Vol.2016-SE-193 No.10
2016/7/13

Javaにおける関数型インタフェースの拡張

三宅 皐[1,a)] 紙名 哲生[1,b)] 丸山 勝久[1,c)]

概要：近年，Java SE 8 が公開され，ラムダ式の記述が可能となった．これによって，抽象メソッドを 1 つのみ持つインタフェースである関数型インタフェースを簡潔に実装することができる．しかしながら，現在の Java 8 において，ラムダ式は複数の抽象メソッドを持つインタフェースには適用できず，コードの記述を簡潔にするという利点が十分に活かされていない．本論文では，複数の抽象メソッドを持つ関数型インタフェースとその実現手法を提案する．具体的には，ラムダ式と抽象メソッドとの対応を明示的に指定できるように構文を拡張したラムダ式を提案する．また，提案手法では，Java 8 のコンパイラ及び実行環境をそのまま利用可能とするために，コード変換器を作成した．このようにしてラムダ式を記述する際の障壁を下げることにより，コード記述が簡潔になる可能性が高まる．さらに，提案手法により記述したラムダ式を含むコードと匿名クラスを用いた従来のコードに対して，その理解容易性と効率性を評価した．匿名クラスのインスタンス化やその抽象メソッドの宣言に関するコード記述が不要になり，読みやすさが向上していることが確認できた．また，それぞれのコードに対して実行時間を計測したところ，大きな差は見られなかった．以上より，提案手法が十分実用的であるといえる．

Extending Functional Interfaces in Java

Ko Miyake[1,a)] Tetsuo Kamina[1,b)] Katsuhisa Maruyama[1,c)]

1. はじめに

ソフトウェア開発に用いられるプログラミング言語のひとつにオブジェクト指向言語 Java [1] がある．近年，Java のバージョンは Java SE 8（以下では Java 8）となり，ラムダ式やメソッド参照などの機能が追加され，関数型に対する操作ができるようになった [2], [3].

Java におけるラムダ式は，抽象メソッドを 1 つだけ持つインタフェース（関数型インタフェースと呼ぶ）を実装した匿名クラスにおけるメソッド（の本体）として実現されている．関数型インタフェースはメソッドを 1 つしか持たないので，匿名クラスで実装されたメソッドを実行するインスタンスとラムダ式を実行するインスタンスは同等であるとみなせる．つまり，ラムダ式は関数型インタフェースを実装した匿名クラスのインスタンスと同様に第一級オブジェクトとして扱うことができる．ラムダ式を用いた簡潔なコードは保守性や再利用性が高く，その記述はソフトウェア開発や保守において望まれている [2].

しかしながら，現在の Java 8 において，ラムダ式は抽象メソッドを複数個持つことが可能な一般的なインタフェースには適用できない．このため，コードの記述を簡潔にするという利点が十分に活かされているとはいえない．たとえば，Java 8 で提供される `MouseListener` インタフェースには 5 つの抽象メソッドが定義されている．このため，現在の Java 8 では，このインタフェースのそれぞれのメソッドの処理をラムダ式で直接記述することはできず，それを継承した匿名クラスをわざわざ定義しなければならないのが現状である．匿名クラスを用いずに抽象メソッドにおける処理をラムダ式で記述できるようにすることも可能であるが，その場合には 5 つの抽象メソッドを別々の関数型インタフェースに分離して定義する必要がある．

ここで，オブジェクト指向では，互いに関係のあるメソッドをひとまとめにして扱うのが一般的である．そのため，ラムダ式を記述するためだけにインタフェースを複数

1 立命館大学
Ritsumeikan University, Kusatsu, Shiga, 525–8577, Japan
a) mii@fse.cs.ritsumei.ac.jp
b) kamina@cs.ritsumei.ac.jp
c) maru@cs.ritsumei.ac.jp

ⓒ 2016 Information Processing Society of Japan

用意するという解決策は一般的に許容されない．複数の抽象メソッドを持つインタフェースの実装においてもラムダ式によるコード記述の簡潔さを維持するためには，それらの抽象メソッドを分離せずに関数型インタフェースとして扱えるプログラミング環境が望まれる．

本論文では，複数の抽象メソッドを持つ関数型インタフェースとその実現手法を提案する．具体的には，ラムダ式と抽象メソッドとの対応を明示的に指定できるように構文を拡張する．これにより，複数の抽象メソッドを持つ関数型インタフェースにおいてそれぞれの抽象メソッドが区別できるようになり，その実装にラムダ式を直接記述できるようになる．これにより，Java プログラムの開発や保守において，ラムダ式によるコード記述の簡潔さの恩恵を受ける機会が大幅に増加する．

