# 停止性の保証による rewrite rules の改良

長田 朋久　市川 和央　千葉 滋

本研究では GHC の rewrite rules を改良し，停止性を保証した機構を Java 言語上に実装する．ホスト言語の構文機能を使用して実現された内部 DSL は実装が単純である一方，コンパイラによる直接的な最適化が困難である．rewrite rules を用いると内部 DSL 向けの最適化処理を DSL 作者が記述できるが，各変換ルールは無制限に適用可能なため，停止しない場合がある．そこで提案手法では構文木の各節に対して変換規則ごとに一度のみの変換を適用させることで，停止性を保証し最適化処理の安全性を向上させる．

## 1　はじめに

内部 DSL は汎用プログラミング言語のライブラリとして作成されることが一般的である．例えば Scala [5] 言語は二項演算子のように見えるメソッドの糖衣構文が存在している．これは様々なライブラリで独自の演算子定義を可能とし，擬似的な言語の構成に寄与している．しかし，内部 DSL の意味論はコンパイラの知るところではないため，最適化が期待しづらい．関数型プログラミング言語の Haskell [2] では，Glasgow Haskell Compiler [4] 固有の拡張として rewrite rules [3] という機構が存在し，パターンマッチの要領で予め定義された関数の組み合わせを別の式へと変換する．これを内部 DSL の作者が記述することで，コードの最適化が期待できる．Rewrite rules の問題点として，書き換え規則適用の停止性が保証されていないことが挙げられる．例えば引数の順序を入れ替える書き換え規則は無限に適用され，結果として停止しない．これを解決するため，本研究では書き換え規則に適用順序を設け停止性を保証することを提案する．書き換え対象コードの構文木を帰りがけ順で走査し規則を試しつつ，書き換えによって生じた新しい構文木には同じ書き換え規則を再適用しない．また，提案手法を Java 言語上に実装し評価することを目標とする．

A refinement of the rewrite rules promising termination.
Tomohisa Osada　Kazuhiro Ichikawa　Shigeru Chiba, 東京大学大学院情報理工学系研究科, Dept. of Computer Science, University of Tokyo.

## 2　ホスト言語の構文を使用した内部 DSL

Scala [5] 等の汎用プログラミング言語では，その言語の構文機能のみで内部 DSL を実装することが容易である．ある汎用プログラミング言語 (ホスト言語) 内で，さらに特定の問題領域に特化されたサブ言語を内部 DSL という．内部 DSL はホスト言語の構文機能を使用し，1 つの言語に見えるようなホスト言語のライブラリとして実装される．図 1 に内部 DSL の例として，Scala のリスト表現を挙げる．1 行目では，1，2，3 より構成されるリストについて，全ての要素に 1 を足し，その後全ての要素に 2 を掛けている．この式は 2 行目と同等である．::は List クラスのメソッドであり，引数をリストの先頭に付け加えた新たなリストを返す．また，Nil は List クラスを継承したシングルトンオブジェクトである．よって，Nil から::メソッドが連鎖的に呼び出され，リストが生成されている．Scala ではメソッド呼び出しの際，呼び出し元オブジェクトと引数におけるドットと括弧を省略できる糖衣構文があるため，1 行目の簡潔な表記が可能となっている．また，_は引数を要求し，全体とし

```
1::2::3::Nil map (_ + 1) map (_ * 2)
Nil.::(3).::(2).::(1).map(x => x + 1).map(
    x => x * 2)
```

**図 1　Scala 言語の List の例**

```
{-# RULES
"map/map" forall f g xs.
          map f (map g xs) = map (f.g) xs
  #-}
```

**図 2　rewrite rules による中間オブジェクトの削減**

て無名関数を構成する．これはリスト表記に特化した Scala の内部 DSL ということができる．

しかし，ホスト言語は内部 DSL の意味論を理解し得ないため，コンパイラによる直接的な最適化は困難である．図 1 では map 関数が連続で使用されている．map 関数は新しいリストオブジェクトを作成するため，中間オブジェクトを無駄に生成している．

