二乗のオーダの *nfib* 関数が，線型時間の等価な関数に変換されることが示されている．またあるリスト処理関数を中間データ構造を生成しないように変換できる．

6.1.2 森林伐採

Wadler の森林伐採 (deforestation) のアルゴリズム [Wad88] は，彼のリストレス性 (listlessness) の研究から発展したものである．これは前節の unfold/fold 戦略に基づいた変換システムで，関数プログラムから中間データ構造 (ここでは一般的な木を仮定) を自動的に除去するものである．これは，unfold/fold 規則を適用するための戦略としてみることも可能で，自動的に実行可能な単独の変換アルゴリズムのパッケージとして扱うことができる．

Deforestation は，あるツリーレス形式 (treeless form，図 6.2) で記述された関数で組み合わされた第一階の関数プログラムを変換の対象とする．名前が示すように，ツリーレス形式の関数は，中間的なデータ構造 (すなわち，木) を生成しないことが保証されている．ツリーレス形式の関数に対するもう一つの制限は，関数の引数は線型でなければならない．すなわち関数の本体上で 2 回以上参照される引数は存在しない．

Deforestation のアルゴリズムは，ツリーレス関数 (本体がツリーレス形式で定義されたもの) とツリーレスでない項から構成されるプログラムを入力とし，ツリーレス項のみからなるプログラムを出力するものである．これにより，ツリーレス関数への呼び出しだけを含んでいる式からすべての中間データ構造を除去することができる．

このアルゴリズムを概説すると以下のようになる．図 6.3 に示されている変換関数は，関数の最初の結合に対して適用される．そして，簡約を適用し転換の変換を交換している (?) 間，関数呼び出しを連続的に unfold する．部分式が現われそれが以前に変換した式の名前を変更したものである場合はいつでも，fold ステップが引き起こされる．fold ステップは新しい再帰関数を生成し，それはツリーレス形式であることが保証されている．アルゴリズムは変換されるべき部分式がなくなるまで繰り返され，最終的にツリーレス形式の式とツリーレス関数の新しい集合が得られる．ツリーレス形式の定義によって，結果のプログラムは中間データ構造を含んではいないことを意味する．

$$
\begin{aligned}
\mathcal{T}\ x &= x \\
\mathcal{T}\ (C\ t_1 \ldots t_n) &= C(\mathcal{T}\ t_1) \ldots (\mathcal{T}\ t_n) \\
\mathcal{T}\ (f\ t_1 \ldots t_n) &= \mathcal{T}\ (t[t_1/x_1, \ldots, t_n/x_n]) \\
&\quad \text{where } f \text{ is defined by } f\ x_1 \ldots x_n = t \\
\mathcal{T}\ (\text{case } x \text{ of } \{p_1 \to t_1; \ldots; p_n \to t_n\}) & \\
&= \text{case } x \text{ of } \{p_1 \to \mathcal{T}\ t_1; \ldots; p_n \to \mathcal{T}\ t_n\} \\
\mathcal{T}\ (\text{case } x \text{ of } \{p_1 \to t_1; \ldots; p_n \to t_n\}) & \\
&= \text{case } x \text{ of } \{p_1 \to \mathcal{T}\ t_1; \ldots; p_n \to \mathcal{T}\ t_n\} \\
\mathcal{T}\ (\text{case } (f\ t_1 \ldots t_n) \text{ of } \{p_1 \to t'_1; \ldots; p_n \to t'_n\}) & \\
&= \text{case } t[t_1/x_1, \ldots, t_n/x_n] \text{ of } \{p_1 \to \mathcal{T}\ t'_1; \ldots; p_n \to \mathcal{T}\ t'_n\} \\
&\quad \text{where } f \text{ is defined by } f\ x_1 \ldots x_n = t \\
\mathcal{T}\ (\text{case } (C\ t_1 \ldots t_n) \text{ of } \{\ldots; C\ x_1 \ldots x_n \to t; \ldots\}) & \\
&= \mathcal{T}\ (t[t_1/x_1, \ldots, t_n/x_n]) \\
\mathcal{T}\ (\text{case } (\text{case } t_0 \text{ of } \{p'_1 \to t'_1; \ldots; p'_n \to t'_n\}) \text{ of } \{p_1 \to t_1; \ldots; p_n \to t_n\}) & \\
&= \mathcal{T}\ (\text{case } t_0 \text{ of} \\
&\qquad p'_1 \to \text{case } t'_1 \text{ of } \{p_1 \to t_1; \ldots p_n \to t_n\} \\
&\qquad \vdots \\
&\qquad p'_n \to \text{case } t'_n \text{ of } \{p_1 \to t_1; \ldots p_n \to t_n\})
\end{aligned}
$$