また，提案手法では，Java 8 のコンパイラ及び実行環境をそのまま利用可能とするために，コード変換器を提供する．この変換器では，明示的に区別された抽象メソッドごとに従来の関数型インタフェースを生成し，それらにラムダ式を割り当てる．その際，ラムダ式の処理が並列処理に有利であるという性質を維持するため，変換によりラムダ式を消去することはせず，そのままの形で変換後のコードに残すようにしている．

ここで，このような変換により生成されたコードは，従来の匿名クラスを利用したコードに比べて複雑になる．このコードは，プログラマが直接扱うわけではないので，その理解性に関して問題となることはないといえるが，実行におけるオーバーヘッドを発生させる可能性がある．そこで，それぞれのコードに対して，実行時間を計測して比較した．その結果，それぞれのコードに対して実行時間に大きな差は現れなかった．これにより，提案手法は，実行時間の観点において，十分実用的であるといえる．

本論文の構成は次のとおりである．2 章では，背景技術としてラムダ式とその利用に関する課題を述べる．3 章で，関数型インタフェースを拡張する手法を提案する．4 章で，提案手法をコード記述の理解容易性と効率性の観点から評価した結果を示す．最後に，5 章で今後の課題を含むまとめを述べる．

2. ラムダ式の導入

本章では，Java 8 のラムダ式について簡単に説明する．

2.1 インタフェース

匿名クラスを用いてインタフェースを実装したコードの例を**図 1** に示す．このコードでは，`ActionListener` インタフェースで宣言されている `actionPerformed()` メソッドを実装することによって，アクションイベントが発生した時の処理（イベントハンドラの処理）を実装している．匿名クラスのインスタンスを `JButton` クラスのインスタンスに登録することで，実行時にアクションイベントが発生した場合に，匿名クラスのインスタンスに定義された `actionPerformed()` が呼び出される．

```
public class Button {
  public JButton createButton() {
    JButton button = new JButton();

    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        pressed();
      }
    });
    ...
  }
}
```

図 1 匿名クラスを用いたコード例

匿名クラスを用いることで，インタフェースを継承するクラスに名前を付けることなく，その実装を定義することができる．しかし，図 1 のコードにおける単純な処理（`pressed()` メソッドを呼び出す）を担うコードをイベントハンドラ（`actionPerformed()`）に記述したいだけでも，それを簡潔に記述することはできない．具体的には，イベントハンドラに関するメソッド宣言を記述したり，匿名クラスのインスタンス化に関するお決まりのコードを挿入したりしなければならない．

2.2 関数型インタフェース

Java 8 で新たに導入された関数型インタフェースとは，1 つの抽象メソッドのみを持つインタフェースである．以前に存在した `Runnable` や `ActionListener` などといったインタフェースは抽象メソッドを 1 つだけ持つので，関数型インタフェースとみなせる．また，`java.util.function` のパッケージには，関数型インタフェースを新たに定義することなくクラスの実装ができるよう，汎用的な関数型インタフェースが用意されている．

関数型インタフェースは，それを型として、ラムダ式やメソッド参照に用いることができる．関数型インタフェースには抽象メソッドが 1 つだけしか存在しないため，実装される抽象メソッドは機械的に一意に特定することができる．そのため，インタフェースを実装する際に，どの抽象メソッドを実装するのかを明示する必要はない．よって，関数型インタフェースを実装するクラス内のメソッド宣言やインスタンス化のためのコードの挿入は不要である．

2.3 ラムダ式

関数型インタフェースの導入により，それを実装するクラスのインスタンスで定義されるメソッド内部の処理を値のように扱うことができる．Java 8 では，このような匿名メソッドの処理をラムダ式で記述する．以下に，ラムダ式

```
public class Button {
  public JButton createButton() {
    JButton button = new JButton();

    button.addActionListener(
      (ActionEvent e) -> pressed());
    ...
  }
}
```

図 2 ラムダ式を用いたコード例

の文法を BNF で示す [4]．イタリック体の文字列は非終端記号を指す．また，$[x]$ は x が 0 回あるいは 1 回出現することを表す．

LambdaExpression:
  LambdaParameters `->` *LambdaBody*
LambdaParameters:
  Identifier
  ([*FormalParameterList*])
  (*InferredFormalParameterList*)
LambdaBody:
  Expression
  Block

ラムダ式（*LambdaExpression*）は，引数部（*LambdaParameters*），矢印（`->`），処理部（*LambdaBody*）で構成されている．引数部は，単一の識別子（*Identifier*），仮引数の型と名前を組にしたリスト（*FormalParameterList*）を括弧に入れたもの，あるいは，型推論により仮引数の型を省略したリスト（*InferredFormalParameterList*）を括弧に入れたものどれかである．処理部には，ラムダ式が実行する処理を記述する．これは，単独の式（*Expression*）か，複数の式で構成されるブロック（*Block*）のどちらかとなる．

