東京大学
情報理工学系研究科 創造情報学専攻
修士論文

内部ドメイン固有言語の実行時情報を利用する書き換え機構の提案

A rewriting system using run-time information of embedded domain specific languages

長田 朋久
Tomohisa Osada

指導教員　千葉 滋 教授

2017 年 1 月

概要

プログラムの実行時の値を利用する書き換え機構を提案する．内部ドメイン固有言語では対象の問題領域のみで適用可能な，式の書き換えが最適化処理として有効である．そのような機構として，関数型言語 Haskell では rewrite rules が存在する．同機構はコンパイル時にプログラム中の式の型情報を利用し，書き換え規則に従って書き換え処理をする．しかしオブジェクト指向言語では型の継承機構により，オブジェクトの実行時型をコンパイル時に取得できない．よって本研究では，実行時に書き換え処理をすることでオブジェクト指向言語において実行時型の情報を補うことを提案する．さらに実行時の値を参照することで，書き換え規則をパターンマッチのように使用できることを示す．また，その機構を Java 上で実装し，性能を評価する．

Abstract

We propose a rewriting system using run-time information. In embedded domain specific languages, rewriting expressions can be applied to the domain is valid for optimization. There are rewrite rules in a functional programming language, Haskell. Rewrite rules rewrite a program following the rules by using type information in compile-time. There are no run-time types of objects, however, in object-oriented languages because of type inheritance system. We propose rewriting in run-time so that it covers information of run-time types in this study. Furthermore, we show that rewriting can be used as pattern matching by looking values of run-time. We implement the system in Java and assess efficiency.

目次

第 1 章

はじめに

ドメイン固有言語 [1]（Domain specific language, DSL）とはある特定の問題領域に特化したプログラミング言語を示す．例えばデータベースへアクセスする DSL として SQL[2] や，プログラムのビルドツール GNU make[3] に使用される設定ファイル Makefile，グラフ可視化ソフトウェア Graphviz[4] が使用する DOT 言語などが存在する．dot 言語による有向グラフの記述と表示を図 1.1，図 1.2 に示す．1 行目はグラフ名とグラフの形式であり，それ以降が頂点と辺の定義である．オペレータ->は有向辺を，その左右は頂点を表す．頂点または辺の定義の右辺には属性を指定することができる．DOT 言語によってグラフ構造を簡単に記述することができる．

汎用プログラミング言語の一部で DSL を使用する場合，直接文字列として挿入すると元の言語（ホスト言語という）の型機構等のサポートが得られない．そのためホスト言語のライブラリとして DSL を作成する．これを内部 DSL という．内部 DSL は問題領域を自然に記述できるように工夫され実装される．例えば Scala[5] 言語などには内部 DSL の実装をサポートするための糖衣構文が存在し，内部 DSL として作成されたライブラリとしては ScalaTest[6]，Scala Parser Combinator[7]，Slick[8] など多義に渡る．図 1.3 に Scala Parser Combinator による構文解析の例を示す．オブジェクト `ParenParsers` の中で定義されている 3 つのメソッドがパーサーであり，`Parser[String]` 型を持つ．メソッド `leftParen` は文字列 `"("` を受理する．メソッド `paren` は左右の対応が取れる括弧が入れ子になった文字列，例えば `"((()(())))"` などを受理する．`paren` の右辺のチルダは `p ~ q` の形で使用され，`p` のパース

```
1 digraph G {
2   a [shape = box];
3   b [shape = triangle];
4   c [color = lightblue];
5   a -> b;
6   a -> c;
7   a -> d [style = dotted];
8 }
```

図 1.1. dot 言語による有向グラフの定義

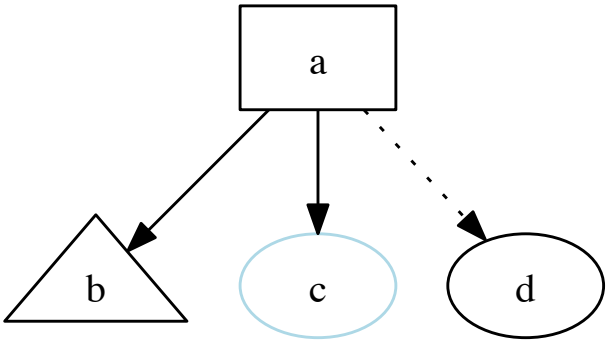

図 1.2. dot 言語による有向グラフのコンパイル結果

```
1  object ParenParsers extends RegexParsers {
2    def leftParen  : Parser[String] = "("
3    def rightParen : Parser[String] = ")"
4    def paren      : Parser[String] =
5      leftParen ~ rep(paren) ~ rightParen ^^ {
6        case left ~ parenList ~ right =>
7          left + parenList.mkString + right
8      }
9  }
```

図 1.3. Scala Parser Combinator による構文解析の例

が成功すれば `q` のパースを実行する `Parser` クラスのメソッドである．`rep(p)` はパーサー `p` の 0 回以上の繰り返しを表す．`p ^^ f` は `p` のパース結果に関数 `f` を適用するメソッドである．このように Scala Parser Combinator は構文解析用の DSL をホスト言語内で実現している．

しかし，内部 DSL はホスト言語のライブラリであるため，意味論を考慮した最適化をコンパイラに期待することができない．内部 DSL の意味論を考慮した最適化を内部 DSL の作者があらかじめ記述できる機構として Grassgow Haskell Compiler[9] には rewrite rules[10] が存在する．Rewrite rules は書き換え規則を定義することが可能で，コンパイラは書き換え規則に従って内部 DSL 使用者の記述したコードを書き換える．これにより内部 DSL の関数があるパターンで使用された場合に別の関数に置き換えるなど，内部 DSL の意味論を考慮した最適化が可能となる．書き換えパターンの例としては連続する関数適用に対して中間データ構造をストリームに変換する stream fusion[11] が代表的である．ここで，オブジェクト指向言語においては rewrite rules の機構を愚直に適用すると問題が生じる．オブジェクト指向言語

ではクラスの継承機構によって，コンパイル時にオブジェクトの実行時型を判別することができないためである．そこで本稿では実行時に一部のメソッド本体を書き換えることでオブジェクトの実行時型を判別させることを提案する．実行時の値が参照可能となるため，書き換え規則をパターンマッチのように利用することも可能である．また，行列の内部 DSL で書き換えを実行することで提案手法の有効性を確認する．

以下， 2 章では研究背景として内部 DSL，rewrite rules を紹介しオブジェクト指向言語における問題を詳説する．また，理論的な研究として項書き換え系を紹介する． 3 章では提案する実行時書き換え機構の概要を説明する． 4 章では今回実装した Java 用の実行時書き換え機構ライブラリ，及び実装に使用したバイトコードを抽象構文木に直すライブラリ Bytespresso について説明する． 5 章では行列の内部 DSL を使用し，行列積の計算に実行時書き換えを利用することで実行速度が向上することを示す． 6 章では関連研究を説明する． 7 章でまとめと今後の課題について説明する．

第 2 章

研究背景

内部 DSL の最適化には書き換え機構が有効だが，オブジェクト指向言語の実行時型は判別できない．内部 DSL はホスト言語の構文を技巧的に使用することで実現されるライブラリであるため，内部 DSL の意味論から導出される計算の最適化をホスト言語のコンパイラに期待することはできない．内部 DSL の実行効率を高めるための工夫として，関数型言語 Haskell [12] には rewrite rules [10] と呼ばれるプログラムの書き換え機構が存在する．特定の内部 DSL に固有な書き換え規則を併記しておくことで実行効率の改善が可能だが，コンパイル時に型情報を使用して書き換えを実行するため，オブジェクト指向言語に存在するオブジェクトの実行時型を判別することができない．以下に詳述する．また，書き換え規則の理論である項書き換え系についても本章で述べる．

2.1 内部 DSL

内部 DSL とは汎用言語内に存在する，ある特定の問題領域を記述することに特化したライブラリのことである．内部 DSL に対してその元になる汎用言語のことをホスト言語という．まず，内部 DSL を定義するためのホスト言語の構文サポートについて説明する．Scala [5][13] 言語ではメソッド定義などで内部 DSL を表現できるように糖衣構文 [14] が用意されている．糖衣構文の例を図 2.1，図 2.2 に示す．図 2.1 の各行の式はコメント以下の式と同等である．1 行目は オブジェクトに対するメソッド呼び出しにおいてレシーバーオブジェクトと引数を明示する場合，ドットと括弧が省略可能であることを表す．Scala ではメソッドに特定種類の記号を使用可能なため，この機能によって独自の演算子を定義することが可能である．実際，2 行目のように整数の加算は Scala ではメソッドで定義されている．3 行目のようにコロンから始まるメソッド名は右結合である．よって，3 行目のメソッドは `List` クラスのオブジェクト `Nil` がレシーバーオブジェクトである．4 行目は配列 `array` の 3 番目の要素を示し，これは `apply` メソッドの呼び出しと同等である．よって，ユーザが `apply` メソッドを定義することで配列アクセスのような式が記述可能なクラスを作成できる．5 行目は配列への代入で，これは `update` メソッドの呼び出しと同等である．また，既存クラスに独自のメソッドを定義するような記述も可能である．図 2.2 では関数 `main` の中の式 `3 ** 4` は暗黙

```
1  receiver method arg // receiver.method(arg)
2  1 + 2               // 1.+(2)
3  1 :: Nil            // Nil.::(1)
4  array(3)            // array.apply(3)
5  array(3) = 5        // array.update(3,5)
```

図 2.1. Scala 言語の糖衣構文

```
1  object ImplicitTest {
2
3    implicit class IntWrapper(int : Int){
4      def **(n : Int) = Math.pow(int, n).toInt
5    }
6
7    def main(args: Array[String]): Unit = {
8      println(3 ** 4)
9      // new IntWrapper(3).**(4) = 81
10   }
11
12 }
```

図 2.2. Scala 言語の implicit class

のクラス IntWrapper のメソッド ** を呼び出している．通常は Int クラスに続くメソッド呼び出しは Int メソッドに所属するが，メソッドを使用したスコープ内に暗黙のクラス定義 implicit class が存在し，その引数がレシーバーオブジェクトと同じクラスの場合，暗黙のクラス内のメソッドも呼び出し対象となる．このとき，整数 3 を実引数とした IntWrapper オブジェクトが生成され，そのオブジェクトから ** メソッドが呼び出されることになる．

次に図 2.3 に内部 DSL として Scala のリスト表現を挙げる．1 行目は $1, 2, 3$ の要素を持つリストのデータ構造を作成し， map 関数を 2 度適用している． Nil はクラス List を継承するシングルトンオブジェクトである． :: は List のメソッドである．引数を 1 つ取り，レシーバーオブジェクトのリストの先頭にその引数を追加して返す． Scala 言語には前述の通りレシーバーオブジェクトと引数を明示する場合，メソッド呼び出しのドットと括弧を省略できる糖衣構文があり，またメソッド名がコロンから始まる場合，そのメソッドは右結合として記述できる．よって， 1 :: 2 :: 3 :: Nil は Nil.::(3).::(2).::(1) と等価である．次に， _ はその部分に値を要求するようなラムダ式を構成する． _ + 1 は x => x + 1 の糖衣構文である．1 行目の式の糖衣構文を取り除いた式は 2 行目である．このように Scala 上でリストの操作を自然に表現できるため，リスト表現はホスト言語である Scala の構文を使用して構成された内部 DSL とみなすことができる．

内部 DSL は組み込みではないホスト言語上のライブラリであるため，意味論から導出される最適化をホスト言語のコンパイラには期待できない．例えば図 2.3 では map 関数が毎回新たなリストオブジェクトを生成するため，実行時に不要な中間データ構造が生じている．

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

図 2.3. Scala 言語のリスト表現

```
1  {-# RULES
2  "map/map" forall f g xs.
3            map f (map g xs) = map (f.g) xs
4    #-}
```

図 2.4. Rewrite rules の記述例

2.2　Rewrite rules

関数型言語 Haskell [12] のコンパイラ Grassgow Haskell Compiler [9] には rewrite rules [10] と呼ばれる，コンパイル時にプログラム中の式を規則通りに書き換える機構が実装されている．複数の書き換え規則をあらかじめ記述することで，コンパイル時にコード中の書き換え可能な式を全て書き換える．ある書き換え規則は $l \to r$ の形をとり，式 l のパターンに一致するコード中の式を r に変換する．式 l 中には変数も含まれ，変数には任意の部分式が一致し，r 中の同じ変数の箇所にその部分式が入る．書き換え規則の理論については 2.4 節で詳説する．

書き換え規則によって内部 DSL の最適化をする例を挙げる．図 2.4 は文献 [15] の 7.14 より引用した．このルールは，連続で `map` 関数が使用された時にその引数の関数同士を先に合成させる書き換え規則である．`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．右辺では `f` と `g` が関数合成される．また，図 2.5 はリスト `[1,2,3]` に 2 回の `map` 関数が適用される Haskell 形式の式について，書き換え規則を適用しない場合とする場合での正格評価された際の比較である．始めの 2 行は関数 `f`，`g` の定義である．それ以降の式は `=` が評価を，`->` が書き換えを表す．書き換え規則を適用しない場合，中間データ構造 `[2,4,6]` が生成された後，もう 1 度評価されてリスト `[3,5,7]` となる．一方，書き換え規則を適用する場合は `->` 部分の書き換えがコンパイル時に発生し，実行時には中間データ構造が生成されずに 1 度の評価で計算されるため効率が良い．よってこのような書き換え規則を `map` 関数が定義される内部 DSL に記述することで内部 DSL を使用したユーザーコードを最適化できる．