図 6.3. Deforestation

このアルゴリズムの停止性は，たんに fold ステップがやがて引き起こされるであろうことにのみ基づいている．ツリーレス形式はこのように定式化されているので fold ステップは必ず起き，Wadler は彼の論文で停止性の証明の概要を示している．停止性に関するより詳しい証明は [FW88] を参照するとよい．

不幸なことにツリーレス形式は，その形式について表現力についてもとても制限の厳しい言語である．また，最近の多くの近代的な関数型言語が高階であるのに対し，第一階に制限されてもいる．他の制限が二つあり，一つは関数適用式の引数は変数に限られる点，もう一つは関数の引数は線型でなければならない点で，すなわちツリーレス形式では任意の関数を書くことはできないことになる．

Wadler はこの制限と取り除くために，論文中に以下の二つの手法を用いた．

- **目印付け** *(blazing)* は引数が変数だけという制限を緩和するため，引数の位置に数値 (または非再帰) 型の式を許す．これらの式は簡単な結果のみ生成し，中間データ構造としてはみなされないほど小さなものである．基本アルゴリズムの拡張である目印付き融合変換 (blazed deforestation) アルゴリズムでは，$\oplus$ 印が付けられた式はそれ以上変換されることはなく，$\ominus$ 印が付けられた式は通常通りさらに変換が進められる．目印付けは，deforestation への入力として利用可能な関数の範囲を広げるが，しかしすべての関数をカバーするものではない．
- 有名な技法である**高階マクロ**を用いれば，引数が関数の型である関数を，問題となる引数を抽象化することによってマクロとして表現することが可能になる．これにより *map* のような高階関数が第一階のツリーレス形式で表わせる．

すべての関数の引数は線型でなければならない，という残りの制限は，deforestation のアルゴリズムが元のプログラムより効率の劣るものを生成しないようにするのを保証するためにまだ存在している．これは unfold の段階で発生し，関数適用がその関数の定義本体のインスタンスで置き換えるときに起こる問題である．もし引数が関数本体の中に複数回現われることが許されていたとすると，unfolding の結果として式の複製が起き

てしまい，その結果，大きな計算が実行時に勝手に複数回行なわれてしまうことになる．

さらに，この森林伐採を高階関数に適用できるよう拡張したのが [Mar95] である．

6.1.3 Supercompiler

[Tur86, Tur88] では Supercompiler では，プログラムを記号として評価することによって操作し，状態図をグラフとして構築し最終的に効率的なプログラムを導出する試みが示されている．

6.2 Cheap Deforestation (foldr/build 型)

6.2.1 shortcut 融合変換

shortcut 融合変換は Andrew Gill らによって開発された融合変換の手法で，cheap deforestation と呼ばれることもある [GLJ93, Gil96]． shortcut 融合変換の基本的な考え方は，構成子である (:) と [] を消費者である foldr によってコンパイル時に置換することにある．

特別な関数 build は，生産側の目印としてなる一方，foldr/build 規則を適用する際に必要な型の条件を明確にする役割も担う．

```
build :: (forall b . (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

この関数 build を用いて，foldr/build 規則は以下のように記述される．

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

コンパイラを作成する際は，標準ライブラリに含まれるすべてのリスト操作関数を build や foldr を用いて記述しないと，その関数に対して融合変換を行なうことはできない．

Gill は GHC の中に shortcut 融合変換を実装し，10 クィーン問題で 43%，大規模なベンチマークに対しては平均で 3%，融合変換を行なわない場合に対して実行時の効率が改善することを示した．

6.2.2 build の制限

shortcut 融合変換の持つ短所として，ライブラリに含まれる標準的なリスト操作関数を予め build を用いて定義しておかなければならないが，これはまた別の問題を抱えている．

リストの連結演算である (++) を考えよう．この連結演算は構成子を抽象化する際に注意が必要となる．具体的には，式 M_1++M_2 の中で (++) が最終的なリスト全体を構成しているわけではなく，式 M_1 によって生成されるリストのみがコピーされるからである．したがって関数 (++) は build で表現することはできない．この問題を解決するために，Gill は build を拡張した二階の型付き関数 augment を導入した．

```
augment :: (forall b. (a -> b -> b) -> b -> b) -> [a] -> [a]
augment g xs = g (:) xs
```

これにより foldr/build 規則に対応する，新たな foldr/augment 規則が以下のように定義される．

```
foldr k z (augment g xs) = g k (foldr k z xs)
```

さらにこの関数 augment を用いて，先述の (++) は以下のように書き直される．

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = augment (\ c n -> foldr c n xs) ys
```

