Crossing the Gap from Imperative to Functional Programming through Refactoring

MD 輪講

博士後期課程 2 年　楊　嘉晨

大阪大学大学院コンピュータサイエンス専攻楠本研究室

2014 年 5 月 29 日（木）

出典

Publication

タイトル: 手続き型と関数型プログラミングの隙間を越え

ESEC/FSE 2013

- 10 ページ + 参考文献
- Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering

Alex Gyori (米 Illinois 大), Lyle Franklin[1] (米 Ball 州立大),
Danny Dig (米 Oregon 州立大), and Jan Lahoda (Oracle)

[1] Lyle Franklin et al. "LambdaFicator: From Imperative to Functional Programming through Automated Refactoring". In: *Proceedings of the 2013 International Conference on Software Engineering*. Formal Demonstration. IEEE Press. 2013, pp. 1287–1290.

Java 8 におけるラムダ式

Lambda Expression in Java 8

ラムダ式（匿名関数ともいう）は名前がない関数、Java 8 から支援

```
1 blocks.stream().filter(b -> b.getColor() == BLUE)
```

これまで Java に匿名内部クラス (Anonymous Inner Class, AIC) で書かれていた単純な処理をより簡単に書ける. 上記と同じ処理を従来の書き方:

拡張 for 文

```
1 List<Block> result = new ArrayList<>();
2 for(Block b:blocks){
3   if(b.getColor() == BLUE){
4     result.add(b);
5   }
6 }
```

AIC と stream を使う

```
1 blocks.stream().filter(
2   new Predicate<Block>(){
3     @Override boolean test(Block b){
4       return b.getColor() == BLUE;
5     }
6 })
```

Java 8 におけるラムダ式

Lambda Expression in Java 8

ラムダ式（匿名関数ともいう）は名前がない関数、Java 8 から支援

```
1 blocks.stream().filter(b -> b.getColor() == BLUE)
```

これまで Java に匿名内部クラス (Anonymous Inner Class, AIC) で書かれていた単純な処理をより簡単に書ける．上記と同じ処理を従来の書き方:

拡張 for 文

```
1 List<Block> result = new ArrayList<>();
2 for(Block b:blocks){
3   if(b.getColor() == BLUE){
4     result.add(b);
5   }
6 }
```

AIC と stream を使う

```
1 blocks.stream().filter(
2   new Predicate<Block>(){
3     @Override boolean test(Block b){
4       return b.getColor() == BLUE;
5     }
6 })
```

関数型に変換による並列化

ラムダ式の導入によって書き方が簡単にするだけでなく、並列化も簡単になれる

逐次処理

```
1 for(ElementRule r:properties.getRules()){
2   r.resetHierarchy();
3 }
```

新式の並列化

```
1 properties.getRules().parallelStream().
2   forEach((ElementRule r) ->
3     r.resetHierarchy());
```

旧式の並列化

```
1  int n = 4; // amount of parallelism
2  Thread[] threads = new Thread[n];
3  final List<ElementRule> rules=properties.getRules();
4  int size = rules.size();
5  for (int i=0; i<n; i++){
6    final int from = i * size / n;
7    final int to = (i + 1) * size / n;
8    thread[i] = new Thread(new Runnable(){
9      @Override public void run(){
10       for (int j = from ; j < to ; j++){
11         rules.get(j).resetHierarchy();
12   }}}});
13   threads[i].start();
14 }
15 for(int i=0; i<n; ++i){
16   try{
17     threads[i].join();
18   } catch (InterruptedException ex) {
19     // print error message
20   }
21 }
```

この研究の貢献

Contributions

２つのリファクタリング手法を提案いたします

AnonymousToLambda AIC からラムダ式に変換

ForLoopToFunctional 拡張 for 文からラムダ式を用いて関数型に変換

NetBeans の機能として実装し、次のバージョンに提供

9 個、総行数百万行越え、性質が違うプロジェクトで評価

- 汎用性が高い。ATL は 55%,FLTF は 46% 適用可能
- 精度が高い。正解集合と比べて ATL は 100%, FLTF は 90% 以上

ツールと実験データは公開 [2] [3]

[2] http://refactoring.info/tools/LambdaFicator

[3] http://www.youtube.com/watch?v=EIyAflgHVpU

AIC からラムダ式に変換[4]

匿名内部クラス (AIC)

```
1 button.addActionListener(new ActionListener(){
2   public void actionPerformed(ActionEvent e) {
3     ui.dazzle(e.getModifiers());
4   }
5 });
```

相当するラムダ式

```
1 button.addActionListener((ActionEvent e) -> {
2     ui.dazzle(e.getModifiers());
3 });
```

[4] *State of the Lambda*. URL: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html.

AIC からラムダ式に変換[4]

匿名内部クラス (AIC)

```
1 button.addActionListener(new ActionListener(){
2   public void actionPerformed(ActionEvent e) {
3     ui.dazzle(e.getModifiers());
4   }
5 });
```

相当するラムダ式

```
1 button.addActionListener((ActionEvent e) -> {
2     ui.dazzle(e.getModifiers());
3 });
```

[4] *State of the Lambda*. URL: http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html.