2.3　オブジェクト指向言語の内部 DSL

オブジェクト指向言語において内部 DSL を実装する場合，メソッドチェーンとクラスの継承が活用される．図 2.6 に Java [16] による座標計算の内部 DSL の例を示す．`Coordinate` は座標を表す抽象クラスであり，それを継承した極座標クラス `PolarCoord` と直交座標クラス `RectCoord` が存在する．`Coordinate` のメソッド `dist` は引数の座標との距離を計算する．

```
 1  f x = x + 1
 2  g x = x * 2
 3
 4  書き換え規則無し
 5  map f (map g [1,2,3])
 6  = map f [2,4,6]
 7  = [3,5,7]
 8
 9  書き換え規則有り
10  map f (map g [1,2,3])
11  -> map (f.g) [1,2,3]
12  = [3,5,7]
```

図 2.5. map 関数の合成規則を適用する例

add は引数の座標との足し算，rotate は原点を中心とした引数の角度分の回転を計算する．内部 DSL ユーザは a.add(b).rotate(c.theta()) のようなコードを記述する．

Rewrite rules はオブジェクト指向言語の内部 DSL の最適化には不十分である．なぜなら，rewirte rules の仕組みでは実行時のオブジェクトの型が判別ができないためである．図 2.6 用の書き換え規則の例を図 2.7 に示す．これは PolarCoord クラスのオブジェクト a と b の 2 点間の距離の二乗を求める場合，余弦定理で計算するように書き換える規則である．点 a,b 間の距離の二乗を求める場合，通常 a,b を直交座標系に揃えて $(a.x - b.x)^2 + (a.y - b.y)^2$ を計算する．書き換え規則左辺はその方法によって計算される．しかし a,b の極座標系上の値が判明している場合，原点と点 a,b からなる三角形における余弦定理 $\overline{ab}^2 = a.r^2 + b.r^2 - 2 \cdot a.r \cdot b.r \cdot \cos|a.theta - b.theta|$ を利用する．これにより a,b を直交座標系に直すオーバーヘッドが削減される．

この書き換え規則は a,b がともに PolarCoord のインスタンスである時のみ適用させたい．しかし，コンパイル時には a と b は Coordinate クラスであることしか分からず，そのサブクラスか否かは判別ができない場合がある．図 2.8 に 2 つの書き換え例を示す．前半は PolarCoord 型変数 p,q について距離の二乗 d をその初期化式で求めている．これは書き換え規則が適用されると，コメントに記述されている通りの書き換えが生じる．後半は Coordinate 型変数 p,q についての距離の二乗の計算である．この場合，p,q のインスタンスは PolarCoord であるが，変数の型は Coordinate であるため，d の初期化式に書き換え規則を適用することはできない．

よってコンパイル時に隠蔽されるサブクラスを指定した場合，書き換え規則を適用させることができない．このように，オブジェクト指向言語上ではサブクラスの情報を書き換え規則が活用できない場合があり，内部 DSL の効率化に制限が生じる．

```
1  abstract class Coordinate {
2    abstract double x();
3    abstract double y();
4    abstract double r();
5    abstract double theta();
6    abstract Coordinate expand(double k);
7    double dist(Coordinate that){
8      return sqrt(pow(x−that.x(),2) + pow(y−that.y(),2));
9    }
10   RectCoord add(Coordinate that){
11     return new RectCoord(x + that.x(), y + that.y());
12   }
13   PolarCoord rotate(double rad){
14     return new PolarCoord(r(), theta() + rad);
15   }
16 }
17
18 class PolarCoord {
19   private double r, theta;
20   ...
21   double x(){
22     return r∗cos(theta);
23   }
24   double y(){
25     return r∗sin(theta);
26   }
27   double r(){ return r; }
28   double theta(){ return theta; }
29   Coordinate expand(double k){
30     return new PolarCoord(r ∗ k, theta);
31   }
32 }
33
34 class RectCoord {
35   private double x, y;
36   ...
37   double x(){ return x; }
38   double y(){ return y; }
39   double r(){
40     return sqrt(pow(x,2) + pow(y,2));
41   }
42   double theta(){
43     return atan2(y,x);
44   }
45   Coordinate expand(double k){
46     return new RectCoord(x ∗ k, y ∗ k);
47   }
48 }
```

図 2.6. 座標計算の内部 DSL

```
1  PolarCoord a,b;
2
3  pow(a.dist(b),2) ->
4    pow(a.r,2) + pow(b.r,2) - 2*a.r*b.r*cos(abs(a.theta-b.theta))
```

図 2.7. 座標計算の内部 DSL 用の書き換え規則

```
1  PolarCoord p = new PolarCoord(1,0);
2  PolarCoord q = new PolarCoord(1,Math.PI);
3  double d = pow(p.dist(q),2);
4  //       = pow(p.r,2) + pow(q.r,2)
5  //             - 2*p.r*q.r*cos(abs(p.theta-q.theta));
6
7  Coordinate p = new PolarCoord(1,0);
8  Coordinate q = new PolarCoord(1,Math.PI);
9  double d = pow(p.dist(q),2);  // cannot rewrite
```

図 2.8. 座標計算の内部 DSL の書き換え適用例

2.4 項書き換え系

項書き換え系 [17][18] とは書き換え規則による項の置換の理論である．項書き換え系は証明支援システム [19][20] などに利用される．図 2.9 に数式の計算過程を項書き換えとして考える例を挙げる．式変形を簡単化する方向に向きづけると，項書き換えとみなすことができる．項間の簡単化する関係 $\rightarrow$ を書き換え関係という．式 $(1+2)\times(3+4)$ は複数の書き換えが可能だが，最終的には 21 に到達する．この性質を合流性という．また，これ以上書き換えができない項 21 を正規形と言う．またこの系には無限の書き換えが存在せず，停止性を持つという．

項書き換え系は rewrite rules の基礎となる理論である．本稿では書き換え規則の説明において項書き換え系の表記を使用する．また，書き換え規則の適用順序の概念である書き換え戦略も使用するため，ここで説明する．以下では項の構造を考慮しない体系である抽象書き換え系，書き換え規則の適用順序を考慮する書き換え戦略を含め，その概論を記述する．

2.4.1 抽象書き換え系

項書き換え系を包含するモデルである，抽象書き換え系について説明する．

定義 2.4.1（抽象書き換え系） 抽象書き換え系は $A = \langle D, \rightarrow \rangle$ である． $\rightarrow$ は集合 D 上の二項関係であり，書き換えという． $(a,b) \in \rightarrow$ であることを $a \rightarrow b$ と表す．

定義 2.4.2 $\rightarrow$ の反射閉包を $\rightarrow^{=}$，推移閉包を $\rightarrow^{+}$，反射推移閉包を $\rightarrow^{*}$ とする．要素 x, y に対してある要素 z が存在し， $x \rightarrow^{*} z \wedge y \rightarrow^{*} z$ を満たすとき， x と y は合流するといい，$x \downarrow y$ とかく．

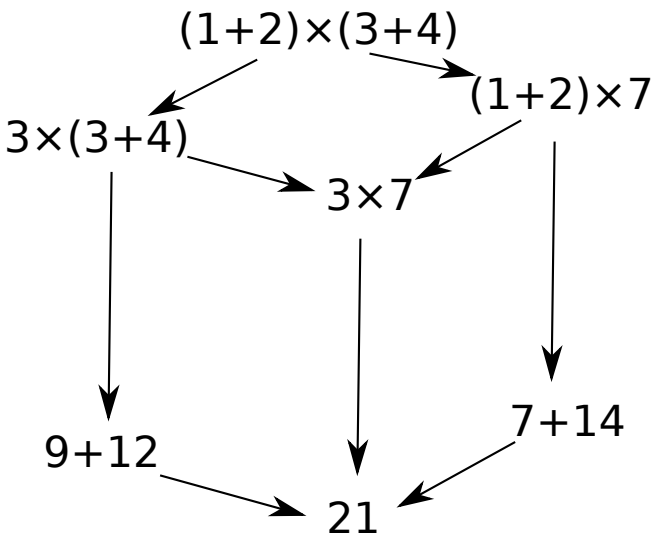

図 2.9. 式の書き換え

定義 2.4.3**（正規形）**　ある要素 $x \in D$ が正規形であるとは，$x \to y$ であるような $y \in D$ が存在しないことを言う．ある要素 x が A のいづれかの正規形に書き換え可能な場合，x は正規形を持つという．正規形の集合を NF_A と表す．

定義 2.4.4**（合流性，停止性）**　$A = \langle D, \to \rangle$ を抽象書き換え系とする．

1. $\forall x, y_1, y_2 \in D$ について，$x \to y_1 \wedge x \to y_2 \Rightarrow y_1 \downarrow y_2$ であるとき，A には弱合流性があるという．
2. $\forall x, y_1, y_2 \in D$ について，$x \to^* y_1 \wedge x \to^* y_2 \Rightarrow y_1 \downarrow y_2$ であるとき，A には合流性があるという．
3. すべての要素が正規形を持つとき，A には弱停止性があるという．
4. 無限の書き換え列 $x_0 \to x_1 \to x_2 \to ...$ が存在しないとき，A には強停止性があるという．このとき，すべての書き換え列は A のいづれかの正規形に到達する．

図 2.10 は弱合流性 (weak confluence) と合流性 (confluence) である．

合流性と停止性について，以下の定理が存在する．

定理 2.4.1**（Newman の補題）**　停止性を持つ抽象書き換え系 A が弱合流性を持つなら，A は合流性を持つ．

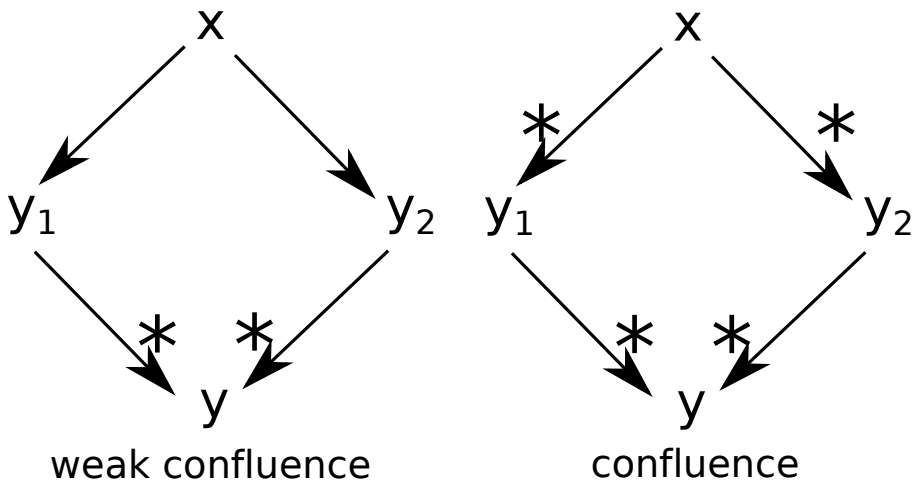

図 2.10. 弱合流性と合流性

2.4.2 項書き換え系

木構造を持った要素である項についての書き換えについて定義する．

定義 2.4.5**（項，線形性）** Σ を関数記号の集合，V を変数の集合とする．項の集合 $T(\Sigma, V)$ は以下で定義される．

1. $V \subseteq T(\Sigma, V)$
2. $f \in \Sigma, t_1, ..., t_n \in T(\Sigma, V)$ のとき $f(t_1, ..., t_n) \in T(\Sigma, V)$

項 t のすべての変数は一度しか出現しない場合，t は線形であるという．

定義 2.4.6**（項書き換え系）** 項 t に出現する変数の集合を $Var(t)$ とする．項 l, r について l が変数でなく，$Var(l) \supseteq Var(r)$ のとき，組 $\langle l, r\rangle$ は書き換え規則といい，$l \to r$ とかく．書き換え規則の集合を E とする．組 $\langle \Sigma, E\rangle$ を項書き換え系という．文脈から明らかなとき，単に E とかく．

項は木構造であり，内部の変数や関数記号は座標を持つ．座標は子要素に左から番号を振ったものの列で表される．根は空記号 ϵ で表される．例えば項 $t = f(e, f(x, i(x)))$ では，図 2.11 に示すように座標 ϵ に項 $f(e, f(x, i))$ があり，座標 1 に部分項 e があり，座標 2.2 に部分項 $i(x)$ がある．以下に座標の定義を示す．

