Vol.2019-SE-201 No.4
2019/3/7

# Stream APIを利用するJavaプログラムにおけるストリーム再利用の静的検出手法

荒木 良仁[1]　桑原 寛明[2]　國枝 義敏[1]

**概要**：本稿では，Java Stream API を利用するプログラムにおけるストリームの再利用を静的に検出する手法を提案する．Java Stream API によるストリーム操作は生成操作，中間操作，終端操作に分類できる．ストリームの再利用とは同一のストリームに対し終端操作を複数回適用することである．ストリームの再利用は禁止されており，プログラムの実行中に検出されると例外がスローされるが，プログラムの実行前に検出できることが望ましい．提案手法では，メソッドの各実行経路に対し，経路上に出現する各終端操作の適用対象であるストリームがすべて異なることをデータフローに基づいて検査することで検出する．提案手法に基づいて検査器を実装し，ストリームの再利用を検出できることを確認した．

# A Static Method for Detecting Reuse of Stream in Java Programs using Stream API

## 1. はじめに

Java Stream API[1] は配列やコレクションなどのデータ列をストリームとして操作する API であり，データ列からストリームを生成する生成操作，ストリームの要素を加工する中間操作，ストリームからデータを生成して操作を完了する終端操作に分類できる．Java Stream API によるストリーム操作は，生成操作で生成したストリームに対して 0 個以上の中間操作を適用し，最後に 1 つの終端操作を適用するという流れになる．同一のストリームに対して終端操作を複数回適用することをストリームの再利用と呼ぶ．プログラムの実行中にストリームが再利用されると例外が発生するため，ストリームの再利用はプログラムの実行前に検出できることが望ましい．

そこで，本研究ではストリームの再利用を静的に検出する手法を提案する．ストリームの再利用は，プログラムのある実行経路上で同一のストリームに対して複数の終端操作を適用することなので，メソッド内の実行経路ごとにストリームの再利用が存在するかを検査する．メソッドごとに制御フローグラフを構築して実行経路を列挙する．列挙された各実行経路において，実行経路上に出現する各終端操作に対してその適用対象のストリームを生成する生成操作を特定し，複数の終端操作が同じ生成操作に対応するかを判定する．終端操作に対応する生成操作を特定するために，メソッドごとにデータフローグラフを構築する．

提案手法に基づくストリーム再利用の検査器を Java 言語を用いて実装する．制御フローグラフとデータフローグラフの生成には Java プログラムをバイトコードレベルで静的に解析するためのクラスライブラリである SOBA(Simple Objects for Bytecode Analysis)[5] を利用する．

本稿の構成は以下の通りである．2 節で Java Stream API について述べる．3 節でストリームの再利用を検出するアルゴリズムについて，4 節で検査器の実装について説明する．5 節で適用例を示す．6 節で関連研究を挙げ，7 節でまとめる．

## 2. Java Stream API

### 2.1 概要

Java Stream API（以下，単に Stream API）はデータ列を操作する API である．Stream API は主に java.util.stream パッケージの Stream<T>，IntStream，DoubleStream，LongStream インターフェースからなる．各インターフェー

1　立命館大学 情報理工学部
College of Information Science and Engineering, Ritsumeikan University
2　南山大学 情報センター
Center for Information and Communication Technology, Nanzan University

© 2019 Information Processing Society of Japan

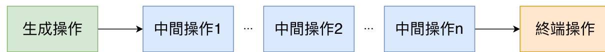


図 1　Stream API における操作の流れ

スはそれぞれ T 型，int 型，double 型，long 型のデータ列から生成されたストリームを扱う．Stream API が提供するストリーム操作は，生成操作，中間操作，終端操作の 3 種類に分類できる．生成操作はデータ列をソースとしてストリームを生成する．中間操作はストリームの要素を加工する操作であり，終端操作はストリームからデータを生成する操作である．Stream API を用いたストリーム操作の流れを**図 1** に示す．ストリーム操作は，生成操作で生成したストリームに対して 0 個以上の中間操作と 1 つの終端操作を適用するという流れで実行される．

### 2.2　ストリームの再利用

Stream API によるストリーム操作では，1 つの生成操作に対し終端操作はただ 1 つでなければならない．プログラムのある実行経路におけるストリーム操作について，1 つの生成操作に対し 2 つ以上の終端操作が存在する時に 2 回目以降の終端操作の適用をストリームの再利用と定義する．ストリームの再利用を含むプログラムはコンパイルエラーにはならないが，実行時に再利用が検出されると例外が発生する．

ストリームの再利用を含むプログラムの例を**図 2** と**図 3** に示す．`range` メソッドが生成操作，`forEach` メソッドが終端操作である．図 2 は変数 `i` と `n` の値が等しければ 4 行目の命令を実行し，等しくなければ 6 行目と 7 行目の命令を実行する．3 行目で分岐しているので，実行経路は次の 2 通り存在する．実行経路を行番号の列として表す．