AIC からラムダ式に変換する場合に特殊の例

匿名内部クラス (AIC)

```
1 String sep = doAction(new PrivilegedAction(){
2   public String run(){
3     return System.getProperty("file.separator");
4   }
5 });
```

ラムダ式に型変換が必要

```
1 String sep = doAction((PrivilegedAction)() ->
2   System.getProperty("file.separator")
3 );
```

呼びだされた doAction の定義は：

```
1 doAction(PrivilegedAction)
2 doAction(ExceptionAction)
```

変換するだけでは曖昧な二択問題が生じる

return 文だけのラムダ式では表現式だけで十分

AIC からラムダ式に変換する場合に特殊の例

匿名内部クラス (AIC)

```
1 String sep = doAction(new PrivilegedAction(){
2   public String run(){
3     return System.getProperty("file.separator");
4   }
5 });
```

ラムダ式に型変換が必要

```
1 String sep = doAction((PrivilegedAction)() ->
2   System.getProperty("file.separator")
3 );
```

呼びだされた doAction の定義は：

```
1 doAction(PrivilegedAction)
2 doAction(ExceptionAction)
```

変換するだけでは曖昧な二択問題が生じる

return 文だけのラムダ式では表現式だけで十分

AIC からラムダ式に変換する場合に特殊の例

匿名内部クラス (AIC)

```
1 String sep = doAction(new PrivilegedAction(){
2   public String run(){
3     return System.getProperty("file.separator");
4   }
5 });
```

ラムダ式に型変換が必要

```
1 String sep = doAction((PrivilegedAction)() ->
2   System.getProperty("file.separator")
3 );
```

呼びだされた doAction の定義は：

```
1 doAction(PrivilegedAction)
2 doAction(ExceptionAction)
```

変換するだけでは曖昧な二択問題が生じる

return 文だけのラムダ式では表現式だけで十分

ForLoopToFunctional の例 1

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      for(GrammarEngine e : importedEngines){
4        if(e.getGrammarName() == null) continue;
5        if(e.getGrammarName().equals(grammarName))
6          return true;
7      }
8      return false;
9    }
10 }
```

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      return importedEngines.stream()
4        .filter(e -> e.getGrammarName() != null)
5        .anyMatch(e -> e.getGrammarName().equals(grammarName));
6    }
7  }
```

ForLoopToFunctional の例 1

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      for(GrammarEngine e : importedEngines){
4        if(e.getGrammarName() == null) continue;
5        if(e.getGrammarName().equals(grammarName))
6          return true;
7      }
8      return false;
9    }
10 }
```

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      return importedEngines.stream()
4        .filter(e -> e.getGrammarName() != null)
5        .anyMatch(e -> e.getGrammarName().equals(grammarName));
6    }
7  }
```

ForLoopToFunctional の例 1

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      for(GrammarEngine e : importedEngines){
4        if(e.getGrammarName() == null) continue;
5        if(e.getGrammarName().equals(grammarName))
6          return true;
7      }
8      return false;
9    }
10 }
```

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      return importedEngines.stream()
4        .filter(e -> e.getGrammarName() != null)
5        .anyMatch(e -> e.getGrammarName().equals(grammarName));
6    }
7  }
```

ForLoopToFunctional の例 1

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      for(GrammarEngine e : importedEngines){
4        if(e.getGrammarName() == null) continue;
5        if(e.getGrammarName().equals(grammarName))
6          return true;
7      }
8      return false;
9    }
10 }
```

```
1  class GrammarEngineImpl implements GrammarEngine{
2    boolean isEngineExisting(String grammarName){
3      return importedEngines.stream()
4        .filter(e -> e.getGrammarName() != null)
5        .anyMatch(e -> e.getGrammarName().equals(grammarName));
6    }
7  }
```

ForLoopToFunctional の例 2

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      int count = 0;
4      for(ElementRule rule : getRules()){
5        if(rule.hasErrors()){
6          count += rule.getErrors().size();
7        }
8      }
9      return count;
10   }
11 }
```

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      return getRules().stream()
4        .filter(rule -> rule.hasErrors())
5        .map(rule -> rule.getErrors().size())
6        .reduce(0, Interger::plus);          // method reference in Java 8
7    }
8  }
```

ForLoopToFunctional の例 2

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      int count = 0;
4      for(ElementRule rule : getRules()){
5        if(rule.hasErrors()){
6          count += rule.getErrors().size();
7        }
8      }
9      return count;
10   }
11 }
```

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      return getRules().stream()
4        .filter(rule -> rule.hasErrors())
5        .map(rule -> rule.getErrors().size())
6        .reduce(0, Interger::plus);          // method reference in Java 8
7    }
8  }
```

ForLoopToFunctional の例 2

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      int count = 0;
4      for(ElementRule rule : getRules()){
5        if(rule.hasErrors()){
6          count += rule.getErrors().size();
7        }
8      }
9      return count;
10   }
11 }
```