## 3　Rewrite rules の停止性

内部 DSL の最適化機構として Haskell [2] 言語には Glasgow Haskell Compiler [4] の拡張機能として rewrite rules [3] が存在する．rewrite rules では記述されているパターンで関数が使用された場合，ルールに基づき式を変換する．ルールは等式で記述され，左辺のパターンにマッチする式は右辺に変換される．この変換はマッチする式がある限り繰り返し実行される．図 2 に rewrite rules の定義例を示す．"map/map" はルール名，f，g，xs は変数であり，map はスコープ中に存在する map 関数を指す．このルールは連続で map 関数が使用される場合に，中間オブジェクトの発生を抑えている．右辺では f と g が関数合成されるため，map g xs による中間オブジェクトは生成されない．

Rewrite rules の問題点として，停止性が保証されていないことが挙げられる．図 3 のように関数 foo の引数の順序を入れ替えるだけのルールは変換結果に再度ルールが適用されるため，無限ループに陥る．

```
{-# RULES
  "commute" forall x y. foo x y = foo y x
  #-}
```

**図 3　無限ループに陥る例**

## 4　順序付き書き換え機構の提案

書き換え機構の停止性が保証できるよう，複数の書き換え規則間に適用順序を設けることを提案する．書き換え規則が停止しない場合とは，以下の 2 つが挙げられる．

- ある単一の規則が無限に再適用される．
- ある規則 $a$ が適用された後別の規則群 $b, c, d, ...$ が適用され，その結果再び $a$ が適用可能となることで無限の再適用が生じる．

よって提案手法では，

- ある構文木に対して特定の規則は有限回しか適用しない．
- ある規則 $a_i$ の適用された構文木にその後適用される規則は必ず $a_i$ 以降とする．

とすることで，書き換え機構の停止性を保証する．そして，停止性が保証できるような機構の中で実用上表現力が十分であるものを目指す．

### 4.1　書き換え規則の動作

図 4 に単一の書き換え規則の定義と動作の例を示す．1 行目は変数定義である． a,b,c は int 型の変数として定義し，書き換え規則中で使用する．2 行目は書き換え規則の本体である．1 つの書き換え規則は返り値型の等しい左辺式と右辺式の対からなり，左辺式のパターンにマッチする構文木上の式について，右辺式に変換する．この規則は 3 個の int の足し算における結合の優先順位を変更する．以下に書き換えを受ける式の例を示す．

```
1 + (2 + List(3,4,5).sum())
  -> (1 + 2) + List(3,4,5).sum()
```

書き換え規則にある二項演算子 + はコード上の + とマッチする．規則にある変数は任意の int 型の式とマッチする．よって，この場合は

```
1 int a,b,c;
2 a + (b + c) -> (a + b) + c
```

**図 4　書き換え規則**

1: TRAVERSE($S_{root}$,1)
2: **function** TRAVERSE($t$,$i$)
3: 　**for** $c \leftarrow$ CHILDREN($t$) **do**
4: 　　TRAVERSE($c$,$i$)
5: 　**for** $j \leftarrow i...n$ **do**
6: 　　**if** $t$ match with $a_j$ **then**
7: 　　　REWRITE($t$,$a_j$)
8: 　　　**if** $j < n$ **then**
9: 　　　　TRAVERSE($t$,$j+1$)

**図 5　書き換え規則列の適用順序**

`a = 1, b = 2, c = List(3,4,5).sum()` となっている．

構文木の構造が一致しない式とはマッチしない．例えば以下の式

`6 + 7 + 8`

は `+` が左結合の演算子なので，実際には `(6 + 7) + 8` である．つまり構文木としては右側の `+` が式の根なので，書き換え規則の左辺とは木の形が異なり，マッチしない．