(1) 1,2,3,4,8
(2) 1,2,3,6,7,8

実行経路 (1) では，1 行目の生成操作で生成されたストリームに対して 4 行目のみで終端操作が適用される．1 つの生成操作に対し終端操作が 1 つしか存在しないのでストリームは再利用されていない．実行経路 (2) では，1 行目の生成操作で生成されたストリームに対して 6 行目と 7 行目で終端操作が適用される．1 つの生成操作に対し終端操作が 2 つ存在するのでストリームが再利用されている．

図 3 は 1 行目で生成されたストリームに対して 3 行目の終端操作を for 文を用いて繰り返し適用するプログラムである．ソースコード上では 1 行目の生成操作に対応する終端操作は 3 行目のみであるが，for 文により複数回適用される可能性がある．繰り返し回数が 0 回の場合，3 行目の終端操作は実行されないのでストリームは再利用されない．繰り返し回数が 1 回の場合，3 行目の終端操作は 1 回適用される．1 つの生成操作に対し終端操作が 1 つだけ存在するためストリームの再利用ではない．繰り返し回数が 2 回以上の場合，3 行目の終端操作が 2 回以上適用される．1 つの生成操作に対し終端操作が 2 つ以上存在するのでストリームの再利用である．

```
1 IntStream a = IntStream.range(0, 10);
2 int i = 0;
3 if (i == n) {
4     a.forEach(System.out::println);
5 } else {
6     a.forEach(System.out::println);
7     a.forEach(System.out::println);
8 }
```

図 2　条件分岐による再利用

```
1 IntStream a = IntStream.range(0, 10);
2 for (int i = 0; i < n; i++) {
3     a.forEach(System.out::println);
4 }
```

図 3　繰り返し処理による再利用

## 3.　ストリーム再利用の検査アルゴリズム

### 3.1　検査の手順

ストリームの再利用の検査はメソッドごとに行う．検査の流れを次に示す．

(1) メソッドの実行経路を列挙する
(2) 実行経路上の各ストリーム操作に対し終端操作に対応する生成操作を特定する
(3) 同じ生成操作に対応する終端操作が複数存在する場合ストリームの再利用と判定する

ストリームが再利用されるかは否かは実行経路による．このためメソッドの実行経路を列挙して，実行経路ごとにストリームが再利用されるかを検査する．実行経路の列挙は SOBA が生成する制御フローグラフを探索することで実現する．終端操作に対応する生成操作の特定は，データフローグラフを終端操作からエッジを逆方向に探索することで実現する．

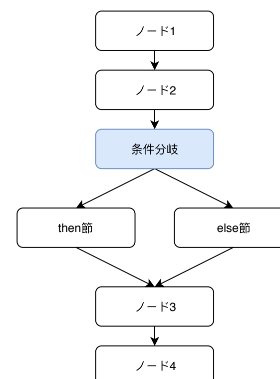


図 4　条件分岐の CFG

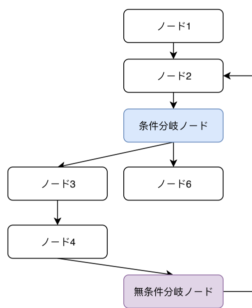


図 **5**　for 文の CFG

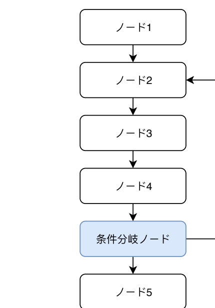


図 **6**　dowhile 文の CFG

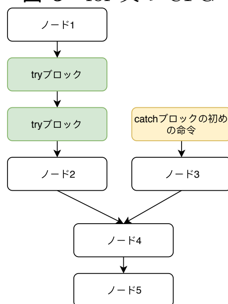


図 **7**　NormalCFG

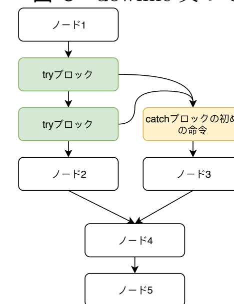


図 **8**　ConservativeCFG

### 3.2　SOBA

#### 3.2.1　制御フローグラフ

制御フローグラフ（以下 CFG とする）は条件分岐や繰り返しによって制御されるプログラム中の命令の実行順を表現するグラフである．SOBA が生成する CFG はメソッド内の各バイトコード命令をノードとして，あるノードから次に実行されるノードへと有向辺（以下エッジとする）を接続する．CFG は SOBA の DirectedGraph クラスのインスタンスとして表現され，ノードには 0 から順にノード番号が割り振られる．エッジにはノードごとに 0 から順にエッジ番号が割り振られる．