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      return getRules().stream()
4        .filter(rule -> rule.hasErrors())
5        .map(rule -> rule.getErrors().size())
6        .reduce(0, Interger::plus);          // method reference in Java 8
7    }
8  }
```

ForLoopToFunctional の例 2

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      int count = 0;
4      for(ElementRule rule : getRules()){
5        if(rule.hasErrors()){
6          count += rule.getErrors().size();
7        }
8      }
9      return count;
10   }
11 }
```

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      return getRules().stream()
4        .filter(rule -> rule.hasErrors())
5        .map(rule -> rule.getErrors().size())
6        .reduce(0, Interger::plus);          // method reference in Java 8
7    }
8  }
```

ForLoopToFunctional の例 2

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      int count = 0;
4      for(ElementRule rule : getRules()){
5        if(rule.hasErrors()){
6          count += rule.getErrors().size();
7        }
8      }
9      return count;
10   }
11 }
```

```
1  class EditorGutterColumnManager{
2    int getNumberOfErrors(){
3      return getRules().stream()
4        .filter(rule -> rule.hasErrors())
5        .map(rule -> rule.getErrors().size())
6        .reduce(0, Interger::plus);          // method reference in Java 8
7    }
8  }
```

ForLoopToFunctional の例 3

```
1  List<String> findReloadedContextMemoryLeaks(){
2    List<String> result = new ArrayList<String>();
3    for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
4      if(isValid(entry)){
5        ClassLoader cl = entry.getKey();
6        if (!((WebappClassLoader)cl).isStart())
7          result.add(entry.getValue());
8      }
9    ...
10 }
```

```
1  List<String> findReloadedContextMemoryLeaks(){
2    List<String> result = new ArrayList<String>();
3    childClassLoaders.entrySet().stream()
4      .filter(entry -> isValid(entry))
5      .forEach(entry -> {
6        ClassLoader cl = entry.getKey();
7        if (!((WebappClassLoader)cl).isStart())
8          result.add(entry.getValue());});
9    ...
```

ForLoopToFunctional の例 3

```
1  List<String> findReloadedContextMemoryLeaks(){
2    List<String> result = new ArrayList<String>();
3    for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
4      if(isValid(entry)){
5        ClassLoader cl = entry.getKey();
6        if (!((WebappClassLoader)cl).isStart())
7          result.add(entry.getValue());
8      }
9    ...
10 }
```

```
1  List<String> findReloadedContextMemoryLeaks(){
2    List<String> result = new ArrayList<String>();
3    childClassLoaders.entrySet().stream()
4      .filter(entry -> isValid(entry))
5      .forEach(entry -> {
6        ClassLoader cl = entry.getKey();
7        if (!((WebappClassLoader)cl).isStart())
8          result.add(entry.getValue());});
9    ...
```

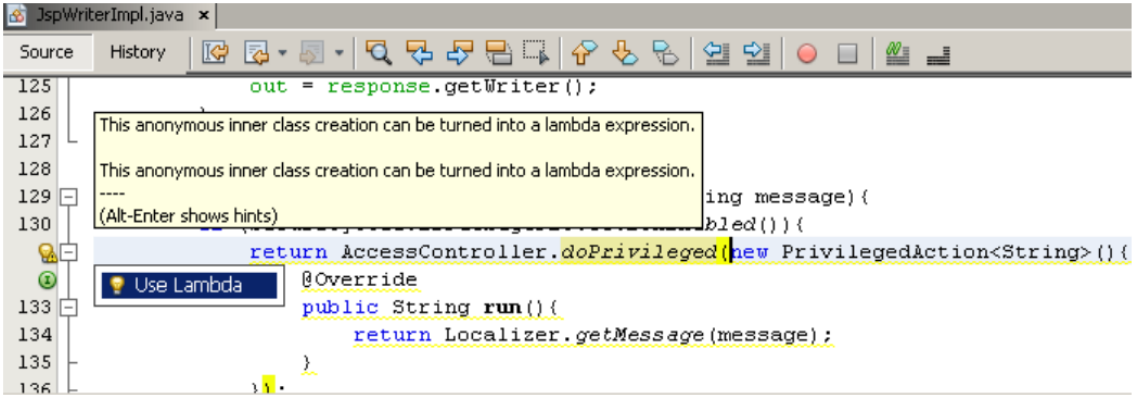
ForLoopToFunctional の例 3

```
1  List<String> findReloadedContextMemoryLeaks(){
2    List<String> result = new ArrayList<String>();
3    for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
4      if(isValid(entry)){
5        ClassLoader cl = entry.getKey();
6        if (!((WebappClassLoader)cl).isStart())
7          result.add(entry.getValue());
8      }
9    ...
10 }
```

