密な演算子呼び出しで実現した内部 DSL の前処理による実行速度改善の試み

長田 朋久 †　　千葉 滋 ‡
東京大学大学院情報理工学系研究科 *

1　はじめに

ユーザ定義演算子は柔軟な内部 DSL を実現するのに役立つ．各構文をユーザ定義演算子の組み合わせで表現することで, ユーザによる擬似的な構文定義が可能となる．例えば正規表現を構文として定義できる．しかしこの方針で設計された DSL は多段の演算子呼び出しが発生するため，実行時のオーバーヘッドが深刻である．本研究では，演算子呼び出しの密な組み合わせを前処理することで多段呼び出しのオーバーヘッドを削減する．ユーザ定義演算子が使える簡単な言語を作成し，その言語上で演算子呼び出しの密な組み合わせに対する前処理としてインライン展開を実装した結果を報告する．

2　内部 DSL

2.1　内部 DSL の問題点

DSL(Domain Specific Language) とは特定の問題領域に特化したプログラミング言語であり, 当該領域に対し自然な記法を提供する．内部 DSL とは汎用的な言語上でライブラリとして実現された DSL である．

内部 DSL の問題点として, ホスト言語の文法制約を継承することが挙げられる．ここで, Scala 言語におけるテスト用ライブラリである ScalaTest を例に説明する．ソースコード 1 は, Stack オブジェクトの振る舞いを検査している．2 行目にて変数 stack に Stack オブジェクトが代入され, 3 行目で整数 1 が入る．そして, 4 行目にて変数 stack から取り出した値が 1 と等しいことを検査している．このように, (左辺) should be (右辺) の形式で両辺の等価性を自然な記法で表明することが可能である．しかし同時に Scala の文法上の制約から右辺の括弧は外すことができない．これは, 4 行目は実際には 5 行目のようなメソッド呼び出しで構成されているためである．詳述すると, メソッド should は糖衣構文により, ドットと引数を囲う括弧が省略可能である．しかし, apply という名前のメソッドはメソッド名自体が省略可能な一方, それを省略した際に引数の括弧が省略できない．この例に示されるように, 内部 DSL はホスト言語以上の表現力を得ることはできない．

ソースコード 1　ScalaTest による内部 DSL の例

```
1  "A Stack" should "pop values" in {
2    val stack = new Stack[Int]
3    stack.push(1)
4    stack.pop() should be (1)
5    // stack.pop().should(be.apply(1))
6  }
```

2.2　密な演算子呼び出しによる内部 DSL

前述のような内部 DSL の表現制約を緩和させる試みとして, 密なユーザ定義演算子呼び出しにより内部 DSL を構成する手法がある．ユーザ定義演算子を組み合わせて使用し, 新たな構文のように見せかけることで内部 DSL とする．例として, 2 進数リテラルを演算子で構成する方法を擬似言語で記述したものをソースコード 2 に示す．コロンより左は演算子の構文定義であり, 右は演算子の (関数) 型である．構文定義は name part と hole part の並びで構成される．name part はダブルクォートで囲まれた部分であり, 文字列そのものを期待する．hole part はアンダースコアの部分であり, 特定型の値を期待する．例えば 1 行目の演算子は前置の単項演算子と解釈できる．また, これら演算子の本体部分の定義例をソースコード 3 に示す．型 Binary は 2 つの整数の組であり, 実際の値と, その値の 2 進数表現での桁数を表す．3, 4 行目は最上位桁に新たに 1 が加わった場合の演算である．このような演算子を定義すれば, 0b101 といった 2 進数リテラルが 0b(1(0(1()))) のように, 入れ子になった密な演算子呼び出しの組み合わせとして解析可能となる．

密な演算子呼び出しにより内部 DSL を定義可能な言語として, ProteaJ[2] が存在する．ProteaJ では Java の構文に加え, protean operators と呼ばれる name part と hole part の並びから成る演算子が定義可能であり, これにより内部 DSL を定義する．

An attempt to improve execution performance by preprocessing an embedded DSL constructed by dense operator calls

†Tomohisa Osada　‡Shigeru Chiba
* Graduate School of Information Science and Technology, The University of Tokyo