CFG では条件分岐や switch 文のノードからのみ複数のエッジを接続し，繰り返し処理の無条件分岐ノードにおいてのみ実行順をさかのぼったノードへエッジを接続する．SOBA が生成する条件分岐の CFG を図 **4** に示す．条件分岐の CFG では条件分岐を表すノードから then 節の初めの命令を表すノードと else 節の初めの命令を表すノードへエッジが接続される．

for 文の CFG は，繰り返し処理の対象となる命令列と繰り返すか否かを判定する条件分岐ノード，命令列の最初の命令へ分岐する無条件分岐ノードから構成される．while 文の CFG も for 文と同様に繰り返し実行するかを判定する条件分岐ノードと最初の命令に戻る無条件分岐ノードから構成されており，SOBA は for 文と while 文に対し同等の CFG を生成する．以下では for 文と while 文を区別せず，単に for 文と書く．SOBA が生成する for 文の CFG を図 **5** に示す．図 5 のノード 2 からノード 4 と条件分岐ノードが繰り返しの対象の命令列である．無条件分岐ノードからノード 2 へエッジを接続することで繰り返しを，条件分岐ノードからノード 6 へエッジを接続することで繰り返し処理の終了を表現している．

SOBA が生成する dowhile 文の CFG を図 **6** に示す．dowhile 文の CFG は，繰り返し処理の対象となる命令列と繰り返し実行するかを判定する条件分岐ノードから構成される．図 6 では条件分岐命令ノードからノード 2 へエッジを接続することで繰り返しを表現している．

Java 言語には例外処理を実装する構文として try-catch 文がある．SOBA は try ブロック内で例外をスローする命令を特定せず，try-catch 文を含むソースコードに対して 2 種類の CFG を用意している．1 つは図 **7** のような try ブロックの命令と catch ブロックの命令の間にエッジが存在しない CFG（以下 NormalCFG とする）であり，もう 1 つは図 **8** のような try ブロックの全ての命令から catch ブロックにおける最初の命令へのエッジが存在する CFG（以下 ConservativeCFG とする）である．

#### 3.2.2　データフローグラフ

データフローグラフ（以下 DFG とする）は宣言あるいは参照されている変数をノードとして，各変数の値がどの変数の値に影響を与えているか，変数に代入された値がどこで使用されているかを表現するグラフである．SOBA の DFG のノードは CFG のノードと同様にバイトコード命令であるが，その命令で定義あるいは参照される変数に対応するとみなし，値をスタックにプッシュする命令を表すノードからプッシュされた値を引数またはオペランドとして利用する命令を表すノードへエッジが接続される．

図 **9** のメソッドに対し SOBA が生成する DFG を図 **10** に示す．図 10 の 31 番のノードが図 9 の 10 行目の終端操作を呼び出す命令である．メソッド呼び出し命令はスタッ

© 2019 Information Processing Society of Japan

```
1  public class ReuseTest {
2    public static void main(String[] args){
3      Scanner s = new Scanner(System.in);
4      int n = s.nextInt();
5      IntStream a = IntStream.range(0, 10);
6      for (int i = 0; i < n; i++) {
7        if ((n - i) == 0) {
8          a = IntStream.range(0, 10);
9        }
10       a.forEach(System.out::println);
11     }
12   }
13 }
```

**図 9**　ストリーム操作を含むプログラム

クにプッシュされた値をレシーバーまたは引数として利用する．10 行目の終端操作は変数 a に格納されているストリームに対して呼び出される．このため変数 a からストリームを読み出してスタックにプッシュするバイトコード命令を表す 25 番のノードから 31 番のノードにエッジが接続される．図 9 の 5 行目と 8 行目で変数 a にストリームのインスタンスが格納される．10 行目の終端操作呼び出しは 7 行目の条件分岐が真であれば 8 行目，偽であれば 5 行目で格納されたストリームに対して適用される．図 10 では 10 番のノードが 5 行目の命令を表し，23 番のノードが 8 行目の命令を表す．10 番のノードで生成されたストリームは 11 番のノードで，23 番のノードで生成されたストリームは 24 番のノードで変数 a に代入される．このため 25 番のノードは 11 番と 24 番のノードからエッジで接続される．

### 3.3　実行経路の列挙

#### 3.3.1　アルゴリズム

SOBA の CFG を始点から終点まで深さ優先探索を行い，通過したノードの列を実行経路とする．図 3 のように繰り返し処理がストリーム操作を含む場合，繰り返し回数によってストリームを再利用する場合と再利用しない場合がある．繰り返し処理が終端操作のみを含む場合，繰り返し回数が 2 回以上の実行経路はすべてストリームを再利用する．このため繰り返し回数が 2 回以上の場合はまとめて扱える．そこで，for 文については繰り返し回数が 0 回，1 回，2 回の場合，dowhile 文については繰り返し回数が 1 回と 2 回の場合の実行経路を列挙する．