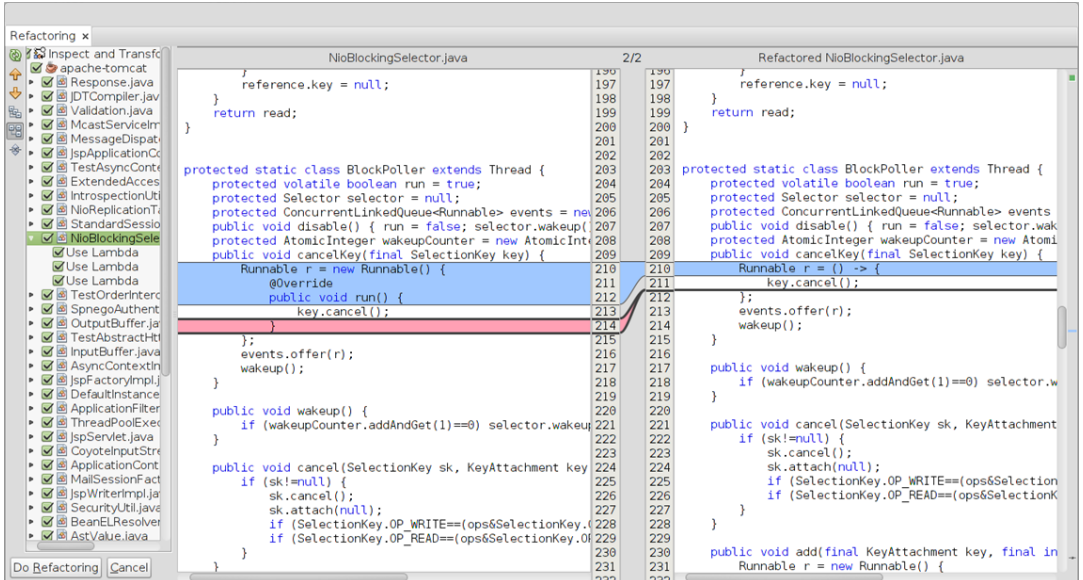
```
1  List<String> findReloadedContextMemoryLeaks(){
2    List<String> result = new ArrayList<String>();
3    childClassLoaders.entrySet().stream()
4      .filter(entry -> isValid(entry))
5      .forEach(entry -> {
6        ClassLoader cl = entry.getKey();
7        if (!((WebappClassLoader)cl).isStart())
8          result.add(entry.getValue());});
9    ...
```

1. 背景と動機の例
2. **AnonymousToLambda のリファクタリング**
   - ツールの使い方 Workflow
   - Java の型システムにおけるラムダの実装
   - 事前条件 (Preconditions)
   - 特殊の処理 (Special Cases)
3. **ForLoopToFunctional のリファクタリング**
4. 評価実験 **(Evaluation)**

ツールの使い方: Quick Hint Mode



ツールの使い方: Batch Mode



Java の型システムにおけるラムダの実装

Lambda Expression Implementation

Python, C# (3.0 以降) 等にラムダ式の型は特殊な関数型になる

Python のラムダ式

```
1 >>> type(lambda x: x==5)
2 <class 'function'>
```

C# のラムダ式

```
1 Func<int, bool> myFunc = x => x == 5;
```

Java 8 に相当する型は存在しないが、関数インターフェイスを使う

Java 8 のラムダ式

```
1 BinaryOperator sum = (x,y) -> x + y;
```

関数インターフェイス

```
1 interface BinaryOperator<T>{
2   T op(T a, T b);
3 }
```

Java の型システムにおけるラムダの実装

Lambda Expression Implementation

Python, C# (3.0 以降) 等にラムダ式の型は特殊な関数型になる

Python のラムダ式

```
1 >>> type(lambda x: x==5)
2 <class 'function'>
```

C# のラムダ式

```
1 Func<int, bool> myFunc = x => x == 5;
```

Java 8 に相当する型は存在しないが、関数インターフェイスを使う

Java 8 のラムダ式

```
1 BinaryOperator sum = (x,y) -> x + y;
```

関数インターフェイス

```
1 interface BinaryOperator<T>{
2   T op(T a, T b);
3 }
```

事前条件

Preconditions

AIC をラムダ式に変換できるのは、これらの事前条件に守れるものである

P1 AIC はインターフェイスを実装する[5]
クラスを実装する AIC は変換できない

P2 AIC にフィルドがない, メソットが一つだけ[6]
複数以上のメソットで変換できない

P3 AIC に `this` と `super` は使っていない
AIC の `this` は AIC 自身を指すが, ラムダ式に相当物がない

P4 AIC に定義したメソットは再帰的ではない (Y-combinator できる？)

[5] *Lambda Expressions*. URL: http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27.

[6] *Functional Interfaces*. URL: http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.8.

事前条件

Preconditions

AIC をラムダ式に変換できるのは、これらの事前条件に守れるものである

P1 AIC はインターフェイスを実装する[5]
クラスを実装する AIC は変換できない

P2 AIC にフィルドがない, メソットが一つだけ[6]
複数以上のメソットで変換できない

P3 AIC に `this` と `super` は使っていない
AIC の `this` は AIC 自身を指すが, ラムダ式に相当物がない

P4 AIC に定義したメソットは再帰的ではない (Y-combinator できる？)

[5] *Lambda Expressions*. URL: http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27.

[6] *Functional Interfaces*. URL: http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.8.