これらの定義を用いることによって，(++) で生成されたリストに対する融合変換が可能になる．さらに関数 build は augment を用いて以下のように定義することも可能である．

```
build :: (forall b. (a -> b -> b) -> b -> b) ->[a]
build g = augment g []
```

これを見ると foldr/build 規則は foldr/augment の変数 xs を [] に具体化したものであることがわかるであろう．これより，実際によく用いられる多くのリスト生成関数は augment を用いて定義される．

6.2.3 Warm Fusion

fold プロモーション則についての説明

fold プロモーション則と shortcut 融合変換の両者の利点を活かしたものとして，Launchbury と Sheard が提案した Warm Fusion が挙げられる [LS95]．Warm Fusion では，fold プロモーション則を明示的な再帰形へ一般化することにより，そこから foldr や build を自動的に導出する試みである．

この変換は shortcut 融合変換と違い，除去の対象となるデータ型がリストに制限されてはおらず，さらに foldr/build 規則を一般的にした規則を用いている．任意のリスト生成式 M を build 形式に変換するための基本的な考え方は，まず M を

```
build (
 c n -> foldr c n M)
```

のように変換し，foldr を M の内側に入れていくことによって，M の内部にある build と相互に打ち消し合わそうとするものである．もし打ち消すことができれば融合変換のために build 形式が用いられ，できなければ上記の変換は破棄される．これと同様にリストの消費関数に対しては，関数の定義が

```
foo xs = M[ xs ]
```

だと仮定すると，これを foldr を用いた

```
foo xs = M[ foldr (:) [] xs ]
```

のように書き換えた上で，関数定義である M と foldr の融合を試みるというものである．このような foldr の内や外への移動というのはすべて，項書換えを用いて行なわれ，消費関数を foldr 形式へ変換する場合には書き換え規則が動的に増減することになる．消費関数に関する後者の変換は catamorphism に対する融合定理に基づいていて，これによりリスト以外の自然な再帰構造にも対応可能となっている．しかし，Warm Fusion はまだ負担の重い処理であり，実装にあたっては実質的な問題があることも報告されている [NJ98].

6.2.4 GHC における foldr/build の実装

GHC は 4.06 版から，worker/wrapper 手法を用いない shortcut 融合変換を内部に実装している．この変換を実装する際の問題点は，foldr や build 形式は融合変換を可能にするためには役立つが，効率の上では非常に悪いものであることである．もし foldr や build が融合変換に用いられなかった場合は，それらは必ずインライン展開されなければならない．そのために増やされた書き換え規則と二段階に及ぶインライン処理によって，GHC はこの問題を解決している．

6.2.5 型推論による自動化手法

Olaf Chitil は，これまでの shortcut 融合変換で問題であった前処理の部分を自動化する手法を提案した [Chi99, Chi00]．型推論を用いた手法では，本来の手法ではリストの生産する関数をあらかじめ決められた `build` 形式で用意しておく必要があったのに対し，これを型推論で自動的に行なうようにしたものである．

Olaf Chitil [Chi99, Chi00] らは，多くの代数的データ型に対して `build` 形式を導出するために，型推論がどのように利用できるかを示した．彼は Fegaras [Feg96] の Wadler の自由定理 [Wad89] を利用に関する研究が生産者と消費者形式の導出になることを言及した．この 2 つのシステムに Hylo を加えると，これらはみな統一した問題として扱うことができるようになる．

6.3 Hylo Fusion (カテゴリ型)

[Jon95] では，関数型言語の型システムとして多く用いられている Hindley/Milner の型推論アルゴリズムを拡張して，型のオーバーロードや高階の多様型を扱えるような型システムを提案している．これによって，Cata や Hylo のような抽象的構造を Haskell のような言語で記述することが可能になっている．

従来の融合変換では，複数の再帰データ構造を同時に扱うような相互再帰関数に対して Hylo を導出することはできず，そのため `zip` のような利用頻度の高いリスト操作関数でも融合することができないでいた．そこで [IHT98] において，Iwasaki らは相互再帰に定義された関数から Hylo を導出し， Hylo に関する定理をそれらの上においても適用可能なように拡張した．従来の拡張に対する試みと異なる点は，単一再帰な Hylo の定理を自然に拡張した点であり，論文の中で複数の再帰的なデータ構造を遷移する様子を新しい拡張記法を用いて表わしている．基本的に元となる酸性雨定理は変更する必要がないため，この記法はより強力なものになっているといえよう．

[THT98] には，近年の構成的アルゴリズム論に基づくプログラム変換の研究成果がサーベイされているとともに，Hylo はそれらの様々な変換を行なう上で重要な役割を担うものとしている．例えば [HTC98] では並列プログラムの自動生成について述べられている．