### 3.4　実行経路上のストリーム再利用の検査アルゴリズム

#### 3.4.1　アルゴリズム

メソッドの実行経路に対してストリームの再利用を検出するアルゴリズムを $Algorithm1$ に示す．ここで，$T_P = \{t_0, \ldots, t_n\}$ は実行経路 $P$ 上に出現する終端操作を表すノードの $P$ 上のインデックスの集合であり，$Generate(P, t_i)$ は $t_i$ が指す終端操作に対応する生成操作

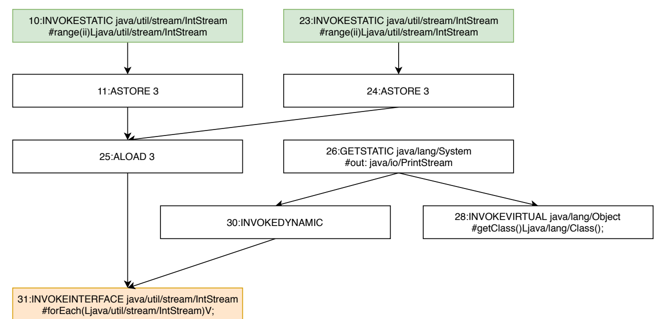


**図 10**　図 9 のメソッドから生成される DFG

を表すノードを指す実行経路 $P$ 上のインデックスである．繰り返し処理により 1 つの実行経路に同じノードが複数存在する場合があり，これらのノードを区別するため実行経路上のインデックスを利用する．実行経路上の各終端操作について対応する生成操作を特定し，複数の終端操作に同じ生成操作が対応するかを判定することでストリーム再利用を検出する．具体的には，実行経路に存在する終端操作のすべての組み合わせに対して，それぞれの終端操作に対応する生成操作が同一であるかを判定し，同一の組み合わせが存在すればストリーム再利用を検出したとする．実行経路のメソッド呼び出しノードについて，メソッド名とオーナークラス名から生成操作あるいは終端操作を表すノードであるか識別する．

**Algorithm 1** 実行経路上のストリーム再利用を検出するアルゴリズム

**Input :** 実行経路 $P$
**Output :** ストリームが再利用される場合は $true$，そうでない場合は $false$

1: **for** $i = 0$ to $|T_P| - 1$ **do**
2: 　**for** $k = i + 1$ to $|T_P|$ **do**
3: 　　**if** $Generate(P, t_i) == Generate(P, t_k)$ **then**
4: 　　　**return** $true$
5: 　　**end if**
6: 　**end for**
7: **end for**
8: **return** $false$

ストリーム操作の DFG は図 10 のようになるので，終端操作に対応する生成操作を特定するためには DFG を終端操作を表すノードからエッジを逆方向に探索すれば良い．中間操作と終端操作を表すノードはメソッド呼び出し命令であるので，操作対象のストリームのインスタンスをスタックにプッシュする命令のノードと引数のデータをスタックにプッシュする命令のノードからエッジが接続されており，ストリームのインスタンスをプッシュする命令のノードへ探索を進める．JVM はインスタンスをプッシュする命令をはじめに実行するので，ノード番号が最も小さいノードを選択すればよい．

終端操作が新しいストリームを返す場合，返されたスト

リームに対して終端操作を呼び出すことが可能である．この時，DFG 上ではストリームを返す終端操作のノードから返されたストリームに対する終端操作のノードへのエッジが存在する．生成操作のノードはメソッド名とオーナークラス名に基づいて識別されるため，ストリームを返す終端操作を生成操作であると認識できない．このことから終端操作が返すストリームに対する生成操作を特定できないので，ある終端操作から DFG を探索して生成操作の前に終端操作が見つかった場合，見つかった終端操作を生成操作とみなす．

条件分岐により図 10 のように中間操作や終端操作の適用対象が一意に確定しない場合があるので，DFG の探索はあるノードへエッジを接続するノードの中から実行経路上で直近に通過しているノードを選択する．

DFG に対して以下の記法を利用する．

- $N = \{n_0, n_1, ..., n_s\}$:DFG のノードの集合
- $PreNode(n_i) : n_i$ へとエッジを接続するノードの集合
- $S = \{s_0, s_1, ..., s_n\} \subseteq N$:DFG の始点ノードの集合
- $G = \{g_0, g_1, ..., g_n\} \subseteq N$:DFG に含まれる生成操作ノードの集合
- $T = \{t_0, t_1, ..., t_n\} \subseteq N$:DFG に含まれる終端操作ノードの集合
- $M = \{m_0, m_1, ..., m_n\} \subseteq N$:DFG に含まれるメソッド呼び出しノードの集合