特殊の処理

Special Cases

AIC からラムダ式に変換する自体は簡単
要らない部分を削除したらできる

- **S1** `return` 文のみの場合に関数のブロックを省略
- **S2** AIC の型と代入されたところの型が違った場合に型変換を挿入
- **S3** AIC を使うメソットが overload された場合にも型変換を挿入
- **S4** AIC 内部に定義されたローカル変数は外部のローカル変数と名前と被った場合に，一意な名前をつける

対象となる Java 8 の新しい操作

New Operations in Java 8

目的：拡張 for 文で書かれた処理を、次の操作の連鎖 (chain) に変換

```
1  package java.util.stream;
2  interface Steam<T> ... {
3    Stream<R> map(Function<? super T, ? extends R> mapper); //lazy
4    Stream<T> filter(Predicate<? super T> predicate);       //lazy
5    T reduce(T identity, BinaryOperator<T> reducer);        //eager
6    void forEach(Consumer<? super T> consumer);             //eager
7    boolean anyMatch(Predicate<? super T> predicate);       //eager, short-circuiting
8    boolean noneMatch(Prediate<? super T> predicate);       //eager, short-circuiting
9    ...
10 }
```

ただし、拡張 for 文の意味が eager である以上、連鎖操作の最後は eager であるべき

事前条件

Preconditions

拡張 for 文で書かれた処理の内、ラムダ式に含まれないものがある

P1 配列ではなく, Collection のオブジェクトを対象とする

Collection から stream を得られるから

P2 チェック例外を外に投げない

ラムダ式に `throws` 文がないから

P3 非 final のローカル変数を一つ以上参照しない

事実上ローカル変数の前に final をつけるなら問題なし

一つだけの場合に `reduce` に変換するヒューリスティック法がある

P4-6 `break, return, continue` 文が存在しない

return boolean 文が一つの場合に `anyMatch/noneMatch` に変換するヒューリスティック法がある (後述)

ラベル無し continue 文の場合に事前にリファクタリングによって削る

変換方法

Algorithm

入力：拡張 for 文　　出力：連鎖された一連の操作
難しい点：for 文内に定義されたローカル変数の作用範囲が変わる

Step 1 拡張 for 文のブロックを文単位で潜在操作 (Prospective Operation, OP) に分割
`else` がない `if` 文 → `filter`
他の文 → `map`
最後の文 → `eager`

Step 2 分割した操作に、変数を使わられた情報を注釈

Step 3 変数依存によって、連鎖できない操作をマージ

Step 4 操作を連鎖させ

Step 2 変数を使われた情報を注釈

拡張 for 文にある各潜在操作 PO に対し

F すべてのアクセスできるフィルド変数

L_{PO} PO に定義したローカル変数

L_{Meth} 現在のメソッドに全てのローカル変数

L_{Loop} 拡張 for 文に定義した全てのローカル変数

U_{PO} PO に使われた変数

したがって、使える変数は：$AV_{PO} = F \cup L_{PO} \cup \{L_{Meth} \backslash L_{Loop}\}$

必要される変数は：$NV_{PO} = U_{PO} \backslash AV_{PO}$

Step 3 連鎖できない操作をマージ

２つの潜在操作 O と O' に関して、連鎖させるか $(O.O')$ をチェックする
即ち, 後ろの操作 O' に必要な変数を前の操作 O から得られるかのチェック

$$(O.O') \quad iff \quad |NV_{O'}| = 1 \quad and \quad NV_{O'} \subseteq (AV_O \cup NV_O)$$

連鎖できない場合に、 O と O' の２つの操作をマージして O'' になり

$$AV_{O''} = AV_O \cup AV_{O'} \quad NV_{O''} = \{NV_O \cup NV_{O'}\} \backslash AV_{O''}$$

Step 4 操作を連鎖させ

最後の操作は eager であるべきから、次のヒューリスティック法を使う

reduce 一つだけ非 final ローカル変数に対し, このいずれの計算をしている：`+=`, `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `<<=`, `>>=`, `++`, `--`

anyMatch `return true`

noneMatch `return false`

forEach 非 final ローカル変数を使ってない, かつ `return` 文がない

最後の操作を決められたら、逆順で前の操作を一つずつ繋ぎあげる。

動機の例の例 3 を用いて説明

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
3   if(isValid(entry)){
4     ClassLoader cl = entry.getKey();
5     if (!((WebappClassLoader)cl).isStart())
6       result.add(entry.getValue());
7   }
```

事前条件の確認

P1 Set は Collection

P2 例外を投げ出していない

P3 for 文外のローカル変数は result, for 文内に代入していない, 実質上 final である

P4-6 return, break, continue 文がない

動機の例の例 3 を用いて説明

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
3   if(isValid(entry)){
4     ClassLoader cl = entry.getKey();
5     if (!((WebappClassLoader)cl).isStart())
6       result.add(entry.getValue());
7   }
```