さらに hylo 導出のアルゴリズムでは，内部で再帰関数の適用式を検出する際に共通部分式の削除 (common sub-expression elimination) と同等の効果をもつ処理を行なっていることになる．これは例えば，3.5.1 節の終わりで触れられている関数 $mkTree$ から hylo を導出する際に，本来は二つの式 $mkTreeas$ に対応して関数呼び出しが二回行なわれるところが，導出後は共有されたことにより一回の呼び出しで済むようになっている．これに対し，共通部分式の削除は複数回の計算を省略することができる長所がある反面，空間使用量とそれに伴なうごみ集めの時間を増加される可能性があることも示されている [Chi88]．

[Tüf98] でも GHC を拡張し，Hylo を扱えるようにする試みが行なわれている．ここでも我々の手法と同様に，プログラム変換が可能なように Hylo を用意し，融合変換を行なう処理をコンパイラに組み込んでいる．論文中では，実用されている Haskell プログラムの関数に対し，その多くからプログラム変換に適した Hylo が導出できることが示されている．

[Sch00] では pH という並列型コンパイラに，融合変換を実装する試みが起こなわれている．対象となるのは Haskell を基にし並列性を内包した言語で，その言語上において Hylo を導出，再構成，インライン展開，τ, σ の導出と融合などのアルゴリズムが示されている．

[Jür00] では，酸性雨定理を証明する新しいアプローチが示されている．また具象関手 (concrete functor) の概念を用いることによって，これを拡張している．Haskell の Core 言語に近い拡張 λ 計算に対し，カテゴリ論を用いた表示的意味論を導入し，これによって彼らが拡張した酸性雨定理をこの言語に適用可能にし既存のものよりも強力な融合変換を行なえるようにしている．

Hylo に関する定理を拡張することによって，より一般的な形式として利用する試みも行なわれている．

Pardo [Par01] は，再帰関数の融合をより効率的に行なえるように工夫した．これは，*monadic unfold* と *monadic hylomorphism* と呼ぶスキームを用い，それらの新しい形式に対する酸性雨定理を導入している．Pardo は，新しい酸性雨定理は従来の融合変換も行なうことが可能で，純粋な関数としての Hylo は，*monadic hylomorphism* が恒等モナドを要素として構成された特別な場合であるとしている．

また，hylo の一種である anamorphism に注目し，その一つである関数 `unfold` を活用する試みが [GJ98] で行なわれている．これは catamorphism である fold が関数プログラミングでは頻繁に用いられるのに対し，その双対である unfold が殆ど利用されていない現状に着目し，木に対する幅優先探索のアルゴリズムが，fold を用いた定義では簡潔ではあるものの効率が低く，これを unfold を用いた定義に書き換えることによって線形のアルゴリズムが導出できることを示した．この中で関数 unfold を用いた定義を導出する際に，unfold と fold の二つの hylo をうまく融合することが必要で，これにより融合変換の実現が新しいアルゴリズムの導出にも有用であることが示されている．

6.4 属性文法の融合

属性文法 (Attribute Grammer) とは，文脈自由文法の構文木上で意味の計算 (属性評価) が行なえるようにした体系である．これは非終端記号に属性を付加するのと，各文法におけるプロダクションに対し意味規則を付加することによって行なわれる．属性文法は本来，コンパイラの設計や実装の際に用いられることが多かったが [ASU86]，その関数型言語に対する類似性と伴なって，最近では関数プログラミングのパラダイムとして扱われることも多くなっている [Joh87]．これにより関数型言語と同様に，二つの属性文法を，その二つを組み合わせたのと同じ機能をもつ一つの属性文法に変換することが可能かどうか，という質問が浮かび上がってくる．そのような結合を可能にするような属性文法中の一定のクラスを求める研究が行なわれてきたが，[CDPR97] ではこれを関数型言語における各種の融合変換とともに比較することを行なっている．さらに [CDPR99] では，どのようにして関数型言語を属性文法へ変換し，それを融合した上で，また関数型言語へ再変換するかについて記述されている．

これらの結合手法は，定式化の際に用いられる再帰形式に何らかの制限を加えることによって成り立っている．したがって，その手法が任意の関数プログラムに一般化可能か否か，また標準的な手法を向上させるための提案となるか否かは調査する必要があるであろう．

第 7 章

結論

本論文で，我々は構成的アルゴリズム論に基づいた運算的融合変換のアルゴリズムは，実在する関数型言語のコンパイラに対しても十分に利用可能であることを示した．我々の貢献は以下の 2 点にまとめることができる．

- 酸性雨定理を適用するために必要な条件判定と，それを適用する際の形式変換を行なうアルゴリズムを示した．
- 上記のアルゴリズムを，関数型言語 Haskell のコンパイラで現在もっとも利用されている Glasgow Haskell Compiler (GHC) 上に実装し，いくつかのベンチマークで融合変換の効果を示した．