$Algorithm1$ における $Generate$ のアルゴリズムを $Algorithm2$ に示す．ここで，$index$ は実行経路 $P$ 上のインデックスであり，$node$ は $index$ が指す実行経路 $P$ 上の位置にある DFG のノードである．実行経路 $P = [p_0, p_1, ..., p_n]$ に対し $Front(P, p_i) = \{p_0, p_1, \ldots, p_{i-1}\}$ とする．ノードの集合 $N_1$，$N_2$ に対し $PreIndex(N_1, N_2)$ は $N_1 \cap N_2$ に含まれるノードのインデックスの中で最大のものを表す．$Algorithm2$ は終端操作から生成操作まで DFG を逆方向に実行経路 $P$ 上に出現するノードのみを探索する．エッジが分岐している場合はエッジを接続するノードの中から，実行経路 $P$ 上で直近に通過しているノードを選択する．メソッド呼び出しノードからはインスタンスをスタックにプッシュするノードへ探索を進めるため，ノード番号が最も小さいノードを選択する．

## 4. 実装

提案するアルゴリズムを Java 言語を用いて実装する．CFG と DFG は SOBA を用いてメソッドごとに生成し，メソッド間解析は実施しない．

### 4.1 設計

実装した検査器のクラス図を**図 11** に示す．ControlFlowGraph は CFG を生成して管理するクラスであり，inspectPath メソッドで実行経路を列挙する．DataFlowGraph は

**Algorithm 2** 終端操作に対応する生成操作を求めるアルゴリズム

**Input :** $P, t_i$
**Output :** 終端操作 $p_{t_i}$ に対応する生成操作を表すノードの $P$ 上のインデックス

1: $index = t_i$
2: $node = p_{index}$
3: **while** $node \notin G \wedge node \notin S$ **do**
4:     **if** $node \in T \wedge node \neq p_{t_i}$ **then**
5:         **return** $index$
6:     **else if** $node \in M$ **then**
7:         $node = PreNode(node)$ で最もノード番号が小さい要素
8:         $index = PreIndex(Front(P, p_{index}), \{node\})$
9:     **else if** $|PreNode(node)| > 1$ **then**
10:         $index = PreIndex(Front(P, p_{index}), PreNode(node))$
11:         $node = p_{index}$
12:     **else if** $|PreNode(node)| == 1$ **then**
13:         $node = PreNode(node)$ の要素
14:         $index = PreIndex(Front(P, p_{index}), \{node\})$
15:     **end if**
16: **end while**
17: **return** $index$

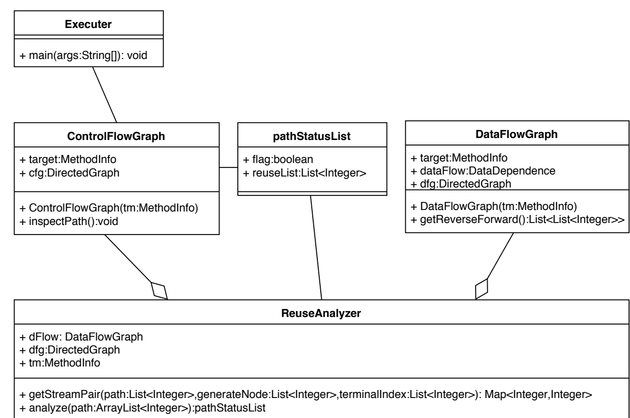


**図 11** クラス図

DFG を生成して管理するクラスである．ReuseAnalyzer はある実行経路に対してストリーム再利用を検出するクラスであり，analyze メソッドが $Algorithm1$，getStreamPair メソッドが $Algorithm2$ を実装している．ControlFlowGraph は inspectPath メソッドで CFG を表す DirectedGraph オブジェクトの実行経路を生成し，ReuseAnalyzer の analyze メソッドへ渡す．ReuseAnalyzer は analyze メソッドで受け取った実行経路を getStreamPair メソッドに渡して終端操作に対応する生成操作を特定する．getStreamPair メソッドは DFG を利用するので DataFlowGraph オブジェクトを生成して getReverseForward メソッドで DFG のエッジを反転させたグラフを取得する．

### 4.2 実装上の工夫

検査対象のメソッドの規模や内容によっては実行経路が非常に多く，すべての実行経路を格納できるだけのメモリを確保できない可能性がある．そこで，すべての実行経路

を生成してから各実行経路について検査を行うのではなく，生成された実行経路を直ちに検査して破棄することでメモリ使用量を削減する．すべての実行経路を格納する必要がなくなるためメモリを節約できる．

実行経路を通過するノードのリスト（以下ノードリストとする）ではなく，通過する条件分岐または switch 文のノード（以下分岐ノードとする）において選択したエッジのリスト（以下エッジリストとする）として表現することでメモリ使用量を削減する．実行経路上のすべてのノードではなく一部のエッジのみを保持することでメモリを節約できる．CFG は DirectedGraph で表現されるので，エッジリストには分岐ノードにおいて選択したエッジを指すエッジ番号を保存する．検査に用いる実行経路のデータ構造はノードリストであるため，エッジリストをノードリストに変換する必要がある．エッジリストからノードリストへの変換は，CFG を始点から探索し，分岐ノードを通過する時にエッジリストに従ってたどるエッジを選択することで実現できる．