`ActionListener` インタフェースの `actionPerformed()` メソッドを，ラムダ式を用いて実装したコードを**図 2** に示す．ラムダ式には，引数部として `ActionEvent` 型の引数 `e`，処理部として `pressed()` メソッドの呼び出し式が記述されている．このラムダ式を `JButton` クラスのインスタンスのイベントハンドラとして登録することにより，図 1 のコードと同様に，実行時にアクションイベントが発生した場合，`pressed()` が呼び出されることになる．

2.4 ラムダ式の利用に関する課題

図 1 に示すように，Java 8 以前では，インタフェースを実装するために匿名クラスを用いることができる．これに対して，Java 8 において関数型インタフェースとラムダ式が導入されたことにより，図 2 に示すように，より簡潔な記述でインタフェースの実装ができるようになった．

しかしながら，ラムダ式の記述が許されるのは，その型が関数型インタフェースの場合だけである．複数の抽象メソッドを持つインタフェースに対して，ラムダ式を適用することができない．たとえば，`MouseListener` インタフェースの実装を考える．このインタフェースは，マウスのボタンに関するマウスイベントを受け取るためのリスナーで，5 つの抽象メソッドを持つ．つまり，これは関数型インタフェースではない．そのため，その実装をラムダ式で記述することはできない．これは，複数の抽象メソッドを持つインタフェースに対して，どのラムダ式がどの抽象メソッドに対応するのかを推論できないからである．

オブジェクト指向において，インタフェースが抽象メソッドを 1 つしか持てないという制約は，一般的には受け入れられない．このため，実際には，抽象メソッドを複数個持つインタフェースは数多く存在する．それにもかかわらず，このようなインタフェースの存在を無視したままでは，ラムダ式の適用場面が限られてしまうというのが課題である．

3. 関数型インタフェースの拡張

2.2 節で述べたように，抽象メソッドを 1 つだけ持つ関数型インタフェースで宣言された変数（あるいは引数）のみに対して，その値としてラムダ式が利用可能である．

このような制限は，ラムダ式の処理を実際に実行する（匿名クラスの）インスタンスにおいて，ラムダ式と抽象メソッドの対応関係を一意に決定するという方針に起因する．逆にいえば，ラムダ式と抽象メソッドとの対応関係が整合性を持って機械的に決定できれば，この制限は取り除かれる．これにより，複数の抽象メソッドを持つインタフェースに対してもラムダ式の利用が可能となる．

そこで，本論文は，一般的なインタフェースに対しても，ラムダ式と抽象メソッドの対応関係を明示的に記述する記法を導入することで，ラムダ式適用可能性を広げる手法を提案する．

3.1 拡張ラムダ式

ラムダ式と抽象メソッドの対応関係を指定するために拡張したラムダ式の BNF を以下に示す．$\{x\}$ は x が 0 回以上出現することを表す．

EnhancedLambdaExpression:
  LambdaExpression
  `Lambda.collect(` [(*InterfaceType*)] *AsList* `)`
AsList:
  As LambdaExpression {`,` *As LambdaExpression* }
As:
  `(@As` *MethodName* `)`

拡張ラムダ式（*EnhancedLambdaExpression*）は，Java

8 のラムダ式（*LambdaExpression*）そのままか，ラムダ式の前に *As* 句を加えたものを 1 回以上繰り返したリスト（*AsList*）を集めたものである．提案手法では，それぞれのラムダ式を `Lambda.collect()` メソッドに対する擬似的な呼出しにおける引数とみなす．このような構文規則を採用することで，メソッド呼出し引数としてラムダ式を構文解析可能である．*As* 句の中では，ラムダ式に対応する抽象メソッドの名前（*MethodName*）を指定する．

拡張ラムダ式の型（ラムダ式が対応する抽象メソッドを持つインタフェース）は，それが記述されている代入文の右辺やメソッド呼出しの引数の型と同一とみなす．これにより，`Lambda.collect()` の戻り値に対する型の指定（*InterfaceType*）を省略することができる．一方，拡張ラムダ式をその型の下位型（サブインタフェース）の抽象メソッドに対応付けたい場合は，明示的に *InterfaceType* を記述することになる．