定義 2.4.7**（座標）** 座標とは，正の整数の有限列である．項 t 上の座標の集合 $Pos(t)$ を以下のように定義する．

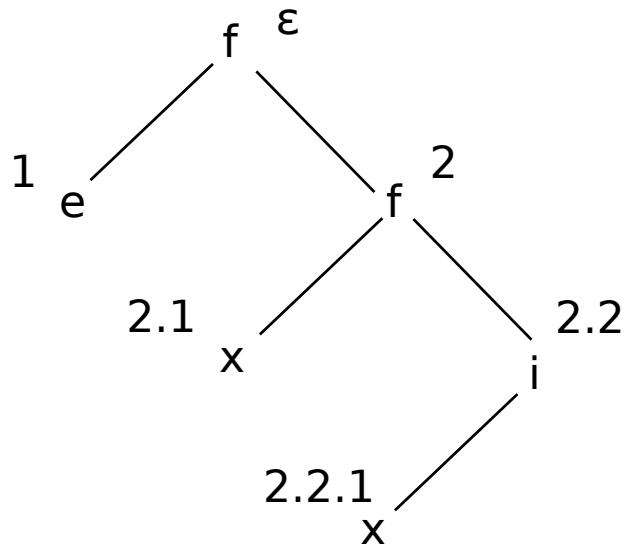

図 2.11. 項 f(e,f(x,i)) の座標

1. $t \in V$ の場合，$Pos(t) = \{\epsilon\}$
2. $t = f(t_1, ..., t_n)$ の場合，$Pos(t) = \{\epsilon\} \cup \{i.p | 1 \leq i \leq n, p \in Pos(t_i)\}$

また，項 t 上の座標のうち，変数を示すもののみの集合を $Pos_V(t)$ ，関数記号を示すもののみの集合を $Pos_\Sigma(t)$ とする．$Pos_\Sigma(t) = Pos(t) \backslash Pos_V(t)$ である．

定義 2.4.8（部分項）　項 t のある座標 $p \in Pos(t)$ に存在する部分項 $t|_p$ を以下で定義する．

1. $p = \epsilon$ の場合，$t|_p = t$
2. $p = i.q \wedge t = f(t_1, ..., t_n)$ の場合，$t|_p = t_i|_q$

定義 2.4.9（置換した項）　項 s, t ，座標 $p \in Pos(t)$ があるとする．項 t の座標 p での部分項 $t|_p$ を項 s で置換した項 $t[s]_p$ を以下で定義する．

1. $p = \epsilon$ の場合，$t[s]_p = s$
2. $p = i.q \wedge t = f(t_1, ..., t_n)$ の場合，$t[s]_p = f(t_1, ..., t_i[s]_q, ..., t_n)$

定義 2.4.10（置換，リデックス）　置換とは，$\sigma(x) \neq x$ であるような関数 $\sigma : V \to T(\Sigma, V)$ である．項の集合 $T(\Sigma, V)$ 上の置換の集合を $Sub(T(\Sigma, V))$ または単に Sub と書く．ある項 t に対してある書き換え規則 $l \to r$ が適用可能であるとは，t のある部分項 t' について $t' = \sigma(l)$ であることをいう．t' は t のリデックスという．

定義 2.4.11（書き換え）　項の集合 $T(\Sigma, V)$ 上の書き換え関係 $\to_E \subseteq T(\Sigma, V) \times T(\Sigma, V)$ は

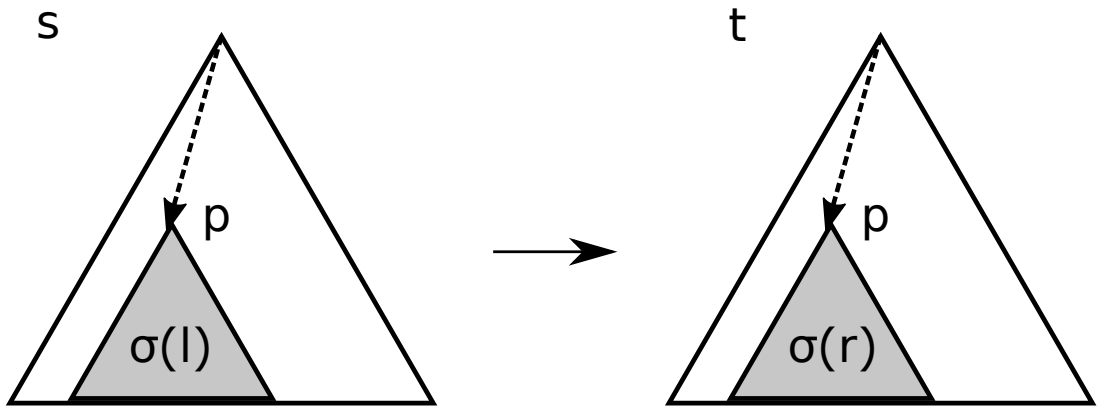

図 2.12. $s \to_E^p t$ の書き換え

以下で定義される．

$$s \to_E t \Leftrightarrow$$
$$\exists (l,r) \in E, p \in Pos(s), \sigma \in Sub.$$
$$s|_p = \sigma(l) \wedge t = s[\sigma(r)]_p$$

書き換えたリデックスの座標を明示したい場合は特に $s \to_E^p t$ などと書く．図 2.12 にこの場合の置換 σ と項 s,t の関係を示す．項 s の座標 p での部分項が $\sigma(l)$ であり，この部分が部分項 $\sigma(r)$ に置換される．

例えば， $\Sigma = \{f,i,e\}, V = \{x\}, E = \{i(x) \to x\}$ とする．項 $f(i(e),e)$ は E の唯一の規則によって書き換え可能であり， $f(i(e),e) \to_E^1 f(e,e)$ となる．この場合，使用された置換は $\sigma = \{x \mapsto e\}$ であり，置換される部分項は $f(i(e),e)|_1 = \sigma(i(x))$ ，置換後の項は $f(e,e) = f(i(e),e)[\sigma(x)]_1$ である．

定義 2.4.12**（左線形性）** すべての書き換え規則 $l \to r \in E$ について l が線形のとき，項書き換え系 E は左線形性を持つという．

定義 2.4.13**（重なり，危険対）** $l_i \to r_i, l_j \to r_j \in E$ とする．全ての $u \in Pos_\Sigma(l)$ について， $i = j \wedge u = \epsilon$ の場合を除いて $\sigma(l_i|_u) = \sigma'(l_j)$ であるような σ, σ' が存在しないとき，項書き換え系 E は重なりが無いという． また，そのような σ, σ' が存在するとき，組 $\langle \sigma(l_j), \sigma'(l_j) \rangle$ を危険対という． $\sigma(l_j) = \sigma'(l_j)$ のとき，自明な危険対という．

例えば，以下 2 つの書き換え規則を持つ項書き換え系を考える．

$$f(g(x)) \to x \tag{2.1}$$
$$g(x) \to g(x) \tag{2.2}$$

項 $f(g(0))$ は書き換え規則 (1)(2) の両方で適用可能である． (1) を使用すると結果は 0 であり，置換は $\sigma_1 = \{x \mapsto 0\}$ である． (2) を使用すると結果は $f(g(0))$ であり，置換は $\sigma_2 = \{x \mapsto 0\}$ である．危険対は $\langle 0, f(g(0)) \rangle$ である．

重なりは単一の書き換え規則からも生じる．以下の書き換え規則

$$f(f(x)) \to x$$

は項 $f(f(f(0)))$ に 2 種類の方法で適用させることができる．この場合の危険対は $\langle f(0), f(0)\rangle$ であり，自明な危険対である．

危険対について以下の定理がある．

定理 2.4.2（危険対の補題）　ある項書き換え系が弱合流性を持つのはその全ての危険対が合流する場合であり，またそのときに限る．

Newman の補題と危険対の補題により，以下が得られる．

定理 2.4.3　ある停止性を持つ項書き換え系が合流性を持つのは全ての危険対が合流する場合であり，またそのときに限る．

定義 2.4.14（直交）　左線形性があり，重なりが無い項書き換え系は直交するという．

直交項書き換え系について以下の定理が知られている．

定理 2.4.4　直交する項書き換え系は合流性を持つ．

定義 2.4.15（左正規）　直交する項書き換え系 E の任意の書き換え規則 $l \to r$ について， l に存在する全ての関数記号が変数より左にある場合， E は左正規であるという．

関数記号集合を $\Sigma = \{f, b, d\}$ とする．

$$f(x, b) \to d \tag{2.3}$$
$$f(b, x) \to d \tag{2.4}$$

(3) は左正規でないが， (4) は左正規である．

2.4.3　書き換え戦略

ある項に存在する複数のリデックスのうち，どのリデックスから書き換えるかを指定する方法を書き換え戦略 [21][22][23][24] という．書き換え戦略には単一ステップ書き換え戦略と複数ステップ書き換え戦略が存在する．

定義 2.4.16（書き換え戦略）　項書き換え系 E のある書き換え戦略を写像 $\mathbb{F} : T(\Sigma, V) \to T(\Sigma, V)$ とする． $\mathbb{F}$ が以下のような写像であれば，単一ステップ書き換え戦略である．

1. $t \in T(\Sigma, V)$ が正規形なら $t = \mathbb{F}(t)$
2. そうでなければ $t \to \mathbb{F}(t)$

また， $\mathbb{F}$ が以下のような写像であれば，複数ステップ書き換え戦略である．

1. $t \in T(\Sigma, V)$ が正規形なら $t = \mathbb{F}(t)$

2. そうでなければ $t \to^+ \mathbb{F}(t)$

定義 2.4.17**（最左最外戦略，並列最外戦略，全置換戦略）** 項 t のあるリデックス t' が他のリデックスの部分項でないとき， t' は最外リデックスである．また t' の部分項に他のリデックスが存在しないとき， t' は最内リデックスである．

1. 項の最も左側にある最外リデックスを書き換えていく戦略を最左最外戦略という．
2. 全ての最外リデックスを並列で書き換える戦略を並列最外戦略という．
3. 全てのリデックスを並列で書き換える戦略を全置換戦略という．

最左最外戦略は単一ステップ書き換え戦略であり，並列最外戦略と全置換戦略は複数ステップ書き換え戦略である．

定義 2.4.18**（正規化戦略）** 書き換え戦略を $\mathbb{F}$ とする．正規形を持つすべての項 t に対して，書き換え列 $\{\mathbb{F}^n(t)|n \leq 0\}$ が正規形を含むなら， $\mathbb{F}$ は正規化戦略である．

定理 2.4.5 書き換え戦略の正規性について以下が成り立つ．

1. 最左最外戦略は左正規の項書き換え系に対する正規化戦略である．
2. 並列最外戦略と全置換戦略は直交項書き換え系に対する正規化戦略である．

第 3 章

実行時書き換え機構

書き換えをプログラム実行時にすることで環境を取得可能とし，オブジェクト言語上の内部 DSL の最適化に用いることを提案する．あるプログラムを実行中，そのプログラムの断片 P が実行される直前に書き換え処理を割り込ませる．書き換え処理は P 実行直前の環境 E を参照し P に対して与えられた書き換え規則を元に書き換えを実行する．そして，環境 E に特化したプログラムの断片 P'_E を書き換え処理後に実行する．この際，書き換え処理は環境 E より得られる値を使用した計算が可能となる．本稿ではこれを書き換え時計算と呼ぶ．書き換え時計算により，環境 E からインスタンスを取得可能なオブジェクトについてはそのサブクラスが判別可能である．よってサブクラスで記述された書き換え規則も適用が可能となり，2 章で説明したオブジェクト指向言語における書き換え規則適用の問題が解決される．以下，提案する実行時書き換えの概要を説明する．

3.1 実行時書き換え

あるプログラムの断片が実行される直前に書き換え処理を割り込ませることで，実行時の環境を利用した書き換えを行う．図 3.1 に実行時書き換えの概念を示す．あるプログラムの一部

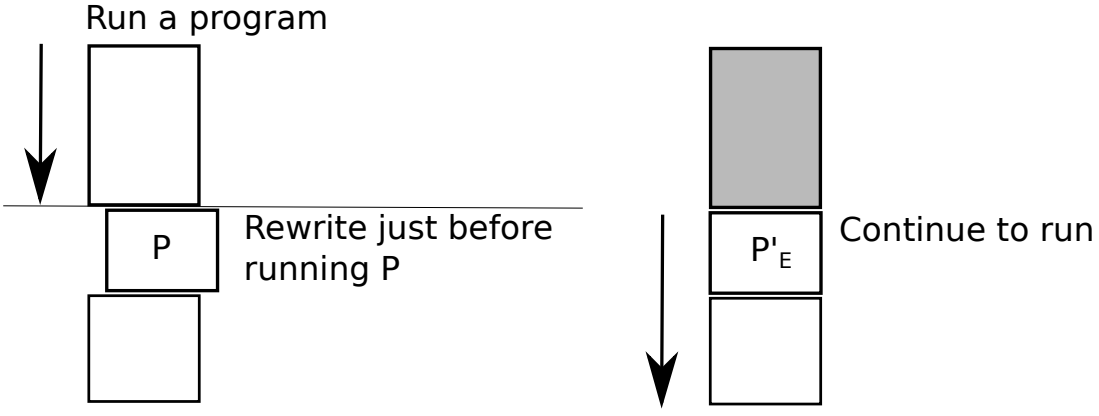

図 3.1. 実行時書き換え

P は書き換え処理対象である．このプログラムを実行中， P の実行直前に書き換えを行う．書き換えはプログラム中に記述された書き換え規則を使用し， P 以前のプログラムの実行より得た環境 E を参照可能である．書き換え処理後，書き換えられたプログラムの一部 P'_E を実行する．

$$
\begin{aligned}
\langle rule \rangle &::= \langle left \rangle \text{ “}\rightarrow\text{” } \langle right \rangle \text{ [“if” } \langle conditions \rangle \text{] } \\
&\quad \{ \text{ “|” } \langle right \rangle \text{ [“if” } \langle conditions \rangle \text{] } \} \\
\langle left \rangle &::= \langle expression \rangle \\
\langle right \rangle &::= \langle expression \rangle \\
&\quad | \langle expression\ contains\ rewriting\text{-}time\ evaluation\ expressions \rangle \\
\langle conditions \rangle &::= \langle expression\ contains\ rewriting\text{-}time\ evaluation\ expressions \rangle
\end{aligned}
\tag{3.1}
$$