SOBA は try-catch 文に対して NormalCFG と ConservativeCFG を生成できる．ConservativeCFG は try ブロックのすべての命令から catch ブロックの先頭の命令へエッジを接続するので，try-catch 文を含むメソッドの実行経路が非常に多くなる．このため，今回は NormalCFG を用いて検査器を実装した．

## 5. 評価

### 5.1 検査例

例として図 9 のプログラムを検査する．for 文の繰り返しごとのストリーム操作を区別できていることと，終端操作に対応する生成操作を特定できることを確認する．図 9 の CFG を図 **12**，DFG を図 10 に示す．実行経路は全部で 7 つあるが，その中から次の実行経路に着目する．

(1) 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,25,26,27,28,29,30,31,32,33,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,14,15,16,34

(2) 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,25,26,27,28,29,30,31,32,33,14,15,16,17,18,19,20,25,26,27,28,29,30,31,32,33,14,15,16,34

実行経路 (1) と実行経路 (2) はともに 6 行目の for 文の繰り返し回数が 2 回であるが，1 回目と 2 回目の if 文の条件の真偽が実行経路 (1) では異なり，実行経路 (2) では一致する．図 10 から，10 行目の終端操作は 31 番のノードに対応しており，終端操作の適用対象は 10 番または 23 番のノードで生成される．実行経路 (1) の場合，for 文の 1 回目では if 文の条件が偽であるので終端操作の適用対象は 10 番のノードで生成される．2 回目では if 文の条件が真であるので終端操作の適用対象は 23 番のノードで生成される．終端操作の適用対象が for 文の繰り返しの 1 回目と 2 回目で異なっており，実行経路 (1) ではストリームは再利用されない．実行経路 (2) の場合，for 文の 1 回目と 2 回目の両方で if 文の条件が偽であるので，2 回実行される終端操作の適用対象はともに 10 番のノードで生成される．同じストリームに対して終端操作が 2 回適用されており，実行経路 (2) ではストリームが再利用される．

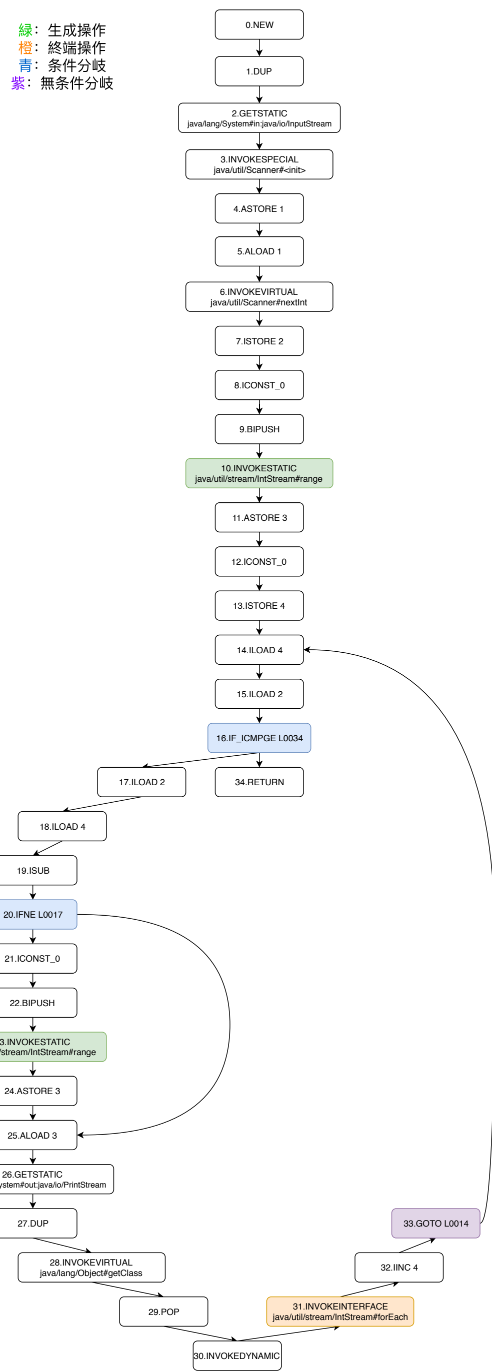


**図 12** 図 9 のメソッドから生成された CFG

実行経路 (1) に対する検査器の出力を以下に示す．ストリームの再利用が存在しないため警告は出力されていない．

```
not reuse
```

実行経路 (2) に対する検査器の出力を以下に示す．スト

リームの再利用が検出されており，検査器の出力は 6 行目の for 文，7 行目の if 文の else 節，6 行目の for 文，7 行目の if 文の else 節の順に実行した場合に再利用が発生することを表している．

```
ReusePath2:
for/while(line:6) in
if (line:7) else
for/while(line:6) in
if (line:7) else
for/while(line:6) out
```

これらの結果から，終端操作に対応する生成操作を正しく特定できていることと，繰り返し処理によって複数回実行されるストリーム操作を実行回数ごとに区別できていることがわかる．