図 3 に，拡張ラムダ式を用いたコードの例を示す．このコードでは，拡張ラムダ式が `addMouseListener()` メソッドに対する呼出しの引数に記述されている．この引数の型（拡張ラムダ式の型）は `MouseListener` であるため，“`e -> clicked()`” および “`e -> pressed()`” に対応する抽象メソッドは，それぞれ `MouseListener` の `mouseClicked()` および `mousePressed()` となる．これらの抽象メソッドが `MouseListener` に存在しない場合，そのメソッドを指定している *As* 句が意味エラーとなる．

次に，キャスト付き拡張ラムダ式を用いたコードの例を**図 4** に示す．このコードでは，`MouseListener` を継承することで，`mouseDoubleClicked()` メソッドを持つ `MyMouseListener` インタフェースを定義している．図 3 のコードと同様に，拡張ラムダ式が `addMouseListener()` に対する呼出しの引数に記述されているため，2 つのラムダ式は `MouseListener` の抽象メソッドと推論される．しかしながら，`mouseDoubleClicked()` を持つのは `MouseListener` のサブインタフェースである `MyMouseListener` である．このままでは，ラムダ式に対応する抽象メソッドを特定することはできないので，拡張ラムダ式の型を `MyMouseListener` にキャストしている．図 3 と同様に，`MyMouseListener` に `mouseClicked()` や `mouseDoubleClicked()` が存在しない場合，そのメソッドを指定している *As* 句が意味エラーとなる．図 4 のコードにおいて，キャストの記述を忘れた場合には，`mouseDoubleClicked` を指定している *As* 句が意味エラーとなる．

3.2 コードの変換

拡張ラムダ式の導入により，複数の抽象メソッドを持つインタフェースにおいて，ラムダ式と抽象メソッドとの対応関係を明示的に指定できる．しかしながら，このように独自に書く拡張した構文規則に基づくコードは，通常の

```
public class Button {
  public JButton createButton() {
    JButton button = new JButton();

    button.addMouseListener(
      Lambda.collect(
        (@As mouseClicked) e -> clicked(),
        (@As mousePressed) e -> pressed()));
    ...
  }
}
```

図 3 拡張ラムダ式を用いたコード例

```
interface MyMouseListener extends MouseListener {
  void mouseDoubleClicked(MouseEvent e);
}

public class Button {
  public JButton createButton() {
    JButton button = new JButton();

    button.addMouseListener(
      (MyMouseListener)Lambda.collect(
        (@As mouseClicked) e -> clicked(),
        (@As mouseDoubleClicked) e -> dclicked()));
    ...
  }
}
```

図 4 キャスト付き拡張ラムダ式を用いたコード例

Java コンパイラでコンパイルすることはできない．

そこで，拡張ラムダ式が記述されているコードを従来のラムダ式が記述されているコードに変換する．変換後のコードは，通常の Java コンパイラでもコンパイル可能となる．コード変換を実現する手順は以下のようになる．

(1) ソースコード全体に対して構文解析を適用することで，拡張ラムダ式に関する情報を抽出する．
(2) それぞれのラムダ式に対応する抽象メソッドを定義する関数型インタフェースを生成する．
(3) ラムダ式の処理を関数インタフェースの抽象メソッドに転送するヘルパーメソッドを生成する．このメソッドを拡張ラムダ式を含むクラスに追加し，拡張ラムダ式をこのメソッドに対する呼び出しに置換する．

以下，手順の詳細をそれぞれ 3.2.1 節，3.2.2 節，3.2.3 節で説明する．

3.2.1 コードの解析

拡張ラムダ式では，ラムダ式の前に`@As` アノテーションが現れる．構文解析により`@As` を見つけた場合，以下の情報を収集する．

- 拡張ラムダ式 L の型 T_L（抽象メソッドを持つインタフェース）．
- それぞれのラムダ式 l（$\in L$）に対応する抽象メソッドのシグニチャ $Sig(l)$．

T_L は，拡張ラムダ式が記述されている構文要素の型（代

入文の左辺の型あるいは引数の型）から推論する（これを *InferredType* と呼ぶ）．同時に，拡張ラムダ式に対するキャストが記述されているかどうかを検査する（キャストにより指定された型を *InterfaceType* と呼ぶ）．キャストが記述されていない場合，$T_L =$ *InferredType* となる．キャストが記述されている場合は，*InterfaceType* が *InferredType* の下位型（サブインタフェース）になっているかどうかを検査する．下位型のとき，キャストは正当と判定され，$T_L =$ *InterfaceType* となる．下位型となっていない場合は，意味エラーを出力する．ここで，T_L は拡張ラムダ式の型であり，拡張ラムダ式に含まれるそれぞれの型（ラムダ式の戻り値の型）は異なることに注意する．