書き換え規則は式 3.1 の形をとる．$\langle x \rangle$ は x の非終端記号を表す．ある非終端記号に続く ::= の右がその定義であり，複数のうちの 1 つをとる場合は | で区切られて示される．“x” は非終端記号であり文字列 x を期待する．$[x]$ は x が 0 または 1 個存在することを，$\{x\}$ は x の 0 個以上の繰り返しを表す．ある 1 つの規則は $\langle rule \rangle$ である．$\langle rule \rangle$ は書き換え対象のプログラム断片 P 中の式について， $\langle left \rangle$ に一致し，かつ $\langle conditions \rangle$ を満たせば $\langle right \rangle$ に書き換えることを示している．if 節はなくても良い．$\langle right \rangle$ が複数ある場合は $\langle conditions \rangle$ を満たすものを 1 つ選択する．$\langle left \rangle, \langle right \rangle, \langle conditions \rangle$ は全て書き換え対象のプログラミング言語中の式である．$\langle left \rangle, \langle right \rangle$ は変数パターンを含み，変数パターンは任意の式に一致し展開される．$\langle right \rangle$ は式中に，書き換え時に評価する部分式を含むことができる．書き換え時とは $\langle left \rangle$ に一致する P 中の式が発見され，各変数パターンが束縛された時点である． $\langle conditions \rangle$ は書き換え時に評価される述語である．

想定される $\langle rule \rangle$ とは以下のようなものである．

$$i + j \rightarrow j + i \tag{3.2}$$

$$i + j \rightarrow \underline{i + j} \tag{3.3}$$

$$i + j \rightarrow i \text{ if } \underline{j = 0} \mid j \text{ if } \underline{i = 0} \tag{3.4}$$

i, j は整数を束縛する変数パターンであり，下線は書き換え時に評価する部分式を表す．式 3.2 は加算を入れ替えている．式 3.3 は変数パターン i, j に束縛された値を右辺で書き換え時に計算する．式 3.4 は右辺が 2 つ存在し，i if $\underline{j = 0}$ と j if $\underline{i = 0}$ である．このような書き換え規則を本稿では使用する．

3.2 書き換え時計算

書き換え時に環境を参照すると，その時点までに定義された変数の値で計算することができる．本稿ではこれを書き換え時計算という．図 3.2 は中括弧で囲まれたコード断片を書き換え対象と考えた場合の書き換え例である．書き換え規則は変数パターン i,j を持ち，`i + j` の式についてその計算結果に書き換える．i,j は書き換え直前の環境によって値を参照できる必要

```
1  rule:
2  int i,j;
3  referable i,j;
4  i + j -> 'i + j'
5
6  program:
7  int a = 1;
8  { // rewrite before running
9    int b = 1 + 2; // 3
10   int c = a + 3; // 4
11   int d = a + b;
12   ...
13 }
14 ...
```

図 3.2. 書き換え時に環境を参照する

```
1  rule:
2  List<Integer> list;
3  referable list;
4  list.sum() -> 'list.head()' + 'list.tail()'.sum()
5    if 'list != Nil'
6
7  program:
8  List<Integer> nums = [1,2,3];
9  { // rewrite before running
10   int s = nums.sum(); // 1 + 2 + [3].sum()
11   ...
12 }
13 ...
```

図 3.3. 再帰的な書き換えによりリストを展開する

がある．そのためある変数 x の値が書き換え時環境に存在することを referable x と記述することにしている．規則右辺の引用符は書き換え時計算をするコード断片を表す．書き換え対象のコードは書き換え後，それぞれの式の右のコメントの式になる．変数 b の初期化式は置換 $\sigma = \{i \mapsto 1, j \mapsto 2\}$ で書き換えを試みられる．i,j はともに定数に束縛されるため書き換え時計算が可能であり，3 に書き換えられる．変数 c の初期化式は置換 $\sigma = \{i \mapsto a, j \mapsto 3\}$ で書き換えを試みられる．i に束縛されている変数 a は書き換え対象のコード断片の以前に定義されている．よって書き換え時に環境を参照することで値 1 を取得可能であり，計算の結果 4 に書き換えられる．変数 d の初期化式は置換 $\sigma = \{i \mapsto a, j \mapsto b\}$ で書き換えを試みられる．しかし，変数 b は書き換え対象のコード断片の内部で定義されているため，書き換え時に環境から値を取得することができない．よって書き換え規則は適用できず，書き換えは発生しない．このように書き換え時にそれまでの実行時環境を参照することで，一部の変数を参照し計算することが可能である．

```
1     nums.sum()
2  -> 1 + [2,3].sum()
3  -> 1 + 2 + [3].sum()
```

図 3.4. リスト展開規則の動作

```
1  rule:
2  double eps,other;
3  referable eps,other;
4  eps + other ->
5      'other' if 'eps < 0.1'
6    | 'eps + other'
7
8  program:
9  double a = 0.001;
10 double b = 0.8;
11 { // rewrite before running
12   double ab = a + b; // 0.8
13   double ba = b + a; // 0.801
14   ...
15 }
16 ...
```

図 3.5. 書き換えの分岐

また，書き換え規則を再帰的に利用しリストを展開する例を図 3.3 に示す．整数のリストの要素を合計する式 `list.sum()` について，それをリストの先頭要素とそれ以降のリストの合計との加算にする．メソッド `head`，`tail` がそれぞれリストの先頭要素の取得とそれ以降のリストの取得を表す．環境から参照する必要がある変数は `list` である．if 節は `list` が空（`Nil`）ではないことを記述している．中括弧のコード断片が書き換え対象であり，それ以前にはリスト `nums` の値として 3 つの整数からなるリスト `[1,2,3]` が定義されている．書き換えを受ける式は中括弧内の `nums.sum()` である．書き換えは再帰的に 2 度適用され，`1 + 2 + [3].sum()` となる．詳しい書き換え規則の適用を図 3.4 に示す．1 度目の書き換えは置換 $\sigma = \{list \mapsto [1,2,3]\}$ で試行され，`'list.head()'` の演算結果として `1` を，`'list.tail()'.sum()` の演算結果として `[2,3].sum()` を得る．2 度目の書き換えは置換 $\sigma = \{list \mapsto [2,3]\}$ で試行され，`'list.head()'` の演算結果として `2` を，`'list.tail()'.sum()` の演算結果として `[3].sum()` を得る．このように，書き換え規則の再帰適用によりリストの展開が可能である．このような展開は 5 章 でも利用する．

さらに，複数の右辺から成る書き換え規則を使用し，パターンマッチのように書き換え時に変数パターンが束縛した値によって書き換えを分岐させる例を示す．図 3.5 は小数同士の加算において 1 つ目の項が 0.1 未満なら捨てる例である．変数 `eps`，`double` は小数であり環境より参照可能である．書き換え規則は右辺が 2 通り存在し，`eps` が 0.1 未満なら `other` の値に書き換え，そうでない場合は加算の演算結果の値に書き換える．書き換えを受けるプログ

ラムの例では 0.1 未満の値を持つ変数 a とそうでない値を持つ変数 b が書き換え対象のコード断片以前に定義されている．書き換え時には書き換え対象のコード断片の式のうち a + b が 'other' if eps < 0.1 を満たし置換 $\sigma = \{eps \mapsto a, other \mapsto b\}$ によって，変数 b の値 0.8 に書き換えられる． b + a は b が 0.1 以上なので 'eps + other' が適用され置換 $\sigma = \{eps \mapsto b, other \mapsto a\}$ によって， 0.801 に書き換えられる．このように複数の右辺を持つ書き換え規則を使用することでパターンマッチ構文のように，書き換え時の値に依存した書き換えの分岐が可能となる．

注意として，書き換え時計算で利用する環境の値は書き換え対象のコード断片内で不変であることを仮定している．例えば図 3.5 において ab と ba の初期化式の間に b = 0.5; など b の値を変更する文が挿入されると，書き換えられた ba の初期化式右辺の値と書き換え無しでコード断片を実行した際の値が異なる．よって実行時書き換えを利用するユーザコード内で，書き換え対象のコード断片中で外側の変数の値を変更しないことが守られる必要がある．

3.3　実行時型の判別

3.2 節 で説明した書き換え時計算によって，特定のオブジェクトについてインスタンスの型を判別することが可能となる．図 3.6 は 2.3 節と同様の座標系の内部 DSL の書き換え例である．ある座標 a,b 間の距離の 2 乗を pow(a.dist(b),2) で計算する．この時，もし a,b が極座標クラス PolarCoord のインスタンスを参照していれば，余弦定理を用いた右辺式に書き換えて処理の高速化を図る．条件節は a,b が書き換え時に参照可能であり，さらに PolarCoord のインスタンスであることが記述されている．書き換えを受けるプログラムは書き換え対象のコード断片以前に 3 つの座標 c1,c2,c3 を定義している．c1 のみが RectCoord クラスのインスタンスであり，それ以外は PolarCoord のインスタンスである．書き換え対象のコード断片は 3 点の間の距離をそれぞれ計算し，変数 d1_2，d1_3，d2_3 に格納する．書き換え時にはそれらの変数の初期化式について書き換えを試行するが，書き換えを受けるのは d2_3 の初期化式のみである．このように書き換え時計算を利用して，オブジェクトの書き換え時におけるサブタイプの判別が可能となる．よって実行時に初めて型が決定する式に書き換えを適用することが可能であり，内部 DSL のさらなる効率化が期待できる．

書き換え時計算と同様の注意として，書き換え対象コード内で初めてインスタンスの型が判明する場合に，その型を利用する書き換え規則は適用できない．図 3.6 の場合，c1,c2,c3 の値は必ず書き換え対象コード外で代入される必要がある．そうでない場合，書き換え時点では c1,c2,c3 の型は Coordinate であることしか利用できないため，PolarCoord を対象とした書き換え規則を適用することはできない．このように実行時書き換えには書き換え対象コード内における特定の変数の値の不変性を要求する制限が存在する．

```
1  rule:
2  Coordinate a,b;
3  pow(a.dist(b),2) ->
4    pow(a.r,2) + pow(b.r,2) - 2*a.r*b.r*cos(abs(a.theta-b.theta))
5    if isReferable(a,b)
6      && a instanceof PolarCoord
7      && b instanceof PolarCoord
8
9  program:
10 Coordinate c1 = new RectCoord(0,1);
11 Coordinate c2 = new PolarCoord(1,0);
12 Coordinate c3 = new PolarCoord(1,Pi);
13 { // rewrite before running
14   double d1_2 = pow(c1.dist(c2),2);
15   double d1_3 = pow(c1.dist(c3),2);
16   double d2_3 = pow(c2.dist(c3),2); // enable to rewrite
17 }
```

図 3.6. 実行時型の判別によるサブタイプを区別した書き換え規則の適用

第 4 章

実装

提案手法を Java 上で実装する．実装にはバイトコードから抽象構文木を構成するライブラリ Bytespresso [25] を使用する．書き換え対象のメソッドが呼び出される際に定義元のクラスファイルを参照し，メソッド本体のバイトコードから抽象構文木を作成する．抽象構文木を書き換え規則を元に変換し，再度 Java コードに直す．そして，Java コードを再コンパイルしクラスローダによって読み込み実行する．

4.1 Bytespresso

Bytespresso [25] は Java バイトコードから抽象構文木を構成するためのライブラリである．Java で記述されたメソッドを C [27] 言語や CUDA [28] で実行するシステム Bytespresso-C の一部であり，Bytespresso-C は Java コードの実行の一部を GPU などの外部ハードウェアに任せる目的で設計されている．図 4.1 は Bytespresso-C によってラムダ式の実行を C 言語で行う例である．メソッド `invoke` の引数は実行時にクラスファイルからバイトコードを読み取られ，C 言語に変換される．C のソースコードは C コンパイラでコンパイル後実行される．ラムダ式内部の `Util.print` は Bytespresso-C が提供する出力用の関数である．実行すると文字列 `sample` が出力される．

Bytespresso で特定のメソッド本体のバイトコードを指定し，その抽象構文木を取得する

```
1 import javassist.offload.javatoc.DriverException;
2 import javassist.offload.javatoc.StdDriver;
3 import javassist.offload.lib.Util;
4
5 public class BytespressoSample {
6   public static void main(String[] args) throws DriverException {
7     new StdDriver().invoke(() -> Util.print("sample").println());
8   }
9 }
```