### 5.2 実行時間

メソッドの実行経路の列挙とストリーム再利用の検出に要する時間を計測する．実行経路の数が約 10 万，約 100 万，約 1000 万のメソッドに対して，実行経路上のノード数の平均が約 300，約 600，約 900 の場合を計測する．実行環境を**表 1** に示す．

実行経路数が約 10 万のメソッドとして，最内に if 文を含む for 文の 3 重ループ，if 文を含む for 文，2 つのネストしない if 文が順に並ぶメソッドを用いる．このメソッドの実行経路数は 92597 本である．実行経路数が約 100 万のメソッドは，最内に if 文を含む for 文の 3 重ループ，最内に if 文を含む for 文と dowhile 文による 2 重ループ，if 文を含む for 文が順に並ぶメソッドであり，正確な実行経路数は 995408 本である．メソッド本体に，最内に if 文を含む for 文の 3 重ループと最内に case 式を 6 個持つ switch 文を含む for 文の 2 重ループを順に並べると実行経路数は 10936250 本となる．

実行経路数が約 10 万，100 万，1000 万で実行経路上のノード数の平均が約 300 のメソッドにおける最内に if 文を含む for 文の 3 重ループを**図 13** に，実行経路数が約 10 万で同じくノード数の平均が約 300 のメソッドにおける if 文を含む for 文を**図 14** に示す．実行経路数が約 10 万のメソッドに対して，図 13 の 5 行目と図 14 の 3 行目の命令を 8 文に増やすと実行経路上のノード数の平均が約 600，16 文に増やすと約 900 となる．実行経路 100 万と実行経路 1000 万のプログラムに対しては，図 13 の 5 行目を 9 文に増やすと実行経路上のノード数の平均が約 600，18 文に増やすと約 900 となる．

**表 1** 実行環境

| | |
|---|---|
| OS | Windows10 Pro 64bit |
| CPU | Intel(R) Core(TM) i7-7660U CPU @ 2.50GHz |
| メモリ | 8GB |

```
1 for (int i = 0; i < 10; i++)
2   for (int j = 0; j < 10; j++)
3     for (int k = 0; k < 10; k++)
4       if (i < 20)
5         IntStream
6           .range(0, 10)
7           .filter(num -> num > 2)
8           .forEach(System.out::println);
```

**図 13** 最内に if 文を含む for 文の 3 重ループ

```
1 for (int i = 0; i < 5; i++)
2   if(i == 2)
3     IntStream
4       .range(0, 10)
5       .filter(num -> num > 2)
6       .forEach(System.out::println);
```

**図 14** if 文を含む for 文

**表 2** 実行経路の列挙に要する時間 (ms)

| 実行経路 | 10 万 | 100 万 | 1000 万 |
|---|---|---|---|
| ノード数 300 | 4,832 | 73,338 | 648,378 |
| ノード数 600 | 12,824 | 129,503 | 1,678,208 |
| ノード数 900 | 26,242 | 194,890 | 2,343,168 |

**表 3** ストリームの再利用の検出に要する時間 (ms)

| 実行経路 | 10 万 | 100 万 | 1000 万 |
|---|---|---|---|
| ノード数 300 | 4,352 | 66,749 | 577,761 |
| ノード数 600 | 29,210 | 193,547 | 2,627,751 |
| ノード数 900 | 83,534 | 474,901 | 5,656,227 |

実行経路の列挙に要する時間は currentTimeMillis メソッドを用いて実行経路の列挙を開始した時刻と完了した時刻の差分を取って計測する．ストリーム再利用の検出に要する時間は実行経路ごとに currentTimeMillis メソッドを用いて検出の開始時刻と終了時刻の差分から検査時間を計測し，全実行経路の検査時間を加算して求める．それぞれについて計測された所要時間を**表 2**，**表 3** に示す．いずれも 10 回計測した値の平均である．実行経路の列挙に要する時間は対象メソッドの実行経路の本数と実行経路ごとのノード数に，ストリームの再利用の検出に要する時間は実行経路の本数に対しておおよそ線形に増加することが表 2 と表 3 から読み取れる．

### 5.3 実プロジェクトに対する適用例

実プロジェクトで開発されている Java プログラムが検査できることを確認するために，GitHub で公開されている elastic-search[*1] と incubator-dubbo[*2] の Java プログラムに

[*1] https://github.com/elastic/elasticsearch
コミット ID:3859d2166194efa5f456ada2639a25188f57d53c
[*2] https://github.com/apache/incubator-dubbo
コミット ID:7d3c6bc0aa379d05013172f11e38c0b301049b41

© 2019 Information Processing Society of Japan

表 4 メソッドの規模

| | elastic-search | incubator-dubbo |
|---|---|---|
| ノード数平均 | 57.1 | 23.4 |
| 実行経路の本数平均 | 122.5 | 1.8 |