### 4.2　書き換え規則列の適用順序

複数の書き換え規則に順序を付け，書き換え対象となるソースコードの構文木に帰りがけ順に適用していくことで，規則の無限適用を防ぐ．

図 5 に書き換え規則の適用順序について，擬似コードを示す．$S$ は書き換え対象のソースコードの構文木であり，$S_{root}$ はその根ノードを指す．$\{a_i\}, i = 1, ..., n$ は書き換え規則列である．関数 TRAVERSE はノードの参照 $t$ と試す書き換え規則の番号 $i$ を引数に取り，構文木を走査しつつ書き換える．関数 CHILDREN は引数のノードの子要素集合を返す．関数 REWRITE はノードの参照と書き換え規則を引数にとり，そのノードを根とする部分木を書き換える．具体的な動作としては， TRAVERSE はまず $t$ の子要素全てについて再帰的に TRAVERSE を実行する．その後，$i \leq j \leq n$ である任意の $j$ について， $t$ が書き換え規則 $a_j$ とマッチすれば書き換えを実行する．さらに，$a_j$ が最後の書き換え規則で無ければ，規則の順序として次のものを使用して TRAVERSE を再び実行する．

書き換え規則列の適用順序を構文木で表現したものを図 6 に示す．$i$ 番目の書き換え規則 $a_i$ は，根が $P$ である部分木にマッチし，$Q$ を根とした部分木に書き換えられるとする．書き換え対象のソースコードの構文木は帰りがけ順に全ての書き換え規則を試す．$P$ が見つかれば，根を $P$ とする部分木を，根を $Q$ とする部分木に書き換える．そして書き換え対象となった部分木については，その根ノードを始点として $i+1$ 番目以降の書き換え規則を帰りがけ順に試す．

### 4.3　書き換え機構の停止性

1 つの書き換え規則は $S$ の各ノードに対して有限回だけ有効である．また，ある書き換え規則 $a_i$ によって書き換えられた部分木に対して使用される可能性のある規則は $a_{i+1}$ 以降であるため，書き換え規則の順序に逆行した適用は発生しない．これにより， 生成された部分木に同じ書き換え規則が再度適用されることにより発生する無限ループを防ぐ．

## 5　表現力の比較

Rewrite rules では書き換え規則の無限適用に陥る問題が表現できない一方，本研究の提案手法は不定回数の再適用が必要な問題が表現できない．例えば 3 章で示したように，ある関数の引数順序を入れ替えることは rewrite rules では表現できない．提案手法では同じノードに同じ書き換え規則は 1 度だけ適用されるため，引数順序の入れ替えは可能である．一方，提案手法では同じノードに不定回数同じ書き換え規則を適用すべき問題が表現できない．例えば，右結合の足し算の列をすべて左結合に直すことは難しい．図 4 の規則を右結合の足し算列に適用する例を図 7 に示す．太字の部分が書き換えの際の根ノードである．図のように構文木として根から遠い順に 3 回，書き換

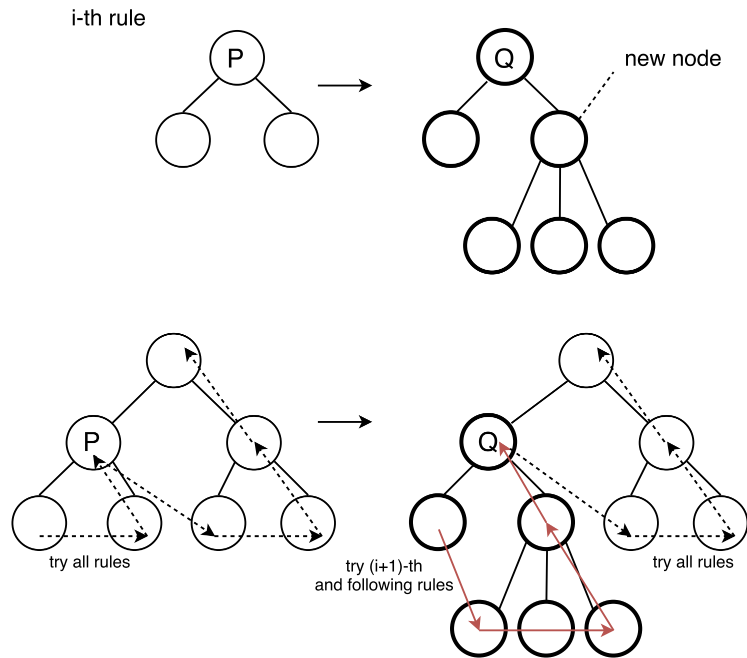


図 6　書き換え規則列の適用順序

```
1     1 + (2 + (3 + (4 + 5)))
2  -> 1 + (2 + ((3 + 4) + 5))
3  -> 1 + ((2 + (3 + 4)) + 5)
4  -> (1 + (2 + (3 + 4))) + 5
```