図 4.1. Bytespresso-C でラムダ式を実行する例

```
1 ClassPool cp = ClassPool.getDefault();
2 CtClass cc = cp.get(clazz.getName());
3 CtMethod cm = cc.getDeclaredMethod(methodName);
4
5 Reifier reifier = new Reifier(cm, arguments);
6 Snapshot image = reifier.snap();
```

図 4.2. Bytespresso による抽象構文木の抽出

```
1 public static class Snapshot{
2   public final Function function;
3   public final ClassTable classTable;
4   public final FunctionTable<...> functionTable;
5   public final HashSet<Object> objects;
6   Snapshot(...){...}
7 }
```

図 4.3. Snapshot クラス

には図 4.2 のように記述する．前半 3 行は javassist ライブラリを使用してコンパイル時のメソッドを表すクラス CtMethod のインスタンスを取得している．具体的にはクラスプールのインスタンス cp からコンパイル時のクラスを表すクラス CtClass のインスタンスを取得する．クラス名は clazz.getName() で指定している．そして cp からメソッド名 methodName をしていして CtMethod インスタンス cm を取得している．後半 2 行は Bytespresso による抽象構文木の取得である．レイフィケーションを表すクラス Reifier のコンストラクタに CtMethod インスタンス cm 及びメソッド呼び出し時の実引数 arguments を渡す．そして Reifier インスタンスのメソッド snap() を呼び出し，抽象構文木を表すクラス Snapshot のインスタンスを得る．

Bytespresso で抽出した抽象構文木は図 4.3 に示すようなクラス Snapshot に格納されている．Snapshot は Reifier クラス内部にある静的クラスである．保存されている情報は直接指定したメソッド本体と，そのメソッドから呼び出されている全てのメソッド本体の抽象構文木である．さらに参照されたクラス情報も保存される．クラス Function が直接指定されたメソッドの抽象構文木を持つ．クラス ClassTable は参照されたクラスの情報を持つ．クラス FunctionTable は副次的に呼び出された全てのメソッドの抽象構文木を持つ．HashSet<Object> はメソッドより呼び出されるオブジェクト集合である．Function はフィールドにメソッド本体を示すクラス Body のインスタンスを持つ．Body は命令ブロックを表すクラス Block のリストであり，Block はさらに個別のバイトコード命令を表す抽象クラス ASTree のリストである．よって，Snapshot の function フィールドをたどることで直接指定したメソッド本体の抽象構文木オブジェクトを取得できる．

4.2　実行時書き換え機構

実行時書き換え機構全体の処理は図 4.5 のようになる．書き換え対象となるクラス B のメソッド m のバイトコードは Bytespresso によってメソッドが定義されているクラスファイル B.class から読み取られる．読み取られたメソッド m の抽象構文木は書き換えられた後，新しく生成されるクラス Generated のメソッドとして Java コードに直され，実行される．以下に詳しい実装を述べる．

実行時書き換え機構ライブラリの本体部分のクラス図を図 4.12 に示す．クラス RewriteTool は書き換え対象となるメソッドの抽象構文木（Bytespresso のクラスである Reifier.Snapshot），最終的に生成される Java コードをフィールドに所有する．ユーザが RewriteTool インスタンス作成後，メソッド rewrite，compile，run を順に呼び出して書き換え対象メソッドを実行することを想定している．メソッド rewrite の引数の Pattern クラスは後述する，実行時書き換え機構ライブラリが提供する書き換え規則定義用内部 DSL である．メソッド rewrite ではクラス Translator を利用して抽象構文木を書き換える．クラス Translator 内部で使用されるメソッド translate はさらにノードの一致と変数パターンの束縛を担当するクラス Matcher と，書き換え規則右辺のインスタンスを生成するクラス InstanceMaker を利用する．Translator などの 3 つのクラスは全てインタフェース Visitor を継承している．Visitor は Bytespresso 内に存在し，抽象構文木のノードオブジェクトに対するビジターパターン [26] を提供している．クラス RewriteTool のメソッド compile はクラス ClassCodeGenerator，クラス ClassCompiler を順に利用しメソッドの再コンパイルを実行する．クラス ClassCodeGenerator は書き換え対象メソッドの抽象構文木を Java コードに直し，新しいクラス Generated 内のメソッドとして再定義する．クラス ClassCompiler はソースコードを文字列から読み込み，コンパイル後クラスローダにロードさせる役割を持つ．以上が実行時書き換え機構ライブラリの実装の概要である．

実行時書き換え機構ライブラリを使用したメソッド呼び出しは図 4.4 のようになる．クラス RewriteTool のコンストラクタにクラス，メソッド名，実引数を指定する．クラス Pattern のオブジェクトをメソッド rewrite の引数として渡し書き換えを実行する．その後，メソッド compile，run を呼び出しコンパイルして実行する．RewriteTool の型パラメータはメソッド run の返り値の型である．

実行時書き換え機構ライブラリは図 4.6 のクラス図で示されるような書き換え規則定義のための内部 DSL を提供する．抽象クラス Pattern 及びそのサブクラス Left，LeftRight は書き換え規則を定義するためのメソッドを提供し，そのインスタンスは書き換え規則を表す．Pattern.left(...).right(...) として 1 つの書き換え規則を定義する．クラス Node は規則で使用する抽象構文木のノードを表し，特にサブクラス VarNode は変数パターンのノードを表す．クラス Nodes は Node インスタンスを作成するためのユーティリティメソッドを提供する．クラス PNode は書き換え時計算を表す特殊なノードを表す．クラス Bind は変数束縛情報，クラス Condition は書き換え時の条件を表し，共に Pattern のフィールドに使

```
1 public static int rewritedRound(float f){
2   RewriteTool<Integer> rewriteTool =
3     new RewriteTool<>(java.lang.Math.class,
4       "round", new Object[]{f});
5   Pattern pattern = ...;
6   rewriteTool.rewrite(pattern);
7   rewriteTool.compile();
8   return rewriteTool.run();
9 }
```

図 4.4. 実行時書き換え機構ライブラリを使用したメソッド呼び出し

用される．

書き換え規則定義用内部 DSL による書き換え規則の定義例を図 4.7 に示す．この例では図 3.3 と同様にリストへの map による関数適用を要素ごとに展開する．始めの数行は使用する変数パターンを定義している．メソッド var は書き換え規則中の変数を表す構文木のノード VarNode を返す，クラス Nodes の静的メソッドである．書き換え規則は Pattern クラスを始点としてメソッドチェーンで記述する．書き換え規則両辺を表す構文木を引数にとる left，right，新たな規則中の変数の値を設定する where，追加条件を設定する _if の 4 つのメソッドを使用して規則を定義している．Node クラスのメソッド call は引数で指定された名前のメソッドを呼び出す．VarNode クラスのメソッド rwtCall は書き換え時計算に相当し，レシーバーオブジェクトが取得した部分式の実行時の値に，引数の関数を適用する．メソッド is も同様の書き換え時計算をするが，関数適用の結果が真の場合のみ書き換えの展開を行う．

Bytespresso によって得られた抽象構文木を書き換え規則の通りに変換する．今回実装した書き換え戦略のアルゴリズムを図 4.8 に示す．書き換え対象の抽象構文木は根ノードから順に深さ優先探索され，抽象構文木中の式を表すノードが与えられた書き換え規則左辺と一致した場合，右辺へと書き換えられる．全てのノードは書き換え規則が適用されなくなるまで繰り返し試行される．複数の書き換え規則が与えられた場合，次の規則は再び抽象構文木の根ノードから順に深さ優先探索される．

書き換えられた抽象構文木を Java コードに再変換する．ここで抽象構文木はバイトコードから生成されたものであり Java コードと 1 対 1 対応はしていない．そのため実装では図 4.9，図 4.10，図 4.11 に示すような変換を行なった．図はそれぞれ Java 標準ライブラリの java.lang.Math クラスのメソッド round(float a) の本来の定義，抽象構文木，その再変換結果の一部である．図 4.9 の変数 initBits，biasedExp，shift，r は図 4.10 では v1，v2，v3，v4 が対応している．FloatConsts.EXP_BIT_MASK などの図 4.9 中の定数は 図 4.10 では数値となっている．また shift 変数の定義式の右辺など，定数同士の演算は計算済みとなっている．バイトコードの抽象構文木には n:{...} で表される番号つきの命令ブロックが存在し， goto n で n 番目の命令ブロックに処理が移動する．図 4.9 の if((shift & -32) == 0) ... に対応する部分は図 4.10 では

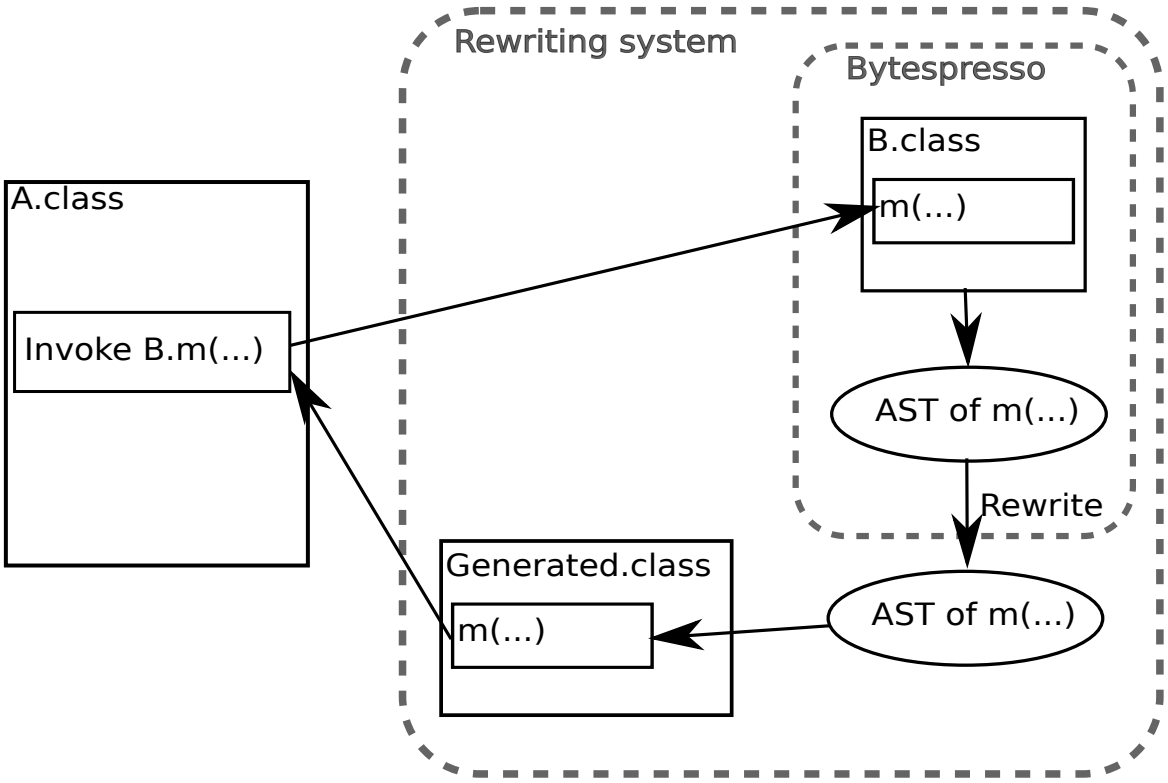

図 4.5. 実行時書き換え機構の処理の流れ

`if ((v3 & -32) != 0) goto 2` であり， Java コードの else 節が 2 番の命令ブロックに対応していることがわかる．このように， Java コードの if 文はバイトコードでは `goto` 文と命令ブロックに変換されている．そのため，再変換においては図 4.11 のようにメソッド本体で制御構造として利用する while 文と switch 文を追加し，命令ブロックの開始地点に case 文を置いた．命令ブロックの番号が case 文の引数と対応し，goto 命令の箇所では switch 文の引数である変数 `BODY_FLAG` の値を変更した後に continue 文で while 文の先頭に遷移させ，再び switch 文にて分岐させる構造とした．

Java コードに再変換された書き換え対象メソッドは新しく生成されるクラス `Generated` が継承するインタフェース `GenClassAccessor` を通じて呼び出しが可能となる．`GenClassAccessor` は `run` メソッドを持つインタフェースであり，`Generated` では実装される `run` メソッドの本体で書き換え対象メソッドが呼び出されている．