本手法で取得する抽象メソッドのシグニチャとはメソッド宣言全体（メソッドの名前，引数の型リスト，戻り値の型，アクセス修飾子）を指す．$Sig(l)$ は T_L に存在する抽象メソッドを検索することで取得する．いま，l に対応する抽象メソッド $M(l)$ を検索するためのキーワードを $K(l)$ とおく．$M(l)$ の名前は，それぞれのラムダ式 l の As 句に記述されたメソッドの名前から取得する（取得したメソッドの名前を $Name(l)$ と呼ぶ）．$M(l)$ の引数の型のリストは，l の仮引数リスト（*LambdaParameters*）から取得する．ここで，l の記述に仮引数の型が明示的に指定されている（*FormalParameterList* が存在する）場合は，その引数の型をリスト化するだけでよい（引数の型のリストを $ParamTypes(l)$ と呼ぶ）．この場合，$K(l)$ は $Name(l)$ と $ParamTypes(l)$ の組で表現する．つまり，$K(l) = (Name(l), ParamTypes(l))$ とする．これに対して，引数の型が省略されている（引数の名前だけのときや引数が省略されている）場合には，引数の型のリストを構築することは難しい．そこで，提案手法では，引数の型が省略されている場合，$Name(l)$ だけを用いて $K(l)$ を構築する．実際には，任意の引数の型のリスト（すべての一致するワイルドカード “*”）を導入し，$K(l) = (Name(l), *)$ とする．T_L において，$K(l)$ に一致する抽象メソッドが一意に決定できた場合，一致したメソッドのシグニチャから $Sig(l)$ を取得する．一致する抽象メソッドが存在しない場合や複数の抽象メソッドが一致する場合，意味エラーを出力する．

図 3 のコードにおける拡張ラムダ式 L の型は $T_L =$ `MouseListener`（正確には，`java.awt.event.MouseListener`）となる．2 つのラムダ式をそれぞれ l_1（“`e -> clicked()`”）と l_2（“`e -> pressed()`”）とすると，$K(l_1)$ = (`mouseClicked`, `*`), $K(l_2)$ = (`mousePressed`, `*`) となる（l_1 と l_2 の引数において型が明示的に指定されていない）．いま，`MouseListener` には `mouseClicked()` と `mousePressed()` が 1 つずつ存在するため，検索により $Sig(l_1)$ と $Sig(l_2)$ が一意に決定される．

```
interface create$1 {
  void mouseClicked(java.awt.event.MouseEvent e);
}

interface create$2 {
  void mousePressed(java.awt.event.MouseEvent e);
}
```

図 5 生成された関数型インタフェースの例

```
public class Button {
  public JButton createButton() {
    JButton button = new JButton();
    button.addMouseListener(
      create$3(
        e -> pressed(),
        e -> clicked()));
    ...
  }

  private java.awt.event.MouseListener create$3(
      create$1 arg$1,
      create$2 arg$2) {
    return new java.awt.event.MouseListener() {
      public void mouseClicked(
          java.awt.event.MouseEvent e) {
        arg$1.mouseClicked(e);
      }
      public void mousePressed(
          java.awt.event.MouseEvent e) {
        arg$2.mousePressed(e);
      }
      public void mouseReleased(
          java.awt.event.MouseEvent e) {
        // empty
      }
      public void mouseEntered(
          java.awt.event.MouseEvent e) {
        // empty
      }
      public void mouseExited(
          java.awt.event.MouseEvent e) {
        // empty
      }
    };
  }
}
```

図 6 変換後のコード例

3.2.2 関数型インタフェースの生成

Java 8 においてラムダ式を用いるためには，その型が関数型インタフェースである必要がある．ここで，コードの解析により，個々のラムダ式 l に対応する抽象メソッドのシグニチャ $Sig(l)$ はすでに特定されている．よって，$Sig(l)$ で宣言された抽象メソッドを 1 つだけ持つ関数型インタフェース $I(l)$ をそれぞれ生成すればよい．

図 5 に，図 3 のコードに基づき自動生成された関数型インタフェースを示す．実際に生成される関数型インタフェースの名前は，衝突を避けるように生成される．ここでは，コードの可読性を考慮して，インタフェース名を