事前条件の確認

P1 Set は Collection

P2 例外を投げ出していない

P3 for 文外のローカル変数は result, for 文内に代入していない, 実質上 final である

P4-6 `return, break, continue` 文がない

動機の例の例 3 を用いて説明

Step 1, 拡張 for 文のブロックを文単位で OP に分割

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
```

```
1   if(isValid(entry)){                          // PO1: Filter
```

```
1     ClassLoader cl = entry.getKey();           // PO2: Map
```

```
1     if (!((WebappClassLoader)cl).isStart())    // PO3: Filter
```

```
1       result.add(entry.getValue());            // PO4: Eager
```

```
1   }
```

動機の例の例 3 を用いて説明

Step 2, 変数を使わられた情報を注釈

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
```

```
1   if(isValid(entry)){                          // PO1: Filter, AV={}, NV={entry}
```

```
1     ClassLoader cl = entry.getKey();           // PO2: Map, AV={cl}, NV={entry}
```

```
1     if (!((WebappClassLoader)cl).isStart())    // PO3: Filter, AV={}, NV={cl}
```

```
1       result.add(entry.getValue());            // PO4: Eager, AV={result}, NV={entry}
```

```
1   }
```

動機の例の例 3 を用いて説明

Step 3, 共通変数がある操作をマージ

$NV_{O4} = \{\text{entry}\} \nsubseteq (AV_{O3} \cup NV_{O3}) = \{\text{cl}\}$ マージすべき

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
```

```
1   if(isValid(entry)){                          // PO1: Filter, AV={}, NV={entry}
```

```
1     ClassLoader cl = entry.getKey();           // PO2: Map, AV={cl}, NV={entry}
```

```
1     if (!((WebappClassLoader)cl).isStart())    // PO3: Filter, AV={}, NV={cl}
```

```
1       result.add(entry.getValue());            // PO4: Eager, AV={result}, NV={entry}
```

```
1   }
```

動機の例の例 3 を用いて説明

Step 3, 共通変数がある操作をマージ

$|NV_{O3} = \{\text{entry, cl}\}| > 1$ マージすべき

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
```

```
1   if(isValid(entry)){                          // PO1: Filter, AV={}, NV={entry}
```

```
1     ClassLoader cl = entry.getKey();           // PO2: Map, AV={cl}, NV={entry}
```

```
1     if (!((WebappClassLoader)cl).isStart())
2       result.add(entry.getValue());  // PO3: Eager, AV={result}, NV={cl, entry}
```

```
1   }
```

動機の例の例 3 を用いて説明

Step 3, 共通変数がある操作をマージ
連鎖できる

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
```

```
1   if(isValid(entry)){                          // PO1: Filter, AV={}, NV={entry}
```

```
1     ClassLoader cl = entry.getKey();    // PO2: Eager, AV={cl,result}, NV={entry}
2     if (!((WebappClassLoader)cl).isStart())
3       result.add(entry.getValue());
```

```
1   }
```

動機の例の例 3 を用いて説明

Step 4, 操作を連鎖させ

```
1 List<String> result = new ArrayList<>();
2 for(Map.Entry<ClassLoader, String> entry: childClassLoaders.entrySet())
```

```
1   if(isValid(entry)){                          // PO1: Filter, AV={}, NV={entry}
```

```
1     ClassLoader cl = entry.getKey();  // PO2: forEach, AV={cl,result}, NV={entry}
2     if (!((WebappClassLoader)cl).isStart())
3       result.add(entry.getValue());
```

```
1   }
```

動機の例の例 3 を用いて説明

Step 4, 操作を連鎖させ

```
1 List<String> result = new ArrayList<>();
2 childClassLoaders.entrySet().stream()
```

```
1   .filter( entry -> isValid(entry))
```

```
1   .forEach( entry -> {
2     ClassLoader cl = entry.getKey();
3     if (!((WebappClassLoader)cl).isStart())
4       result.add(entry.getValue());
5       })
```

動機の例の例 3 を用いて説明

Step 4, 操作を連鎖させ