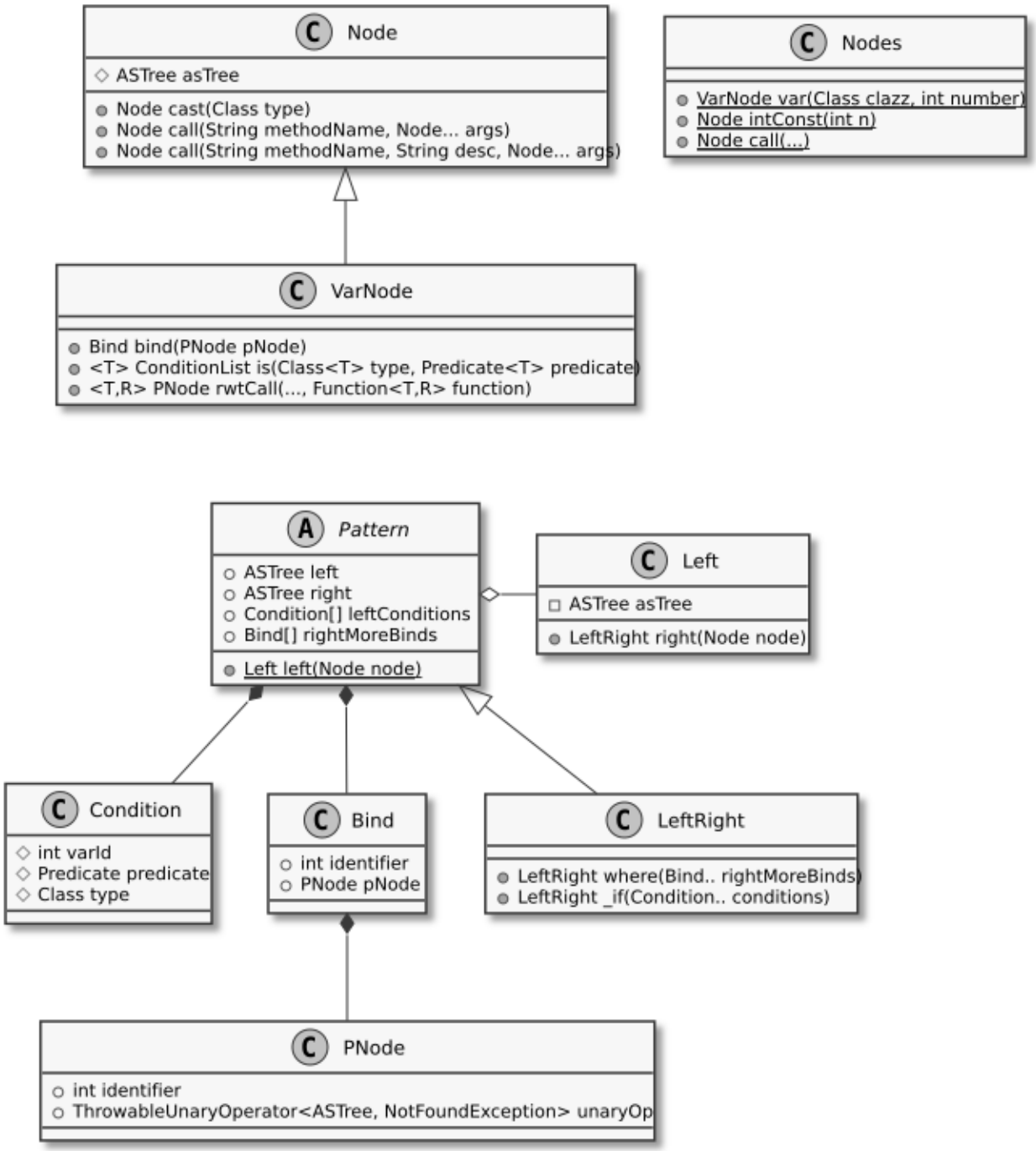

図 4.6. 書き換え規則定義用 DSL のクラス図

```
1  // variables
2  Node.VarNode list = var(List.class, 0);
3  Node.VarNode f = var(Function.class, 1);
4  Node.VarNode g = var(Function.class, 2);
5  Node.VarNode tail = var(List.class, 3);
6  Node.VarNode head = var(Object.class, 4);
7
8  // list.map(f) -> 'list.tail()'.map(f).prepend(f.apply('list.head()'))
9  //   if '!list.isEmpty()'
10 Pattern expandList = Pattern.left(
11   list.call("map",f)
12 ).right(
13   tail.call("map",f).call("prepend",f.call("apply", head))
14 ).where(
15   tail.bind(list.rwtCall(List.class, List.class, List::tail)),
16   head.bind(list.rwtCall(List.class, Object.class, List::head))
17 )._if(
18   list.is(List.class, lst -> !lst.isEmpty())
19 );
```

図 4.7. 実行時書き換え機構ライブラリを使用した map 関数を展開する例

1: **function** TRANSLATE($node$)
2:　　**for** $i \leftarrow 0$ until $rules.length$ **do**
3:　　　　**if** $rules[i].left$ match with $node$ **then**
4:　　　　　　$node \leftarrow$ SUBSTITUTE($node$)
5:　　　　　　$i \leftarrow 0$
6:　　**for** $i \leftarrow 0$ until $node.length$ **do**
7:　　　　$subNode \leftarrow node.$ GET(i)
8:　　　　$subNode' \leftarrow$ TRANSLATE($subNode$)
9:　　　　$node.$ SET(i, $subNode'$)
10:　　**return** $node$

図 4.8. 実行時書き換え機構が使用する書き換え戦略のアルゴリズム

```
1  public static int round(float a) {
2      int intBits = Float.floatToRawIntBits(a);
3      int biasedExp = (intBits & FloatConsts.EXP_BIT_MASK)
4              >> (FloatConsts.SIGNIFICAND_WIDTH - 1);
5      int shift = (FloatConsts.SIGNIFICAND_WIDTH - 2
6              + FloatConsts.EXP_BIAS) - biasedExp;
7      if ((shift & -32) == 0) {
8          int r = ((intBits & FloatConsts.SIGNIF_BIT_MASK)
9                  | (FloatConsts.SIGNIF_BIT_MASK + 1));
10         if (intBits < 0) {
11             r = -r;
12         }
13         return ((r >> shift) + 1) >> 1;
14     } else {
15         return (int) a;
16     }
17 }
```

図 4.9. Math.round メソッドの定義

```
1  int Math_round_0_0(v0:float ){
2    local vars: v1:int v2:int v3:int v4:int
3    0:{reset:
4      v1 = java.lang.Float#floatToRawIntBits(v0)
5      v2 = ((v1 & 2139095040) >> 23)
6      v3 = (149 - v2)
7      if ((v3 & -32) != 0) goto 2
8      v4 = ((v1 & 8388607) | 8388608)
9      if (v1 >= 0) goto 1
10     v4 = -v4
11   }
12   1:{reset: v4
13     return (((v4 >> v3) + 1) >> 1)
14   }
15   2:{reset:
16     return ((int)v0)
17   }
18 }
```

図 4.10. Bytespresso によって生成された Math.round メソッドの抽象構文木の文字列表現

```
1  public static int round(float v0) {
2    ...
3    BODY_FLOW: while(true){
4      switch(BODY_FLAG){
5        case 0:
6        ...
7        if (( v3 & -32 ) != 0) {
8          BODY_FLAG = 2; continue BODY_FLOW;
9        };
10       ...
11       if (v1 >= 0) {
12         BODY_FLAG = 1; continue BODY_FLOW;
13       };
14       ...
15       case 1:
16       return ...
17       case 2:...
18       return ...
19     }
20     break;
21   }
22   DUMMY_RETURN: return 0;
23 }
```

図 4.11. Math.round メソッドの Java コードへの再変換結果

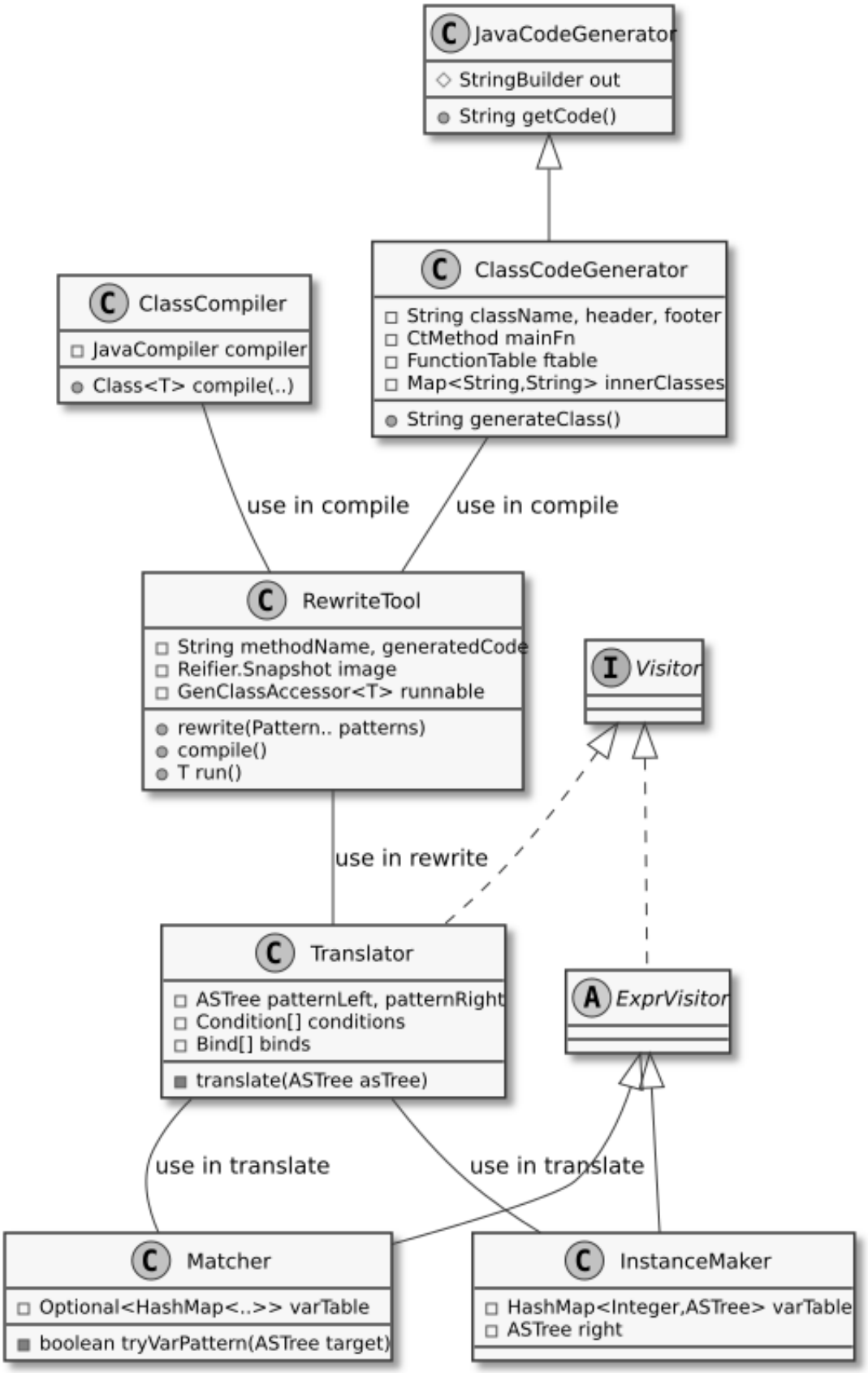

図 4.12. 実行時書き換え機構ライブラリのクラス図

第 5 章

実験

行列の内部 DSL を使用したメソッドについて，書き換え機構によって実行速度が上がることを示し，提案手法の有効性を確認する．ランダムに生成された正方行列の積を複数回実行する．このとき積を取る一方の行列が疎行列の場合のみ，疎行列に特化した行列積メソッドに式を書き換える．通常の定義通りの行列積メソッドで全て実行した場合，疎行列に特化した行列積メソッドで全て実行した場合，書き換え処理を挿入した場合を比較し，書き換え処理によって実行時間が削減されることを示す．実験環境は Mac OS X 10.12.2，1.8 GHz Intel Core i7，メモリ 4GB である．

5.1 行列の内部 DSL

行列の内部 DSL を図 5.1 に示す．`Matrix` クラスは $n \times n$ の正方行列を表し，要素を表すフィールド `array` を持つ．以下にメソッドの説明を記述する．

`Matrix generate(int n, double density)`
: 0 でない要素が `density` の割合で存在する $n \times n$ 行列をランダムに生成する．すべての要素 e は $0 \leq e < 100$ の整数である．

`Matrix mul(Matrix that)`
: 引数の行列 `that` との行列積を計算する．行列積の定義通りのアルゴリズムで実装されている．

`Matrix halfSparseMul(Matrix that)`
: 引数の行列 `that` が疎である場合に高速に計算可能なアルゴリズム list of lists によって，行列積を求める．

`double density()`
: 行列の 0 でない要素の割合を求める．

`halfSparseMul` のアルゴリズムを図 5.3 に示す．二次元配列 $array$ 及び $that.array$ が行列を表す．前半のループにおいて $array$ より 0 でない要素のみを収集したリストのリスト $elems$ を作成する．$elems$ は整数 $i, 0 \leq i < n$ と， i 行目に関する 0 でない要素のリスト

row の対 $\langle i, row\rangle$ のリストである．row は整数 $j, 0 \leq j < n$ と，ある行 i における 0 でない j 列目の要素 $array[i][j]$ の対 $\langle j, array[i][j]\rangle$ である．このリストのリスト $elems$ を後半のループで展開し，$that.array$ と行列積を計算する．その際，最内ループはリスト row の長さ分のみ実行されるため，$array$ が疎である場合は行列積の定義通りの実装より高速である．