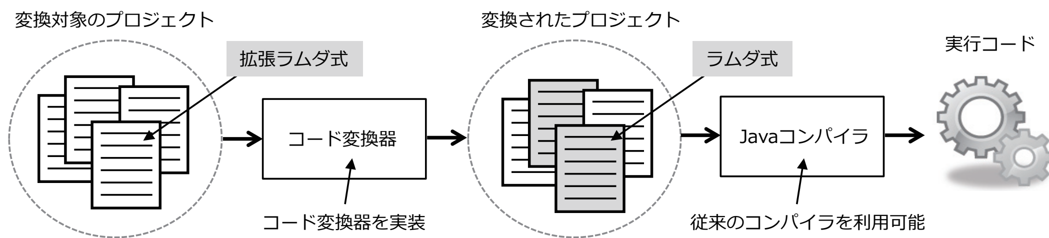

図 7 コード変換器を含むシステム全体の構成

create$1 と create$2 と表現している（実際には，識別番号が付与された名前になっている）．

3.2.3 ヘルパーメソッドの生成

ラムダ式 l のために新規に生成した関数型インタフェース $I(l)$ に対して，l の処理を $Sig(l)$ を持つ抽象メソッドに転送するヘルパーメソッドを生成する．このメソッドの引数は，新規に生成した関数型インタフェースを並べたものである．メソッド本体には，以下のコードが自動的に挿入される．

- 拡張ラムダ式の型 T_L を実現した匿名クラス C のインスタンスを生成し，それを返すコード
- T_L に存在するすべての抽象メソッドと同じ宣言を持つメソッド宣言のコード
- C に存在するメソッド本体のコード

ここでは，C に存在するメソッド本体に挿入コードについて詳しく説明する．いま，T_L に存在する抽象メソッドを m とおく．m がラムダ式 l に対応付けられている場合，C の内部に存在する $Sig(l)$ を持つメソッドの本体に，関数型インタフェース $I(l)$ への転送コードが埋め込まれる．具体的には，ヘルパーメソッドの引数として受け取った $I(l)$ の値（ラムダ式 l）に対して，$I(l)$ の m を呼び出すコードが挿入される．m がラムダ式に対応付けられていない場合は，転送コードは埋め込まれず，以下の方針でコードが挿入される．ただし，このコード挿入は，コンパイルエラーを回避するための対策であり，ラムダ式に対応しないメソッドが呼び出されることはない．

- 戻り値が void の場合，空文が挿入される（実際には，コードは挿入されない）．
- 戻り値の型のデフォルト値（その型の変数をフィールドとして宣言した際のデフォルトの戻り値）を返す return 文が挿入される．たとえば，戻り値の型が int の場合，コード “return 0;” が埋め込まれる．戻り値がプリミティブ型でない場合，コード “return null;” が埋め込まれる．

最後に，拡張ラムダ式をヘルパーメソッドに対する呼び出しに置換する．具体的には，“Lamda.collect” をヘルパーメソッドの名前に置換し，As 句を削除する．

図 6 のコードでは, 新規に生成した関数型インタフェース create$1 と create$2 を引数の型とするヘルパーメソッド create$3 が生成されている．メソッドの名前は，そのクラスに存在しないものが自動的に生成される．同時に，拡張ラムダ式がヘルパーメソッド create$3 に対する呼出しに置換され，$As$ 句が削除されている．

ヘルパーメソッドの本体を見ると，T_L = MouseListener であることより，このインタフェースを実現する匿名クラスのインスタンス化とそれを返すコードが挿入されていることが分かる．また，MouseListener には 5 つの抽象メソッドが存在するので，匿名クラスにおいて，それらと同一のシグニチャを持つメソッドが全部で 5 つ定義されている．mouseClicked() と mousePressed() については，対応するラムダ式が存在するので，それぞれの関数型インタフェースの抽象メソッドへの転送コードが埋め込まれている．残りの 3 つのメソッドは対応するラムダ式を持たないため，空文（戻り値が void であるため）が挿入されている．

3.3 実装

拡張ラムダ式を通常の Java コンパイラでコンパイル可能なコードに変換するための変換器を実装した．変換器を含むシステム全体の構成を**図 7** に示す．変換されたプロジェクトに含まれるコードは従来の Java コンパイラでコンパイル可能である．

コード変換器は Eclipse のプラグインとして実装されている．コードの変換を行う際には，まず変換対象のプロジェクトを選択する．コンテキストメニューが表示されるので，メニュー項目 “Generate Code” を選択することで，変換後のコードを含むプロジェクトが生成される．生成されたプロジェクト名は，変換対象プロジェクトの名前の最後に”Save” を付け加えたものとなる．

コード変換器の実装には，ASTParser（org.eclipse.jdt.core.dom.ASTParser クラス）を活用した．複数のラムダ式をメソッド呼出しの引数のように見せかけること，さらには，新たに導入した As 句をラムダ式に対するキャストに見せかけることで，ASTParser による構文解析を成功させている．これにより，変換前のコードからなるべく多くの解析情報（メソッドや型に関する意味情報）を取得できるようにしている．