図 7　提案手法によって右結合の足し算列に，左結合に直す規則を適用させる

```
1     1 + (2 + (3 + (4 + 5)))
2  -> (1 + 2) + (3 + (4 + 5))
3  -> ((1 + 2) + 3) + (4 + 5)
4  -> (((1 + 2) + 3) + 4) + 5
```

図 8　右結合の足し算列の理想的な書き換え

えが発生しているが，結果出力される式は完全な左結合の足し算列とはならない．図 8 にこの場合の理想的な書き換え例を示す．各段階で構文木の根となるノードに規則を 3 回適用させると，結果は完全な左結合の足し算列となることが分かる．これは提案手法では，単一の書き換え規則で表現することができない．なお，図 4 の規則を複数並べることで，あらかじめ予想される長さの足し算列に対して対応できる．しかし，任意長の右結合の足し算列に対して対応することはできない．

## 6　関連研究

コンパイル時に最適化されたコードを生成させる機構としては，MetaOCaml [7] などのマクロ関数がある．マクロ関数と比べ本研究が対象とする書き換え規則は，複数の関数配置のパターンを処理することができる一方，事前の引数評価によるコード生成ができない．書き換え規則では図 2 で見たように，複数の関数があるパターンで適用されていれば，それをより効率的な式に書き換えることが可能である．しかしマクロ関数のように事前に一部の引数のみコンパ

イル時に評価することはできない．そのためマクロ関数では可能な `pow(2,3) -> 2 * 2 * 2` といった，引数の回数だけコード断片を再帰的に展開する，などの動作はできない．

C++[6] 用の書き換え機構として CodeBoost[1] がある．CodeBoost も本研究と同様に，書き換え規則に従って書き換えを実行する．しかし特徴的な点として，規則に付加的な条件を設定できる．例えば

```
z + (x * y) -> (x * y) + z,
not(is_mult_expr(z))
```

のように `z` は掛け算の式ではないと条件を付けることが可能であり，表現可能な問題領域が広い．

## 7 まとめと今後の課題

ホスト言語のライブラリである内部 DSL はコンパイラが直接最適化することは難しい．パターンマッチによる最適化機構として GHC には rewrite rules があるが停止性が保証されていない．よって本研究では停止性が保証されるような書き換え規則の適用順序を提案した．各書き換え規則は，書き換え対象の構文木の各ノードにつき 1 度だけ適用される．また，複数の書き換え規則間には順序があり，遡った適用はされない．提案手法は rewrite rules で無限ループに陥る問題を記述可能である．しかし，あるノードに複数回同一の書き換え規則を適用させることが本質である問題に対処できない．

今後の課題としては，提案手法の停止性を厳密に証明すること，同手法の表現力が向上するような改良を加えること，Java で提案手法を実装し評価することが挙げられる．

## 参考文献


[1] Bagge, O. S., Kalleberg, K. T., Haveraaen, M., and Visser, E.: Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs, *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, IEEE, 2003, pp. 65–74.

[2] Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P.: A History of Haskell: Being Lazy with Class, *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, New York, NY, USA, ACM, 2007, pp. 12–1–12–55.

[3] Jones, S. P., Tolmach, A., and Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC, *Haskell workshop*, 2001, pp. 203–233.

[4] Jones, S. P., Hall, C., Hammond, K., Partain, W., and Wadler, P.: The Glasgow Haskell compiler: a technical overview, *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, Citeseer, 1993.

[5] Odersky, M., Spoon, L., and Venners, B.: *Programming in scala*, Artima Inc, 2008.

[6] Stroustrup, B.: *The C++ programming language*, Pearson Education India, 1995.

[7] Taha, W.: A gentle introduction to multi-stage programming, *Domain-Specific Program Generation*, Springer, 2004, pp. 30–50.