また本実験で行列の内部 DSL と共に使用するリストを表すインタフェース `List` を図 5.2 に示す． `List` インタフェースは Java 標準ライブラリの `List` クラスと異なり関数型言語に準拠している．`List` インタフェースを継承するクラスには `Cons` と `Nil` が存在し，`Cons` はフィールドに単一の要素 `head` とそれ以降の要素のリスト `tail` を持つリストである．`Nil` はリストの終端を表すシングルトンオンブジェクトである．以下にインタフェース `List` の主要なメソッドの機能を説明する．

`List<T> prepend(T e)`
: リストの先頭に `T` 型の要素 `e` を加えた新たなリストを生成する．

`T get(int i)`
: `i` 番目の要素を返す．

`List<U> map(Function<? super T, ? extends U> function)`
: 関数 `function` をリストのすべての要素に適用し，新たなリストを生成する．

`List<Tuple2<T,U>> zip(List<? extends U> that)`
: レシーバーのリストと引数のリスト `that` について，それぞれのリストの先頭から要素を 1 つずつ取り出し，タプル `Tuple2<T,U>` を作成する．これを繰り返し，すべてのタプルを要素とした新たなリストを返す．

`List<T> drop(int n)`
: リストの先頭から `n` 個の要素を捨てたリストを返す．

`boolean isEmpty()`
: リストが空なら `ture` を返す．

`int length()`
: リストの長さを返す．

`T head()`
: リストの先頭要素を返す．

`List<T> tail()`
: リストの先頭以外の要素を持つリストを返す．

また，以下の 2 つは静的メソッドである．

`List<T> of(T... ts)`
: ts を要素にもつリストを生成する．

`Nil<T> empty()`
: 空のリストを生成する．

```
1  public class Matrix {
2    public int[][] array;
3    public int n;
4
5    ...
6
7    public static Matrix generate(int n, double density) {...}
8
9    public Matrix mul(Matrix that) {...}
10
11   public Matrix halfSparseMul(Matrix that) {...}
12
13   public double density() {...}
14 }
```

図 5.1. Matrix クラス

この実装は Java の関数型ライブラリである Javaslang [29] に基づいている．関数型仕様のリストを使用することでリストの関数適用式を各要素毎に展開することが可能となる．

5.2　最適化実験

既知である行列のリスト A とランダムに生成された行列のリスト B の各要素同士を積する．その時 A 中の行列が疎行列の場合に疎行列専用の積メソッドに書き換える最適化を行い，実行速度を比較する．書き換え対象となるメソッド muls を図 5.4 に示す．行列のリスト A は muls の引数であり，書き換え時に値を参照可能である．行列のリスト B は muls 内でランダムに生成される． A と B のリスト長は等しい．このメソッドの返り値は A と B の要素をそれぞれ積することで作成される新たな行列のリストである．muls に適用する書き換え規則列を図 5.5 に示す．規則 1，2 は zip および map メソッドの展開である．規則 1 は list2 に対して drop(0) を呼び出すのみであり，規則 2 を適用するための前処理である．規則 2 は list1 を先頭とそれ以外の要素に分割し，また list2 を i 番目の要素とそれ以降の要素に分割する．そして list1 の先頭要素と list2 の i 番目の要素のタプルに対して関数 f を適用したものを prepend メソッドによって，zip メソッドの返り値であるリストの先頭に追加する．規則 2 はその if 節に記述されている通り，list1 に要素がある限り適用可能であり，再帰的に適用することを想定している．以上により規則 1，2 は zip と map メソッドの処理をリストの個別の要素について展開するものである．展開された式に規則 3 を適用する．規則 3 は規則 1，2 によって展開された関数適用式について，タプルの 1 番目の要素である行列 p の密度が変数 bound の値未満なら halfSparseMul メソッドの式に書き換える．if 節の条件式の計算は書き換え時であり，書き換え時に密度の判定を実行している．規則列が適用される様子を図 5.6 に示す．規則 1 適用後，リスト A の要素が無くなるまで繰り返し規則 2 が適用される．リスト展開後生じたメソッド prepend の引数部分に規則 3 が試される．この例では A[1] を疎行

```
1  public interface List<T> {
2
3    default List<T> prepend(T e){...}
4
5    default T get(int i){...}
6
7    default <U> List<U> map(
8      Function<? super T, ? extends U> function){...}
9
10   default <U> List<Tuple2<T,U>> zip(
11     List<? extends U> that){...}
12
13   default List<T> drop(int n){...}
14
15   default boolean isEmpty(){...}
16
17   int length();
18
19   T head();
20
21   List<T> tail();
22
23   static <T> List<T> of(T... ts){...}
24
25   static <T> Nil<T> empty(){...}
26
27   final class Nil<T> implements List<T> {...}
28
29   class Cons<T> implements List<T> {...}
30 }
```

図 5.2. List インタフェース

列と仮定したため `p = A[1]` のとき規則 3 の条件節 `p.density() < bound` を満たし，書き換えられる．以下の実験ではリスト長を 50，行列は全て 500×500 とする．ある行列が疎行列である基準は，0 でない要素が全体の 0.01 以下とする．行列のリスト `A` には 0 でない要素が 0.005 の割合で存在する行列（疎行列）と，0.9 の割合で存在する行列（密行列）がある割合 $r : 1 - r$ で含まれている．r を 0 から 0.2 ずつ増やし 1.0 までの実行速度を計測した．

行列のリスト `A` の疎行列の割合と実行速度の関係を図 5.7 に示す．横軸は行列のリスト `A` に含まれている疎行列の割合 r で，縦軸は実行時間である．normal は全ての積をメソッド `mul` で実行したもの，list of lists は全ての積をメソッド `halfSparseMul` で実行したもの，hybrid は書き換え処理をしたものである．hybrid には書き換え対象メソッド `muls` の書き換え時間及びコンパイル時間を含める．書き換え処理によって疎行列の積と密行列の積を適切に分岐させることで，広範囲の r の値において通常の実装より実行時間の低下が確認できる．

```
1: elems ← empty
2: for i ← 0 until n do
3:     row ← empty
4:     for j ← 0 until n do
5:         if array[i][j] ≠ 0 then
6:             row. ADD(⟨j, array[i][j]⟩)
7:     if row ≠ empty then
8:         elems. ADD(⟨i, row⟩)
9: newArray ← empty
10: for ⟨i, row⟩ ← elems do
11:     for k ← 0 until n do
12:         sum ← 0
13:         for ⟨j, value⟩ ← row do
14:             sum ← sum + value × that.array[j][i]
15:         newArray[i][k] ← sum
```

図 5.3. 引数が疎行列の場合に高速に積を実行するメソッドのアルゴリズム

```
1    public static List<Matrix> muls(List<Matrix> A) {
2      List<Matrix> B = List.empty();
3      for (int i = 0; i < A.length(); i++) {
4        B = B.prepend(Matrix.generate(A.get(0).n, 1));
5      }
6      return A.zip(B).map(tpl -> tpl._1.mul(tpl._2));
7    }
```

図 5.4. 最適化実験にて書き換え対象となるメソッド

```
1  List list1, list2;
2  Function f;
3  int i;
4  Matrix p, q;
5
6  1. list1.zip(list2).map(f) -> list1.zip(list2.drop(0)).map(f)
7  2. list1.zip(list2.drop(i)).map(f) ->
8       'list1.tail()'.zip(list2.drop('i + 1')).map(f)
9         .prepend(f.apply(Tuple.of('list1.head()', list2.get(i))))
10      if '!list1.isEmpty()'
11 3. f.apply(Tuple.of(p,q)) -> p.halfSparseMul(q)
12      if 'p.density() < bound'
```

図 5.5. 最適化実験にて使用される書き換え規則列

```
1  lambda = tpl -> tpl._1.mul(tpl._2) とする.
2  A と B のリスト長を 3 とする.
3  リスト X の i 番目の要素を X[i] と表す.
4  リスト X の i 番目から j 番目までの部分リストを X[i,j] と表す.
5  リスト A について A[1] のみ疎行列とする.
6
7  A.zip(B).map(lambda)
8  -> A.zip(B.drop(0)).map(lambda)                              (1)
9  -> 'A.tail()'.zip(B.drop('0+1'))                             (2)
10       .prepend(lambda.apply(Tuple.of('A.head()', B.get(0))))
11   = A[1,2].zip(B.drop(1))
12       .prepend(lambda.apply(Tuple.of(A[0], B.get(0))))
13 -> ...
14 -> Nil.zip(B.drop(3))                                        (2)
15       .prepend(lambda.apply(Tuple.of(A[2], B.get(2))))
16       .prepend(lambda.apply(Tuple.of(A[1], B.get(1))))
17       .prepend(lambda.apply(Tuple.of(A[0], B.get(0))))
18 -> Nil.zip(B.drop(3))                                        (3)
19       .prepend(lambda.apply(Tuple.of(A[2], B.get(2))))
20       .prepend(A[2].halfSparseMul(B.get(2)))
21       .prepend(lambda.apply(Tuple.of(A[0], B.get(0))))
```

図 5.6. メソッド中の式に書き換え規則列を順に適用する

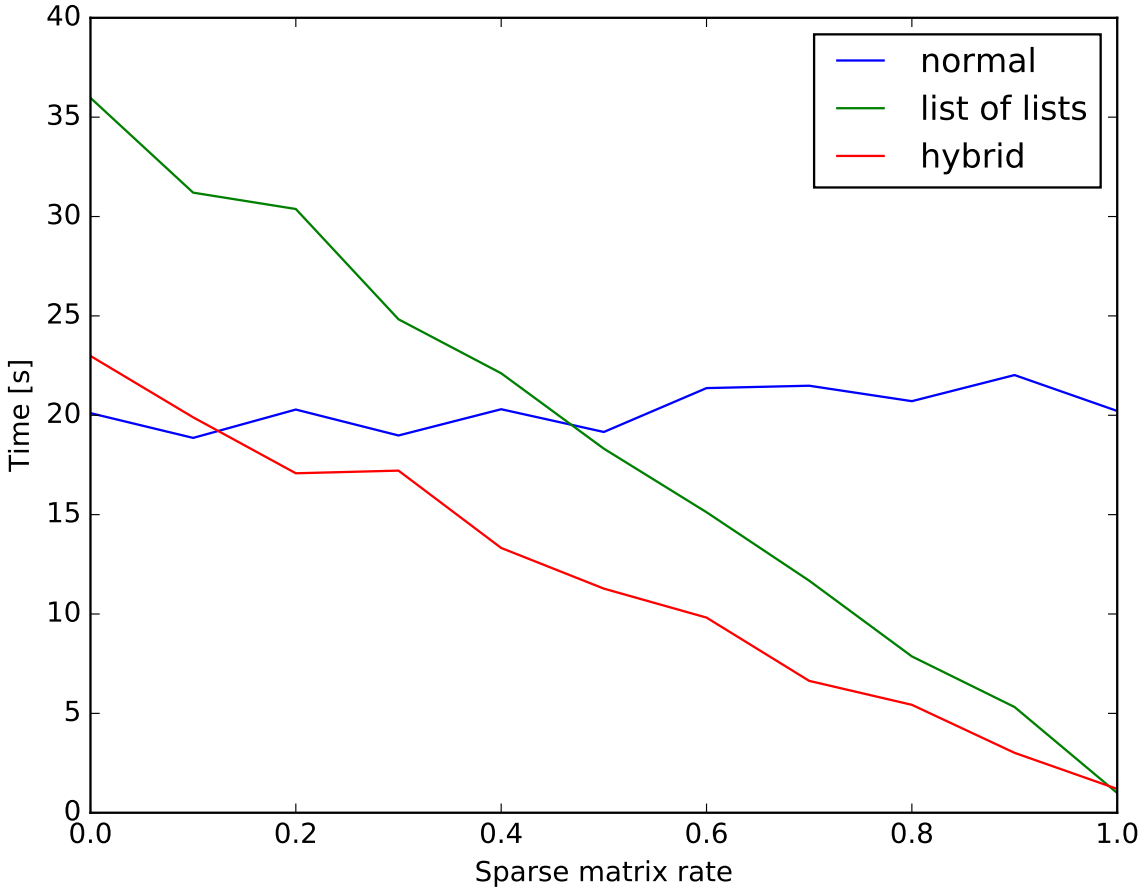

図 5.7. 疎行列の割合と実行速度

第 6 章

関連研究

6.1 Lisp マクロ

Common Lisp [30] のマクロ関数はコンパイル時に評価され Lisp の式を生成する．図 6.1 は通常の関数によるべき乗の定義，図 6.2 はマクロ関数によるべき乗の定義である．共に引数 n, x を持ち，返す値は x^n である．n が 0 でない場合に $x \times pow(n-1, x)$ を計算する，再帰的定義をしている．図 6.2 で使用されている記号について説明する．Lisp マクロでは式に記号を付加することで評価を避ける指定等を行う．クオートは付加されている式内部の評価を行わない．データ構造としてのリストを表現することに使用され，例えば `‘(1 2 3)` を評価するとリスト `(1 2 3)` が返る．バッククオートも同様に付加された式の内部の評価を行わないが，その式内部でコンマがある場合，コンマの付加された式のみは評価する．例えば $a = 1, b = 2$ として `’(,a ,b 3)` を評価するとリスト `(1 2 3)` が返る．また，マクロ関数はコンパイル時にマクロ関数部分が繰り返し展開される．例えば `pow(2,4)` は 3 度のマクロ展開を受け，

```
   pow(2,4)
-> 4 * pow(1,4)
-> 4 * 4 * pow(0,4)
-> 4 * 4 * 1
```