一方，ASTParser の活用だけでは限界がある．特に，拡

張ラムダ式の意味解析に失敗することは問題である．この失敗のために，変換前のコードに対して，それぞれのラムダ式の正しさが正確に判定できない．また，意味解析の失敗は，Eclipse のエディタ上に警告として現れる．このため，拡張ラムダ式として正しいコードを記述した場合でも警告が発生する．このような警告の発生は，プログラミング環境にとって不利益であり，早急に改善すべき課題である．

4. 評価

本論文では，Java 8 において導入されたラムダ式の制限を指摘し，それを解決するためにラムダ式の拡張を提案した．さらに，拡張ラムダ式を含むコードを，通常の Java コンパイラでコンパイルできるコードに変換するための変換器を実装した．

本章では，拡張ラムダ式の導入に関して，理解容易性の向上という観点から考察を述べる．さらに，効率性の観点から実行速度を評価した結果を述べる．

4.1 拡張ラムダ式の簡潔さ

図 3 に示したコードを，従来の匿名クラスを用いて実現したコードを**図 8** に示す．このコードを見れば分かるように，インタフェースの抽象メソッドを実現したメソッドを必ず宣言する必要がある．また，匿名クラスのインスタンス化を担うコードも必ず記述する必要となる．一方，2.3 節で述べたように，ラムダ式では引数部と処理部の記述だけでその処理の実現が可能であり．メソッド宣言やインスタンス化のためのコードは不要である．さらに，ラムダ式の引数の型は関数型インタフェースから推論可能であるため，それを省略することができる．このように，ラムダ式を導入することで，煩雑なコードを取り除くことができる．

次に，ラムダ式のみを用いる場合と拡張ラムダ式を用いる場合を比較する．図 3 に示したコードを従来のラムダ式のみで実現しようとすると，図 5 および図 6 のようなコードをプログラマが記述しなければならない．これらのコード記述の大部分は定型的であるものの，その記述はプログラマにとって煩雑である．将来の保守や進化を考えると，ラムダ式を積極的に導入し，コードを簡潔に記述できることは重要である．このような観点から，ラムダ式の適用範囲を広げることは，簡潔なコード記述の普及に貢献する．

4.2 実行速度の低下

拡張ラムダ式を用いることで，プログラマが記述するコードは簡潔になる．その一方で，拡張ラムダ式を変換した後のコードは，従来の匿名クラスを利用したコードに比べて，メソッドの転送というオーバーヘッドを招いている．これにより，コンパイル後の実行コードに対する実行速度の低下が懸念される．そこで，匿名クラスを用いたコードと変換器によって生成されたコードの実行時間を計測する

```
public class Button {
  public JButton createButton() {
    JButton button = new JButton();

    button.addMouseListener(new MouseListener() {
      public void mouseClicked(MouseEvent e) {
        clicked();
      }
      public void mousePressed(MouseEvent e) {
        pressed();
      }
      public void mouseReleased(MouseEvent e) {
      }
      public void mouseEntered(MouseEvent e) {
      }
      public void mouseExited(MouseEvent e) {
      }
    });
    ...
  }
}
```

図 8　匿名クラスを用いたコード例（2）

ことにより，拡張ラムダ式の記述における実行時間の低下の影響を調べた．また，提案手法では，ラムダ式の処理が並列処理に有利であるという性質を失わないようにするため，変換によりラムダ式を消去することはせず，そのままの形で変換後のコードに残すようにしている．そこで，並列化における実行速度への影響も同時に調査した．

実行速度の計測において，特定の処理を実行する 3 つのコードを用意した．

anonymousClass　インタフェースにおける抽象メソッドの処理を匿名クラスで実現した（ラムダ式を利用しない）コード

sequential　拡張ラムダ式により処理を直接記述したコード

parallel　コード sequential を Java 8 の Stream API を用いて並列化したコード

これら 3 つのコードを複数回実行し．それぞれの実行時間を測定した．測定には，OpenJDK から提供されている JMH [5] を用いた．実験に使用したコンピュータの CPU は 4GHz Intel Core i7，メモリは 32GB 1600MHz DDR3 である．実際の計測前には，ウォームアップとしてまず 30 回の実行を行った．その後，100 回の実行時間を計測した．ウォームアップと計測のセットを 2 ラウンド実施し，それらの計測値の平均の値を算出した．

想定結果を**図 9** に示す．縦軸は実行時間 (μs) で，横軸はコードを実行した回数を指す．anonymousClass, sequential, parallel は，実験に用いた 3 つのコードを指す．

anonymousClass と sequential を比較すると，sequential の方が実行時間が少し長くなっている．今回の実験だけでは正確な原因を把握することはできないが，実行時間の差はそれほど大きくなく，実環境においては問題とならないと考えている．