本章では，これらの行なったことについてもう一度振り返り，さらに現在残されている問題点と今後の課題について解説する．

7.1 中間データ構造を除去するためのアルゴリズム

関数型プログラミングでは，基本となる機能をもった複数の関数群を組み合わせることによってプログラムを記述することが多い．このようなプログラミングスタイルでは，プログラムの可読性が高く再利用も容易となるなどの長所があるものの，関数合成の際に不要なデータ構造が生成され受け渡しに利用されるなどの問題もある．そこで，そのようなユーザが書きやすい簡潔なプログラムを入力としてコンパイラ内部でプログラム変換を行なうことによって，自動的に効率のよいプログラムを生成することが重要である．

このような変換の一つとして知られる融合変換に対し，構成的アルゴリズム論の観点からこれを実現しようとするのが Hylo 融合変換である．カテゴリ論などに基づいた背景となる理論については，これまで研究が進められており，また Hylo 形式を導出する具体的なアルゴリズムも提案された．そこで本論文では，この手法を具体的に実装するにあたり，自動化しなければならない部分を明確にし，そのためのアルゴリズムを明示的に示した．こ

- ψ 側の Hylo 再構成のアルゴリズムを提示
  これまでに ϕ 側の再構成アルゴリズムは示されてきたが，双対の概念である ψ 側も示すことによって，再構成を行なった際 ϕ 部と ψ 部の両側を簡略化することが可能になった．
- τ, σ 導出アルゴリズムの提示
  これまでは酸性雨定理を適用する際に必要であった，ある条件を満たすべき関数 τ, σ について，元となる関数 ϕ, ψ の構造に注目して，そこから文法的に τ, σ を導出するアルゴリズムを明示した．

7.2 融合変換アルゴリズムの実装と評価

我々は上記の融合変換アルゴリズムを，関数型言語 Haskell のコンパイラで現在もっとも広く用いられている Glasgow Haskell Compiler 上の最適化パスの一部として実装を行なった．独自のシステムでなく，このように汎用的に用いられている処理系上に変換部分を組み入れることによって，より多くの人々に利用されることが可能になり，問題点の発見やさらなる改善が行ないやすくなるものと思われる．GHC コンパイラは高速な実行ファイルを生成することに定評があり，内部では複雑な最適化を行なっているのだが，その長所をさらに押し進める働きを担うであろう．

また，Haskell コンパイラに対して CGI を用いた WWW のインターフェースを作成することによって，我々が実装したコンパイラ内部の変換処理の結果を手軽に視覚的に見ることを可能にした．これにより，現在のコンパイラでは行なえないような，ユーザ定義の再帰型に対する融合変換などを，ネットワーク越しに簡単に体験することが可能になっている．GHC は先に述べたような利点があるものの，その反面規模が大きくインストールすると 200MB 近くのディスクを必要とする．したがって手軽にインストールできる環境が無いような場合には，ネットワーク経由でのコンパイルを試すことができるのが有効である場合もあるであろう．

さらに，Haskell 言語における標準的なベンチマークである NoFib ベンチマークを用いて，Hylo 融合変換の有無によるコンパイル時と実行時の時間/空間消費量を比較することによって，融合変換の有効性を示した．また構成子式への適用や大域変数の展開など，Hylo 融合変換をより広く適用するための手法も提案し，それらを用いることによるヒープ使用量の差も検証した．本来 GHC に備わっている foldr/build 融合変換との比較，すなわち両者の融合変換を有効無効にした四つの各組み合わせや，両者の順序を入れ替えた変換など様々な試行実験を行なうことによって，両方の変換を最大限活かせる条件を示した．

7.3 今後の課題

本研究により構成的アルゴリズム論からなる理論面の研究成果が，コンパイラなどの実用的な分野にも有用であることが示されたといえよう．しかしまだ多くの実装上の問題点や，変換が適用できる対象範囲などについて今後すべき課題も残されている．

変換対象となる Core 言語プログラムの拡大

4.2.3 節で述べたように，現在の Hylo 融合変換の実装では `[m..n]` のような列挙により生成されたリストに対する融合変換が行なえない．このような列挙によるリストは関数型言語のプログラムでは多く用いられているため，消費されるヒープ使用量を他の融合変換手法と比べたときに，この点で大きな不利となってしまう．