となる．コンマつきの式 `,(- n 1)` が各マクロ展開前に評価されているために pow 関数の第一引数の値が減少していることに注意する．このような Lisp マクロはコンパイル時にマクロ関数の展開をする点で本研究の実行時書き換えとは異なる．また，書き換え規則のように複数の関数パターンを発見し書き換えることはできない．

```
1  (defun pow (n x)
2    (if (= n 0)
3        1
4      (* x (pow (- n 1) x))))
```

図 6.1. Lisp によるべき乗関数の定義

```
1 (defmacro pow (n x)
2   (if (= n 0)
3       1
4     `(* ,x (pow ,(- n 1) ,x))))
```

図 6.2. Lisp マクロによるべき乗関数の定義

```
1 (* int * int -> int *)
2 let rec power (n, x) =
3    match n with
4        0 -> 1
5      | n -> x * (power (n-1, x))
```

図 6.3. OCaml によるべき乗関数の定義

6.2 MetaOCaml

MetaOCaml [31][32] は OCaml [33] 用のマルチステージ計算機構である．OCaml の式を引用符で囲むことで評価を保留することが可能であり，最適なコード生成などができる．図 6.3 は通常の OCaml によるべき乗関数を定義する例，図 6.4 は MetaOCaml によって特殊化されたべき乗関数を定義する例である．図 6.3 について，1 行目はコメントで関数の型を記述している．整数の組 `int * int` を取り，単一の整数 `int` を返す．`let` 文が関数定義である．`power` が関数名であり，引数 `(n,x)` を取り x^n を計算して返す．図 6.4 は 2 つの関数を定義している．`power_code` は型 `int * int code -> int code` であり，整数と整数を返すコード断片の組 `int * int code` を取り整数を返すコード断片 `int code` を返す．`.<` と `>.` で囲まれた式は評価が保留されるコード断片となる．`.~` はコード断片中で使用され，評価の保留を解除する．例えば `power_code (3, .<2>.)` は `.<2 * (2 * (2 * 1))>.` を返す．`power2` は `x` を引数にとり x^2 を返す． `.!` はコード断片を実行する．このようにして引数 `n` に特化した関数を生成できる．

Lisp マクロと異なり MetaOCaml は `.!` 演算子によりプログラム実行中の任意のタイミングでコード断片を実行可能である．よって MetaOCaml は本稿の提案手法と同様に実行時の値を実行前のコードに使用できるが，書き換え機構が有する複数の関数の使用パターンに応じた変換などはできない．

6.3 CodeBoost

CodeBoost [34] は C++ [35] 用の書き換え機構である．Rewrite rules のように規則に基づいてコンパイル時にコードを書き換える．実装には Stratego [36] という構文木変換言語が使用されている．

```
1  (* int * int -> int code *)
2  let rec power_code (n, x) =
3     match n with
4       0 -> .<1>. | n -> .<.~x * .~(power_code (n - 1, x))>.
5
6  (* int -> int *)
7  let power2 = .! .<fun x -> .~(power_code (2, .<x>.))>.
```

図 6.4. MetaOCaml によるべき乗関数の定義

```
1  void rules(){
2    sort: z + (x * y) = (x * y) + z, not(is_mult_expr(z));
3    is_mult_expr: (a * b) = true;
4  }
```

図 6.5. CodeBoost の書き換え規則の例

CodeBoost 特有の機能として，条件節の追加が存在する．図 6.5 は乗算の式について，左側に並べ替える例である．書き換え規則は `void rule()` メソッドの本体に記述し，スコープ内で参照可能な変数は全て規則に使用できる．ある書き換え規則は規則名，等式，条件節から構成され，条件節は無くても良い．ここで規則 `sort` は `not(is_mult_expr(z))` を満たす場合に有効となる．`is_mult_expr` は下に定義されている書き換え規則であり，引数 `z` が束縛した構文木が `a * b` の形であれば `true` に変換する．`not(c)` は条件節に使用できる組み込みのプリミティブであり，構文木 `c` を評価してそれが `true` なら `false` を返す．

また，書き換え戦略については組み込みで定義される数種類の戦略を選択可能である．例えば CodeBoost が用意している Stratego モジュール simplify では simplify という名前の書き換え規則を繰り返しトップダウンで適用する戦略が記述されている．よって，CodeBoost ユーザは simplify という名前の書き換え規則を定義することでこの戦略を利用できる．CodeBoost は書き換えに実行時の値を利用せず，本稿の提案手法とは異なる．

6.4 部分評価

部分評価 [37][38] とは，あるプログラムに対する入力の一部を既知のものとして，その入力に特化した新たなプログラムに変換する手法である．オフライン特殊化とオンライン特殊化の 2 つの手法が存在し，オフライン特殊化では分析と特殊化という 2 つの段階に分かれる．分析では，コンパイル時に値が与えられている変数と実行時まで値の不明な変数を区別する．そして，静的な変数の値から消去可能な部分と，残余と呼ばれるその他の部分を分ける．特殊化はコンパイル時または実行時に実行される．コンパイル時には既知の値に特殊化されたコードが生成される．実行時にはバイナリコードを使用し，新たに判明した変数の値に特殊化されたバイナリコードを生成する．図 6.6 と図 6.7 は C 言語による x^n を求めるべき乗関数について，C 言語用の部分評価器 C-Mix [39] を適用した例である．この例は文献 [39] の 2 章より引用し

```
1  int power(int n, int x){
2    int pow;
3    pow = 1;
4    while(n > 0){
5      pow = pow * x; n--;
6    }
7    return pow;
8  }
```

図 6.6. C 言語のべき乗関数の定義

```
1  int power_5(int x){
2    int pow;
3    pow = 1;
4    pow = pow * x; pow = pow * x;
5    pow = pow * x; pow = pow * x;
6    pow = pow * x;
7    return pow;
8  }
```

図 6.7. C 言語のべき乗関数を $n = 5$ として特殊化する例

ている．図 6.6 について $n = 5$ とすると図 6.7 が得られる．べき乗関数の引数 n は削除され，ループが展開されるため 5 回の乗算が残る．ループ変数 n が削除されているため特殊化された関数の実行は高速化される．

部分評価器は書き換え規則と異なり，全てプログラムが自動で実行される．そのため規則の記述といったユーザの負担がない一方，プログラムの変換は自動で導出可能な範囲に限られる．

第 7 章

まとめと今後の課題

7.1 まとめ

本研究ではオブジェクト指向言語上の内部 DSL の実行を最適化する方法として，実行時に構文木の書き換えを行うシステムを提案した．内部 DSL はホスト言語の構文を使用した，ホスト言語上での特定の問題領域に特化した言語内言語である．内部 DSL を使用することで自然な問題領域の表現が可能となる一方，ホスト言語の組み込み構文ではコンパイラによってサポートされていたような，DSL の意味論に基づいた最適化ができない．GHC にはコンパイラの機能として rewrite rules が存在し，内部 DSL 作者が DSL の意味論に基づいた書き換え規則を提供することで，コンパイル時に内部 DSL ユーザーのコードを書き換えることでプログラムの最適化が可能となっている．しかし，オブジェクト指向言語上ではクラスの継承機構が存在し，オブジェクトの実行時型はコンパイル時に判別することができない．そこで特定の書き換え対象メソッドに対して，メソッド呼び出しの際にメソッド実行前に書き換える機構を提案した．これにより書き換え対象メソッドの実引数やそれ以前の実行時の値から実行時型を判別可能となった．さらに，書き換え時に実行時の値を参照し計算することが可能であり，書き換え規則をパターンマッチのように使用することが可能となった．提案手法は Bytespresso を利用し，Java 上で実装した．行列の DSL と関数型のメソッドを持つリストクラスを使用し，未知の行列と既知の行列の積について，既知の行列が疎行列の場合に高速な行列積メソッドに置換する最適化を実行し，実行速度が向上することを示した．

7.2 今後の課題

書き換え規則に型検査を導入することが挙げられる．現在の実装では書き換え規則の両辺の構文木が示す Java コードの型の正当性は検査されていない．これは書き換え規則定義時に検査することが可能で，書き換え対象メソッド実行前に型検査結果を示すことで利便性が向上すると考えられる．

また，書き換え規則定義の内部 DSL を通常の DSL とすることが考えられる．現在は Java 上で Java の構文木を表現する内部 DSL を作成しているが，表現が冗長となっている．これ

は書き換え規則専用のパーサを作成し構文解析することでユーザビリティが向上すると考えられる．

書き換え戦略は今回簡易的にトップダウンの実装としたが，書き換え戦略についても書き換え規則と共に記述できるようにすることが考えられる．書き換え戦略を任意に指定することで書き換えの表現範囲が拡張されることが期待できる．

提案手法に項書き換え系の理論的分析をすることも考えられる．現在の実装では書き換えが停止しないパターンが存在する．これを書き換え規則の検査によって停止性を保証することなどが考えられる．

発表文献と研究活動

(1) 長田朋久，市川和央，千葉滋．停止性の保証による rewrite rules の改良． 日本ソフトウェア科学会第 33 回大会， 2015.9.6-9.

参考文献

[1] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, Vol. 35, No. 6, pp. 26–36, 2000.

[2] Chris J Date and Hugh Darwen. *A Guide To Sql Standard*, Vol. 3. Addison-Wesley Reading, 1997.

[3] Robert Mecklenburg. *Managing projects with GNU make.* " O'Reilly Media, Inc.", 2004.

[4] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pp. 483–484. Springer, 2001.

[5] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala.* Artima Inc, 2008.

[6] Scalatest. `http://www.scalatest.org`.

[7] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in scala. 2008.

[8] Slick. `http://slick.lightbend.com`.

[9] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, Vol. 93. Citeseer, 1993.

[10] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, Vol. 1, pp. 203–233, 2001.

[11] Geoffrey Mainland, Roman Leshchinskiy, S Peyton Jones, and Simon Marlow. Haskell beats c using generalized stream fusion. *Under submission*, 2013.

[12] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[14] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sug-

arj: library-based syntactic language extensibility. In *ACM SIGPLAN Notices*, Vol. 46, pp. 391–406. ACM, 2011.

[15] Ghc language features. `https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/ghc-language-features.html`.

[16] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[17] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[18] Bernhard Gramlich. *Termination and confluence properties of structured rewrite systems*. Universität Kaiserslautern. Fachbereich Informatik, 1996.

[19] Pierre Castéran and Matthieu Sozeau. A gentle introduction to type classes and relations in coq. Technical report, Citeseer, 2012.

[20] Aaron Stump. *Verified Functional Programming in Agda*. Morgan & Claypool, 2016.

[21] Sergio Antoy and Aart Middeldorp. A sequential reduction strategy. *Theoretical Computer Science*, Vol. 165, No. 1, pp. 75–95, 1996.

[22] Takashi Nagaya. *Reduction strategies for term rewriting systems*. PhD thesis, Japan Advanced Institute of Science and Technology, 1999.

[23] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[24] Gérard P Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, i. In *Computational Logic-Essays in Honor of Alan Robinson*, pp. 415–443, 1991.

[25] Shigeru Chiba, YungYu Zhuang, and Maximilian Scherr. A design of deep reification. In *Companion Proceedings of the 15th International Conference on Modularity*, pp. 168–171. ACM, 2016.

[26] Jens Palsberg and C Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*, pp. 9–15. IEEE, 1998.

[27] Steve Oualline. *Practical C programming*. " O'Reilly Media, Inc.", 1997.

[28] CUDA Nvidia. Programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2008.

[29] javaslang. `http://www.javaslang.io`.

[30] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

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

[32] Oleg Kiselyov. The design and implementation of ber metaocaml. In *International Symposium on Functional and Logic Programming*, pp. 86–102. Springer, 2014.

[33] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: functional programming for the masses*. " O'Reilly Media, Inc.", 2013.

[34] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the codeboost transformation system for domain-specific optimisation of c++ programs. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pp. 65–74. IEEE, 2003.

[35] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995.

[36] Lennart CL Kats and Eelco Visser. *The spoofax language workbench: rules for declarative specification of languages and IDEs*, Vol. 45. ACM, 2010.

[37] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, Vol. 28, No. 3, pp. 480–503, September 1996.

[38] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Eugen-Nicolae Volanschi. A Uniform Approach for Compile-time and Run-time Specialization. Research Report RR-2775, INRIA, 1996.

[39] The C-Mix Group. C-mix: Making easily maintainable c-programs run fast.

謝辞

本研究を進めるにあたり，多大なご指導ご鞭撻賜りました指導教員の千葉滋教授に心より感謝いたします．また，研究や論文の執筆にあたり様々なご助言を頂きました市川和央氏に厚くお礼を申し上げます．

最後に，日々の研究において様々なアドバイスを与えてくださった千葉研究室の皆様に感謝いたします．