```
1 List<String> result = new ArrayList<>();
2 childClassLoaders.entrySet().stream()
3   .filter( entry -> isValid(entry))
4   .forEach( entry -> {
5     ClassLoader cl = entry.getKey();
6     if (!((WebappClassLoader)cl).isStart())
7       result.add(entry.getValue());
8   });
```

1. 背景と動機の例
2. AnonymousToLambda のリファクタリング
3. ForLoopToFunctional のリファクタリング
4. 評価実験 (Evaluation)
   - 実験設定 (Experimental Setup)
   - AnonymousToLambda の実験
   - ForLoopToFunctional の実験
   - 妥当性への脅威 (Threats to Validity)

Research Questions

Q1. 汎用 リファクタリングはどのぐらい汎用できる？
Q2. 価値 コードの品質を向上した？
Q3. 効果 プログラマーの仕事をどのぐらいに楽にしたのか？
Q4. 精度 batch mode で実行したらどのぐらいの精度がある？
Q5. 安全 安全に使える？

実験設定

Experimental Setup

Project	SLOC	Tests
ANTLR+Works v1.5.1	97795	674
Ant v1.8.4	129053	1637
Ivy v2.3.0	68193	944
Tomcat v7.0.29	221321	-
FindBugs v2.0.2	121988	163
FitNesse v20121220	63188	250
Hadoop v0.20.3	307016	-
jEdit v5	114899	0
JUnit v4.11	10443	692
Total	1133896	4360

総行数は百万行越え
jEdit に JUnit のテストケースがない
Tomcat と Hadoop は、Java 8 でテストケースが実行不可能

実験設定

Experimental Setup

Project	SLOC	Tests
ANTLR+Works v1.5.1	97795	674
Ant v1.8.4	129053	1637
Ivy v2.3.0	68193	944
Tomcat v7.0.29	221321	-
FindBugs v2.0.2	121988	163
FitNesse v20121220	63188	250
Hadoop v0.20.3	307016	-
jEdit v5	114899	0
JUnit v4.11	10443	692
Total	1133896	4360

総行数は百万行越え

jEdit に JUnit のテストケースがない

Tomcat と Hadoop は、Java 8 でテストケースが実行不可能

実験設定

Experimental Setup

Project	SLOC	Tests
ANTLR+Works v1.5.1	97795	674
Ant v1.8.4	129053	1637
Ivy v2.3.0	68193	944
Tomcat v7.0.29	221321	-
FindBugs v2.0.2	121988	163
FitNesse v20121220	63188	250
Hadoop v0.20.3	307016	-
jEdit v5	114899	0
JUnit v4.11	10443	692
Total	1133896	4360

総行数は百万行越え

jEdit に JUnit のテストケースがない

Tomcat と Hadoop は、Java 8 でテストケースが実行不可能

実験設定

Experimental Setup

Project	SLOC	Tests
ANTLR+Works v1.5.1	97795	674
Ant v1.8.4	129053	1637
Ivy v2.3.0	68193	944
Tomcat v7.0.29	221321	-
FindBugs v2.0.2	121988	163
FitNesse v20121220	63188	250
Hadoop v0.20.3	307016	-
jEdit v5	114899	0
JUnit v4.11	10443	692
Total	1133896	4360

総行数は百万行越え
jEdit に JUnit のテストケースがない
Tomcat と Hadoop は、Java 8 でテストケースが実行不可能

評価指標

Evaluation Metrics

batch mode でリファクタリングを適用

Q1. 汎用 適用された数と比率、事前条件や特殊処理の数

Q2. 価値 行数と AST のノード数に減らされた数

Q3. 効果 変更を加えたファイル数と行数、実行時間

全部対象のなかで 10% を選んで、もどのソースコードを専門家に任せて、手作業で正解集合を作る

Q4. 精度 ツールの出力と正解集合と比べて recall と precision で評価

Q5. 安全 配布された JUnit を実行し、変換前後新たな障害が起こらず
手作業で 10% を確認し、コードの意味が変わらず

AnonymousToLambda の汎用性

Project	#AIC	Conv.	Conv. [%]	P1	P2	P3	P4	S1	S2	S3	S4
ANTLR	151	118	78%	27	14	4	0	2	8	0	0
Ant	104	38	37%	50	22	0	0	14	0	1	1
Ivy	107	88	82%	11	8	2	0	10	1	0	5
Tomcat	153	87	57%	53	32	1	0	29	3	57	0
FindBugs	338	174	51%	124	90	0	0	18	0	1	0
FitNesse	102	36	35%	61	23	0	0	11	0	0	18
Hadoop	31	8	26%	23	5	0	0	7	0	2	0
jEdit	201	134	67%	59	16	2	0	2	4	0	0
JUnit	76	10	13%	63	23	0	0	7	0	0	1
Total	1263	693	55%	471	233	9	0	100	16	61	25

AnonymousToLambda の価値、効果と精度

	Value		Effort		
Project	SLOC Red.	AST Red.	Files Mod.	SLOC Mod.	Time [s]
ANTLR	243	52.6%	41	513	9
Ant	125	52.3%	21	203	15
Ivy	203	48.4%	27	387	9
Tomcat	368	52.4%	37	584	17
FindBugs	507	44.6%	53	853	17
FitNesse	99	45.8%	19	179	8
Hadoop	40	66.2%	8	60	8
jEdit	593	53.4%	61	873	13
JUnit	35	59.5%	7	55	6
Total	2213	52.8%	274	3707	102

精度に関して
recall = 100%
precision = 100%

1. 背景と動機の例
2. AnonymousToLambda のリファクタリング
3. ForLoopToFunctional のリファクタリング
4. 評価実験 (Evaluation)
   - 実験設定 (Experimental Setup)
   - AnonymousToLambda の実験
   - ForLoopToFunctional の実験
   - 妥当性への脅威 (Threads to Validity)

ForLoopToFunctional の汎用性

Project	#for loops	%Refactored	Applicability					
			P1	P2	P3	P4	P5	P6
ANTLR	589	60.10%	78	15	102	44	56	18
FitNesse	242	48.34%	79	28	22	1	25	0
Hadoop	1772	28.04%	800	600	476	130	77	27
Tomcat	425	44.70%	149	42	88	29	45	3
jEdit	190	32.10%	82	10	51	13	22	6
FindBugs	1075	40.09%	359	131	226	79	114	32
jUnit	109	54.12%	27	17	9	1	9	0
Total	4402	46.02%	1574	843	974	297	348	86

P1 が高い：配列で拡張 for 文が多い
P3 が高い：非 final ローカル変数を触ってるコード

ForLoopToFunctional の価値

Project	Value								
	`#forEach`	`#anyMatch`	`#noneMatch`	`#reduce`	`#map`	`#filter`	*#Singleton*	*#Chains*	*Avg chain length*
ANTLR	322	17	4	10	117	93	216	138	2.51
FitNesse	98	5	10	4	35	17	83	34	2.52
Hadoop	453	9	4	30	198	65	330	167	2.56
Tomcat	173	1	15	1	151	22	129	61	3.83
jEdit	61	0	0	0	23	14	37	24	2.54
FindBugs	402	14	3	12	140	82	295	136	2.63
jUnit	48	6	3	2	12	5	48	11	2.54
Total	1557	52	39	59	676	298	1138	571	2.72

ForLoopToFunctional の効果と精度

Project	Effort		
	#Files Mod.	#SLOC Mod.	Time [s]
ANTLR	140	2293	16
FitNesse	83	3336	8
Hadoop	196	2654	36
Tomcat	83	1283	16
jEdit	33	401	7
FindBugs	182	2346	16
jUnit	32	277	4
Total	717	12313	97

精度に関して

precision = 90%

専門家は操作を連鎖する癖がある
意味的には同じコード

recall = 92%

対象外の操作が使える (Stream.min 等)
`reduce` に関するヒューリスティック法が改善の余地がある

reduce のヒューリスティック法はうまく行かない例

ツールの出力