これは `let` を多く含むような Core プログラムで問題が発生することが知られている．これに対応する方法として，3 章のアルゴリズムには手を加えず，その式が持つ性質，例えば可換性や結合性，if 文全体への処理を then 部と else 文に分配する処理などを適切に行なうことによって Hylo 融合変換で扱える形式を用意する方法が考えられる (4.2.3 節参照)．しかしこの手法のように，何らかの処理を施してから融合変換を行なう場合，その処理が本来の Core プログラムを操作してしまい，その結果が後に行なわれる最適化変換に悪影響を及ぼす可能性もあることに注意しなければならない．本論文の実装では，そのような効率悪化の要因を不必要に混入してしまうのを避けるため，安全寄りの実装，すなわち必要以上に元のプログラムは操作せず，それで融合変換できない場合はあきらめるといったスタンスをとったため，十分な融合変換が行なえなかった箇所もあるであろう．

またもう一つの方法としては，入力となる Core プログラムはそのままで Hylo 変換アルゴリズムの方で対応させることが考えられる．この方法では，本論文で示したような簡潔なアルゴリズムとして記述することは

難しくなるが，より細かい処理も変換アルゴリズムの中で制御できるので，変換後の実行効率を追及するなら，こちらの方が適当であろう．

他の最適化手法との組み合わせ

[Chi00] でも言われているが，融合変換はそれ単体ではコンパイルされたプログラムの実行時間を大幅に減らすことは多くないことが知られている．これは本来ヒープ使用量を減らす目的で用いられる変換であるため，大量の高速なメモリが利用できる環境では融合変換を行なわなくともあまり影響しない，という側面がある．しかし融合変換を行なうことによって分離されていた式が相互に近付き一つの式になる可能性があり，その後に行なわれる第二，第三の最適化においてさらなる変換が進むことがあり得るからである．

このような側面を融合変換は備えているため，本論文での実装は，入力としてできるだけ簡略化された形を想定していたため Core 言語上の変換の終わり近くで融合変換を行なったが，これを改善し任意の箇所で十分な融合変換を行なえるように修正し，Core 言語上の最適化変換のなるべく早い段階で Hylo 融合変換を行なうことによって，さらに効率的なプログラムが生成することも可能であろう．

また 5.3.4 節で述べたように， shortcut 融合変換を行なわなければ Hylo 融合変換が可能な場合でも，shortcut 融合変換を先に行なうことによりその後では形が変わってしまったために Hylo 形式が導出できず融合変換が行なえない例が存在した．これも 7.3 節の問題と関連するが， shortcut 融合変換後の Core 言語の形に依存しないように Hylo 関連のアルゴリズムを適応させることで，このような問題を避けることができると思われる．

相互 Hylo の融合変換

[IHT98, IHT01] で示されるように，組化手法と融合変換を組み合わせることによって，`zip` のような複数のリストを同時に扱う関数も融合変換の対象とすることができる．このためには 3 章で述べた各種のアルゴリズムを，相互再帰を扱えるように拡張する必要がある．

プログラムの電卓

これまでに，構成的アルゴリズム論に基づくいくつかのプログラム変換が提案されてきた．これらはみな入力と出力が同じプログラム上の変換であり，本来は任意の順番で適用することが可能なはずである．そこで，ユーザが自分のプログラムに対して，好きなプログラム変換を選択しそれを様々な順序で適用できるようなシステムが考えられるであろう．これはちょうど，人間が数の計算を行なう際に電卓を用いるのと良く似ていて，数をプログラムへ，四則演算をプログラム変換へと自然に拡張した形になっている．このようなシステムであるプログラムの電卓 (program calculator) を作成すれば，新しいアルゴリズムの発見や変換の正当性の証明などに活かせるものと考えられる．

謝辞

まずはじめに，指導教官である武市正人教授には，研究全般に関して温かく御指導頂き，心から感謝いたします．なかなか研究の進まない私を，ときには励まし，ときには奮いたたせ，ここまで支え続けていただけたことへの感謝の念は，言葉で表わすことが難しいほどであります．

胡振江助教授の熱心な指導も，大きな励みになりました．私がプログラム変換システムを作成するにあたって，変換の基となる理論を開発した胡助教授が身近にいなければ，この研究はまったく進まなかったことでしょう．

電通大の岩崎英哉助教授には，胡助教授も含め，システム開発当初，毎週のようにセミナーを開いていただき，実装が軌道にのるまでの強い後押しとなることができました．またお忙がしい時でも，私のつたない文章を校正してもらえたことは数知れません．ここに感謝いたします．

また研究のきっかけとなった構成的アルゴリズム論に基づく融合変換の定理を開発された国立情報学研究所の高野明彦教授には，実装の初期の段階において多くの有益なアドバイスを頂きました．先生は，その幅広い見識により，私の研究の位置付けを的確にアドバイスしていただきました．ここに感謝いたします．

情報処理工学研究室の皆様には，研究室輪講やふだんの生活における何気ない議論なども含めて，いろいろな面で助けてもらったと思います．中でも田中久美子講師と渡邉瑠璃子秘書には，私が助手の仕事をしながらの論文執筆だったため，仕事の負担をかけないように気を遣っていただけたことは忘れません．ありがとうございました．

最後になりますが，私がこうして研究を続けてこられたのは，隠れた家族の支えがあったからに他なりません．これまでどうもありがとう．

参考文献

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BD77] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, Vol. 24, No. 1, pp. 44–67, 1977.