ソースコード 2　演算子による 2 進数リテラル

```
1  "0b" _ : Binary => Int
2  "0"  _ : Binary => Binary
3  "1"  _ : Binary => Binary
4  "0"    : Binary
5  "1"    : Binary
```

ソースコード 3　演算子本体の定義例

```
1  "1"  _ : Binary => Binary =
2  { (p : Binary) =>
3    (p.acc + (1 << p.numOfDigits),
4    p.numOfDigits + 1)
5  }
```

この手法の問題点として, 演算子呼び出しが密に発生することにより, 実行速度が低下することが挙げられる. 表 1 に, 2 進数リテラルがホスト言語に組み込みで実装されている場合と, 密な演算子呼び出しの内部 DSL の場合, さらに, 密な演算子の内部 DSL かつインライン展開を実行した場合 (後述) の実行速度比較を示す. 実験はユーザ定義演算子により内部 DSL が定義可能な独自言語のインタプリタを Scala 言語にて作成し, その上で行った. 2 進数 `0b10101` を 40,000 回評価することを 1 試行とし, 100 回試行した際の実行時間の平均と標準偏差を求めた. このように, 密な演算子呼び出しの内部 DSL の場合は演算子呼び出しのオーバーヘッドにより, 評価に時間がかかることが分かる.

表 1　内部 DSL の実装方法による実行時間比較

実装方法	平均時間 [ms]	標準偏差 [ms]
組み込み実装	22.00	2.38
演算子呼び出し	324.72	16.66
インライン展開使用	1975.54	154.75

3　構文木変換による高速化の試み

演算子呼び出しのオーバーヘッドを改善するためには, 構文解析後の構文木を変換することが有効であると考えられる. 具体的には, 同一 DSL 上で複数定義された演算子の関係性を自動抽出し, 変換規則を作成して適用することが考えられる. それにより評価時間の短い構文木を得る. また, 構文木変換の最も単純な例として, 演算子呼び出しのインライン展開が考えられる. 本稿では第一歩として, 先述の独自言語にインライン展開機能を実装した.

先述の 2 進数リテラルの実験を同様に行った結果を, 表 1 の「インライン展開使用」の項に示す. 今回はインライン展開後の方が評価が遅くなった. これはインライン展開前に演算子呼び出しの部分で適用されていた Java の JIT コンパイラによる最適化機能が有効に動作しなくなったためと考えられる.

4　関連研究

構文木の書き換え規則を明示的にプログラマが書く先行研究として Glasgow Haskell Compiler (GHC) の rewrite rules [3] が存在する. これは, コンパイル中に書き換え規則を参照し, 適合するものがあれば随時変換する Haskell の拡張機能である. Haskell のライブラリ作者がそのライブラリ固有の書き換え規則を記述することを想定し, GHC でサポートしていないドメイン固有のコンパイル最適化の手段を提供している. また, このような書き換え規則を C++ で実現したフレームワークとして CodeBoost [1] がある. これらの研究は書き換え規則を主にユーザが記述する点で本研究とは異なる. 本研究では内部 DSL に定義されたユーザ演算子の関係性から自動で書き換え規則を生成することを目指す.

5　まとめと今後の課題

本稿では, 密なユーザ定義演算子の呼び出しの組み合わせによる内部 DSL は実行速度が比較的遅いことを問題とし, ユーザ定義演算子の構文木変換を提案した. また, その簡単な例としてインライン展開を独自言語に実装し, 2 進数リテラルの例で有効性を検証した. 今後の課題としては, インライン展開以外の具体的な構文木変換手法を考案することや, 実用的な言語で同手法を実装し性能を評価することが挙げられる.

参考文献

[1] O. S. Bagge and M. Haveraaen. Domain-specific optimisation with user-defined rules in codeboost. *Electronic Notes in Theoretical Computer Science*, 86(2):119–133, 2003.

[2] K. Ichikawa and S. Chiba. Composable user-defined operators that can express user-defined literals. In *Proceedings of the 13th international conference on Modularity*, pages 13–24. ACM, 2014.

[3] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, volume 1, pages 203–233, 2001.