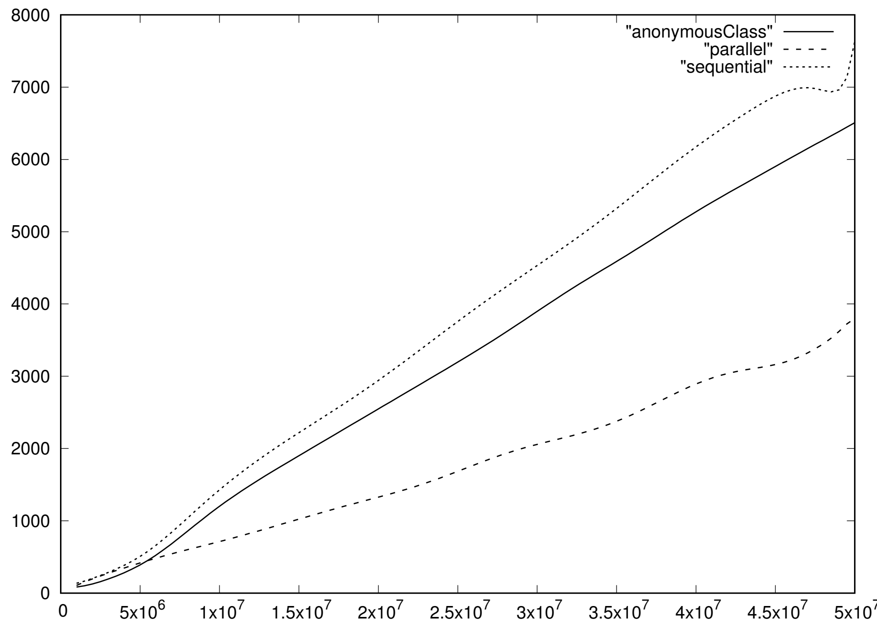

図 9 実行時間の測定結果

anonymousClass と parallel を比較すると，その実行時間はおよそ 1/2（実際には，sequential のおよそ 1/2）になっている．これは，拡張ラムダ式においても，並列化の性質が維持されていることを指す．並列化を意識した場合でも，拡張ラムダ式による記述を利用可能であることが確認できた．

5. おわりに

本論文では，関数型インタフェースの概念を拡張し，複数の抽象メソッドを持つインタフェースにおいても対応可能な拡張ラムダ式を提案した．また，拡張ラムダ式を従来と同じ開発環境でも利用できるように，Eclipse のプラグインとしてコード変換器を実装した．ラムダ式を用いた記述は従来の匿名クラスを用いる記述よりも簡潔であり，開発効率が増加や保守性の向上が期待できる．

さらに，従来のコードと拡張ラムダ式を導入したコードを用いて実行時間を測定することで，実行速度という観点からは大きなオーバヘッドはないことを示した．同時に，拡張ラムダ式での記述は並列化においても，その性質を維持することを確認した．

今後の課題として，3.1 節に示した拡張ラムダ式の構文の改善があげられる．現在の構文がプログラマにとって真に分かりやすいかどうかは不明である．提案手法の有用性を立証するためには，理解性に関する実験も早急に実施する必要がある．さらには，実行時間の評価という観点でも実験は不十分である．現在，拡張ラムダ式で記述可能なさまざまなコードを用いた実験の実施を予定している．

コード変換器の実装という観点からは，3.3 節で述べた意味解析の限界に対処する必要がある．特に，正しい拡張ラムダ式に対して警告が発生するという状況は早急に改善すべきである．また，現在のコード変換器の実装では，構文エラーや意味エラーの処理が十分でないと考えている．本論文で紹介したコード例や実験のために用意したコードについては，正しい変換が行われることを確認しているが，さまざまなエラーを含むコードに対して，エラーが正しく検出される保証はない．エラーが含まれるコードがそのまま変換されてしまう恐れも残されている．コード変換器の実装についても，さまざまなコードを用意することで，エラー検出の妥当性と変換の正しさを立証していく予定である．

謝辞 本研究の一部は，科研費 (15H02685) の助成を受けたものである．

参考文献

[1] Ken Arnold, James Gosling, David Holmes: プログラミング言語 Java，ピアソンエデュケーション (2007).
[2] Venkat Subramaniam: Java による関数型プログラミング, オライリージャパン (2014).
[3] JDK 8 の新機能 `http://www.oracle.com/technetwork/jp/java/javase/overview/8-whats-new-2157071-ja.html` (2016.06.08).
[4] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html` (2016.06.11).
[5] OpenJDK: Code Tools: jmh, `http://openjdk.java.net/projects/code-tools/jmh/` (2016.06.08).

ⓒ 2016 Information Processing Society of Japan