[BdM94] R.S. Bird and O. de Moor. Relational Program Derivation and Context-free Language Recognition. In A.W. Roscoe, editor, *A Classical Mind*, pp. 17–35. Prentice Hall, 1994.

[CDPR97] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute Grammars and Functional Programming Deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, Paris, France, 1997.

[CDPR99] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative Program Transformation: A Deforestation Case-Study. In *Principles and Practice of Declarative Programming*, pp. 360–377, 1999.

[Chi88] Olaf Chitil. Common Subexpressions are Uncommon in Lazy Functional Languages. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages 1997*, LNCS 1467, pp. 53–71, St. Andrews, Scotland, 1988. Springer.

[Chi90] W. Chin. *Automatic Methods for Program Transformation*. PhD thesis, University of London, 1990.

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

[Chi99] Olaf Chitil. Type Inference Builds a Short Cut to Deforestation. *ACM SIGPLAN Notices, Proceedings of ICFP'99*, Vol. 34, No. 9, pp. 249–260, 1999.

[Chi00] Olaf Chitil. *Type Inference Based Deforestation of Functional Programs*. PhD thesis, RWTH Aachen, 2000.

[Feg96] Leonidas Fegaras. Fusion for free! Technical Report CSE-96-001, Oregon Graduate Institute of Science and Technology, 1996.

[FlM91] Pascal Fradet and Daniel le Métayer. Compilation of Functional Languages by Program Transformation. *ACM TOPLAS*, Vol. 13, No. 1, pp. 21–51, 1991.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics*. Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[FW88] A. B. Ferguson and Philip Wadler. When will Deforestation Stop? In *1988 Glasgow workshop on Functional Programming*, pp. 39–56, 1988.

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

[Gil96] Andrew John Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, 1996.

[GJ98] Jeremy Gibbons and Geraint Jones. The Under-Appreciated Unfold. In *Proceedings 3rd ACM*

SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept 1998, pp. 273–279. ACM Press, New York, 1998.

[GLJ93] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A Short Cut to Deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture '93*, pp. 223–232. ACM Press, 1993.

[Har91] Pieter H. Hartel. Performance of Lazy Combinator Graph Reduction. *Software — Practice and Experience*, Vol. 21, No. 3, pp. 299–329, 1991.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. An Extension of the Acid Rain Theorem. In *2nd Fuji International Workshop on Functional and Logic Programming (Fuji'96)*, pp. 91–105, Shonan Village, Japan, November 1996. World Scientific.

[HIT96b] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving Structural Hylomorphisms from Recursive Definitions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pp. 73–82, Philadelphia, USA, May 1996. ACP Press.

[HL93] Pieter H. Hartel and Koen G. Langendoen. Benchmarking Implementations of Lazy Functional Languages. In *ACM Conf. FPCA '93*, pp. 341–349, 1993.

[HTC98] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in Calculational Form. In *POPL '98*, pp. 316–328, 1998.

[IHT98] Hideya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards Manipulation of Mutually Recursive Functions. In *Fuji International Symposium on Functional and Logic Programming*, pp. 61–79, 1998.

[IHT01] 岩崎英哉, 胡振江, 武市正人. 漸次的組化と融合による関数プログラムの最適化. コンピュータソフトウェア, Vol. 18, No. 0, pp. 46–59, 2001.

[Iwa98] 岩崎英哉. チュートリアル 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, pp. 57–70, 1998.

[Joh87] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture*, pp. 154–173, 1987.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Jon94] Mark P. Jones. *Gofer 2.30a Release Notes*, 1994.

[Jon95] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, pp. 97–136, 1995.

[JS98] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, Vol. 32, pp. 1–3, 1998.

[Jür00] Claus Jürgensen. A Formalization of Hylomorphism Based Deforestation with an Application to an Extended Typed λ-Calculus. Technical Report TUD-FI00-13, Technische Universität Dresden, Fakultät Informatik, D-01062 Dresden, Germany, November 2000.

[LS95] John Launchbury and Tim Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *FPCA '95*, pp. 314–323. ACM Press, 1995.

[Mar95] Simon David Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Glasgow University, 1995.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 124–144, Cambridge, Massachuetts, August 1991.