```
1 List<String> result = new ArrayList<>();
2 childClassLoaders.entrySet().stream()
3   .filter( entry -> isValid(entry))
4   .forEach( entry -> {
5     ClassLoader cl = entry.getKey();
6     if (!((WebappClassLoader)cl).isStart())
7       result.add(entry.getValue());
8   });
```

もっといい案

```
1 List<String> result = childClassLoaders.entrySet().stream()
2   .filter( entry -> isValid(entry) )
3   .filter( entry -> !((WebappClassLoader)entry.getKey()).isStart() )
4   .map( entry -> Arrays.asList(entry) )
5   .reduce(new ArrayList<String>(), List::addAll);
```

1. 背景と動機の例
2. AnonymousToLambda のリファクタリング
3. ForLoopToFunctional のリファクタリング
4. 評価実験 (Evaluation)
   - 実験設定 (Experimental Setup)
   - AnonymousToLambda の実験
   - ForLoopToFunctional の実験
   - 妥当性への脅威 (Threads to Validity)

妥当性への脅威

Threads to Validity

構造的妥当性 開発の作業量を行数や AST のノード数で評価できるか？
理想的には開発者がツールを使っている様子を観察したいが...

内部的妥当性 手作業で作られた正解集合は開発者に違いが生じるか？
正解集合を作る開発者は全部 Java ラムダ式に関する専門家

外部的妥当性 実験で得られる結果は他のプロジェクトにも汎用できるか？
百万を超える行数の性質が違うプロジェクトを 9 個実験した

信頼性 実験の方法やデータが信頼できるか？
ツールや実験環境は公開した [7]

[7] http://refactoring.info/tools/LambdaFicator/

1. 背景と動機の例
2. AnonymousToLambda のリファクタリング
3. ForLoopToFunctional のリファクタリング
4. 評価実験 (Evaluation)
5. 議論と結論
   - 議論 (Discussion)
   - 結論 (Conclusions)

議論

Discussion

ラムダ式の引数に型をつけるべきか？

- コンパイラーが推測できるから、付けないほう短くて読みやすいが...
- 付けるほうが保守性にいい[8]
- プログランマーに選択できる

拡張 for 文の他に、旧式の for 文はどう？

- 先行研究に旧式 for 文を拡張 for 文に変換する研究があった[9]

[8] Victor Pankratius, Felix Schmidt, and Gilda Garretón. "Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java". In: *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press. 2012, pp. 123–133.

[9] *Quick-assist to convert for to enhanced-for*. URL: http://archive.eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/eclipse-news-all.html#part2.

議論: 自動並列化

Discussion: Automatic Parallelism

並列化を自動化しないか？

- 先行研究で loop 文を並列化のスケーラビリティを議論した[10]
- プログランマーに正確性を確かめたうえ、簡単にできる
  `collection.stream()` → `collection.parallelStream()`
- 先行研究として data-race detector[11]もあった、組み合わせて使える

[10] Robert L Bocchino Jr et al. "A type and effect system for deterministic parallel Java". In: *ACM Sigplan Notices* 44.10 (2009), pp. 97–116.

[11] Cosmin Radoi and Danny Dig. "Practical static race detection for Java parallel loops". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 178–190.

結論

Conclusions

この研究では２つのリファクタリング手法を提案した

- AnonymousToLambda
- ForLoopToFunctional

それぞれ設計し、NetBeans の一部として実装し、実験した。
論文は発表時に、Java 8 はまだ公開していない

- 初めて Java 言語の新機能とそれを補助するツールが一緒にプログランマーに提供
- ラムダ式新機能の普及に役立てたらいいなぁ

ありがとうございました