表 5 検査に要した時間 (ms)

| | elastic-search | incubator-dubbo |
|---|---|---|
| 実行経路の列挙 | 244.8 | 27.9 |
| ストリーム再利用の検出 | 39.1 | 0.8 |

対して検査器を適用した．Stream API を用いるメソッドが elastic-search には約 850 個，incubator-dubbo には 10 個存在する．elastic-search から 20 個，incubator-dubbo から 10 個のメソッドを選択して検査した．検査結果としてストリームの再利用は検出されなかったが，選択したメソッドの中にストリームの再利用が本当に存在しないことを目視で確認した．

検査器を適用した 30 個のメソッドの実行経路の本数と経路ごとのノード数の平均を**表 4** に，検査に要した時間を**表 5** に示す．表 4 と表 5 から実プロジェクトにおいて大規模なメソッドは少なく，規模の小さなメソッドに対する検査時間は充分に短いことが観察できる．

## 6. 関連研究

長谷川らは，Java Stream API の利用時に要素数が無限個のストリームに対する停止しない API 呼び出しの出現を静的に検出する型システムを提案している [3].

田邉らは，複数のヒューリスティックを用いて C プログラムの CFG からブロック間の遷移確率を求め，各ブロックの実行回数を推測した上でデータの局所性を推測し，キャッシュメモリの利用効率が向上するようにソースコードを変換するツールを提案している [6]．プログラムの実行経路を列挙するために CFG が用いられている．Rupakheti らは，CFG を利用して equals メソッドの実行経路を列挙し，列挙した実行経路について処理のパターンを解析して equals メソッドが満たすべき規則に違反しているか検査する手法を提案している [2]．榛葉らの研究 [4] では [2] の手法に基づいてプログラムの実行経路を CFG を用いて列挙し，それぞれの実行経路に対してデータフロー解析を行うことでメソッド内に存在する変数の値や参照変数がどのクラスのオブジェクトを指すかを求めている．

## 7. おわりに

本稿では Java Stream API を用いたストリーム操作におけるストリームの再利用を静的に検出する手法を提案した．ストリームの再利用はプログラムのある実行経路上で同一のストリームに対して複数の終端操作を適用することであり，メソッドの各実行経路ごとにストリームの再利用が存在するかを検査する．実行経路上で各ストリーム操作の終端操作に対応する生成操作を特定し，同じ生成操作に対応する終端操作が複数存在する場合にストリームが再利用されていると判断する．提案した手法に基づいて検査器を実装し，ストリームの再利用を検出できることを確認した．

今後の課題として，ストリームがメソッドの引数や返り値として受け渡しされるケースに対応する必要がある．メソッド間解析によりプログラム全体のフローグラフを生成し，引数や返り値として受け取ったストリームに終端操作が適用されているかを検査する手法や，終端操作を適用したストリームを引数や返り値として渡すメソッドをあらかじめ特定しておき，該当メソッドの引数や返り値に対して終端操作が適用された場合をストリームの再利用として検出する手法などが考えられる．大規模メソッドの検査時間が長くなるので，実行時間の短縮のためにアルゴリズムと実装の改善も今後の課題である．アルゴリズムの改善としてストリーム操作を含まない分岐命令と繰り返し処理に対する実行経路の列挙を省略することが考えられる．実装の改善としてマルチスレッドや GPU を用いて検査器を並列化することが考えられる．本研究では検査器を Java プログラムとして実装したが，実用性のために Java コンパイラに組み込むことや，Java コンパイラプラグインのようなプラグインとして実装することは今後の課題である．

**謝辞** 本研究の一部は JSPS 科研費 JP15K00112，JP17K12666，JP18K11241 および 2018 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

### 参考文献

[1] Oracle: Package java.util.stream (Java Platform SE 8), `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`.

[2] Rupakheti, C. and Hou, D.: Finding Errors from ReverseEngineered Equality Models using a Constraint Solver, *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 77–86 (2012).

[3] 長谷川健太，桑原寛明，國枝義敏：Java Stream API によるストリーム操作の停止性検査のための型システム．，ソフトウェア工学の基礎 XXV (FOSE 2018)，pp. 75–84 (2018).

[4] 榛葉浩章，尾ノ上博樹，岡野浩三，楠本真二：Java における equals メソッドと hashCode メソッドの整合性の検査手法の提案，電子情報通信学会技術研究報告，Vol. 113, No. 489, pp. 19–24 (2014).

[5] 秦野智臣，石尾 隆，井上克郎：SOBA:シンプルな Java バイトコード解析ツールキット，コンピュータソフトウェア，Vol. 33, No. 4, pp. 4_4–4_15 (2016).

[6] 田邉博之，太田 剛：キャッシュメモリ効率化のための CFG 解析に基づくソースコード自動変換ツールの開発，第 75 回全国大会講演論文集，Vol. 2013, No. 1, pp. 347–348 (2013).