[NJ98] László Németh and Simon Peyton Jones. A Design for Warm Fusion. In *The 10th International Workshop on Implementations of Functional Languages*, pp. 381–393, 1998.

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

[OHIT97] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A Calculational Fusion System HYLO. In Richard Bird and Lambert Meertens, editors, *IFIP TC2 WG2.1 Algorithmic Languages and Calculi*, pp. 76–106, Alsace, France, Feburary 17–22 1997. Chapman & Hall.

[OHIT00] 尾上能之, 胡振江, 岩崎英哉, 武市正人. プログラム融合変換の実用的有効性の検証. コンピュータソフトウェア, Vol. 17, No. 3, pp. 81–85, 2000.

[OHT98] 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, pp. 52–56, 1998.

[Par01] Alberto Pardo. Fusion of Recursive Programs with Computational Effects. *Theoretical Computer Science*, Vol. 260, No. 1–2, pp. 165–207, 2001.

[PP93] A. Pettorossi and M. Proietti. Rules and Strategies for Program Transformation. In *IFIP TC2/WG2.1 State-of-the-Art Report*, pp. 263–303. LNCS 755, 1993.

[Sch00] Jacob B. Schwartz. Eliminating Intermediate Lists in pH. Master's thesis, Massachusetts Institute of Technology, 2000.

[SF93] Tim Sheard and Leonidas Fegaras. A Fold for All Seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture '93*, pp. 233–242. ACM Press, 1993.

[Tea96] The AQUA Team. *Glasgow Haskell Compiler User's Guide ver0.29*, 1996.

[THT98] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program Transformation in Calculational Form. *ACM Computing Surveys*, Vol. 30, No. 3, 1998.

[TM95] Akihiko Takano and Erik Meijer. Shortcut Deforestation in Calculational Form. In *Proc. Conference on Functional Programming Languages and Computer Architecture '95*, pp. 306–313. ACM Press, 1995.

[Tüf98] Chiristian Tüffers. Erweiterung des Glasgow-Haskell-Compilers um die Erkennung von Hylomorphismen. Diplomarbeit in Informatik, Lehrstuhl für Informatik II, RWTH-Aachen, Germany, 1998.

[Tur86] Valentin F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS*, Vol. 8, No. 3, pp. 292–325, 1986.

[Tur88] Valentin F. Turchin. The Algorithm of Generalization in the Supercompiler. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 531–549. North-Holland, 1988.

[Wad87] Philip Wadler. List comprehension. In Simon L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1987.

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

[Wad89] Philip Wadler. Theorems for Free! In *The Fourth International Conference on Functional Programming Language and Computer Archtecture*, FPCA '89, pp. 347–359, Imperial College, London, September 11–13 1989. ACM Press and Addison-Wesley.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 60–76. ACM, 1989.

発表論文

学術雑誌掲載論文

1. 尾上能之. プログラミング言語 Haskell とその処理系, コンピュータソフトウェア, **18**, 6 (2001), 59–66.
2. 尾上能之, 胡振江, 岩崎英哉, 武市正人. プログラム融合変換の実用的有効性の検証, コンピュータソフトウェア, **17**, 3 (2000), 81–85.
3. 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, **15**, 6 (1998), 52–56.
4. 金子敬一, 尾上能之, 武市正人. 完全遅延評価に適した関数プログラムの共有解析, 情報処理学会論文誌, **35**, 3 (1994), 391–403.

学会発表 (査読あり)

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

学会発表 (査読なし)

1. 尾上能之, 武市正人. Shortcut Deforestation における効率の解析, 日本ソフトウェア科学会 第 12 回大会 (September 12–14 1995).
2. 尾上能之. 完全遅延評価による部分計算の問題点, Functional and Logic Programming Symposium (July 18–20 1994).
3. 尾上能之, 金子敬一, 武市正人. 関数プログラムのコンパイラにおける型情報を用いた効率的な共有解析の実現, 第 73 回 情報処理学会 記号処理研究会 (November 19 1993).
4. 尾上能之, 広津千尋, 栗木哲. いくつかのノンパラメトリック検定法の正確な有意確率評価, 応用統計学会 年会 (April 25 1992).

研究報告

1. Kaneko, K., Onoue, Y. and Takeichi, M. Sharing Analysis of Functional Programs for Fully Lazy Evaluation, Technical Report METR 93-10, Faculty of Engineering, University of Tokyo (1993).

卒業/修士論文

1. 尾上能之. 型情報を用いた関数プログラムの共有解析に関する研究, 修士論文, 東京大学 大学院 工学系研究科 計数工学専攻 (1994).
2. 尾上能之. いくつかのノンパラメトリック検定法の正確な有意確率評価, 卒業論文, 東京大学 工学部 計数工学科 (1992).