7J-04

Scala 向けの差分解析アルゴリズム

文　海辰　　趙　建軍

九州大学　システム情報科学研究院

1.はじめに

ソフトウェア進化において、現状のソフトウェアに対して変更を加えることが何度も行われる。その変更の目的にはバグの修正や、新たな機能の追加、パフォーマンスの改善、環境の変化に伴って挙げられる。

ソフトウェアの進化中、異なるバージョンのプログラムの変化情報はソフトウェアの理解、テスト及び保守などへ多くのソフトウェア工学の活動に有用である。このような活動に支援するのは、複数のプログラムの差分情報を計算する必要がある。具体的には、あるプログラムの二つのバージョン：オリジナルバージョンとモディファイドバージョンに基づいて、その差分情報を解析する(*Differencing Analysis*)。解析された差分情報を利用し、プログラムの追加、削除、修正と変更がないなどを識別する情報を提供することができる。これらの情報によって、*Impact Analysis* と *Regression Testing* ができる。

近年 *Scala*[1]というプログラミング言語の人気が高まっている。*Scala* はオブジェクト指向言語と関数型言語の特徴を統合したマルチパラダイムのプログラミング言語であり、特徴としては開発生産性を高める簡潔な表記が可能である。しかし、*Scala* は関数型とオブジェクト指向プログラミング言語を融合しているため、プログラムの挙動の予測が非常に難しいである。さらに、既存の差分解析アルゴリズムはオブジェクト指向プログラム[2]とアスペクト指向プログラム[3]に対応することができるが、*Scala* のようなオブジェクト指向と関数型特徴が統合しているプログラミング言語に対応することができない。このため、本研究では、*Scala* プログラムにおける差分解析手法を提案し、その開発支援ツールを開発する。

2.提案手法

2.1 差分解析アルゴリズム

差分解析アルゴリズムの入力としてはコンパイラから抽象構文木(*Abstract Syntax Tree,　AST*)を取得し、それらを用いて制御フローグラフ(*Control Flow Graph,　CFG*)を構築し、必要なデータを提供する。

差分解析アルゴリズムはオリジナルバージョンのプログラムの *CFG* データとモディファイドバージョンのプログラムの *CFG* データを入力し、*Lookahead* と *Threshold* はパラメーターにする。

Lookahead はノードがマッチされない場合には、お互いに探索する深さである。探索は前向きだけではなく、後ろ向きも探索する。例を挙げると、現在のポジションは *p* とすると、探索の範囲は(*p-Lookahead,　p+Lookahead*)である。

Threshold はメソッドのボディーの相似性の閾値である。計算方法は(マッチされるノード数　/　二つメソッドのノード数中大きい方)である。この数が *Threshold* より以上の場合、二つのメソッドがマッチされる、以下の場合にはマッチされないと認める。

メソッド内部でノードを比較する際に統一的な比較単位が必要なので、*Hammock*[2]の概念を導入する。グラフ上で *Hammock* は単一入口、単一出口の *CFG* のサブグラフである。本アルゴリズムでは文字列の配列で表示する。

マッチされるメソッドのボディーを *Hammock* 文字列 *h* と *h'*の配列に変換し、モディファイドバージョン *P'*に *Hammock* が削除、追加されるかを判断する。判断の方法は：

1. *h* と *h'*がマッチされる。変化がない。
2. *h* と *nh'*がマッチされる。*h'*はモディファイドバージョンで追加される。
3. *nh* と *h'*がマッチされる。*h* はモディファイドバージョンで削除される。

図 1： 差分解析アルゴリズムの擬似コード

Algorithm 1 Differencing algorithm for Scala

Input: *Original version data P , Modified version data P'*
Lookahead L , Threshold S
Output: *Statistics , class pairs C , object pairs O*
trait pairs T , method pairs M with label

```
1: compare classes in P and P', add matched class pairs, object pairs and
   trait pairs to C, O and T, keep a record of unmatched
2: for each pair (c, c') in C or O or T do
3:    compare methods, add matched method pairs to M, keep a record of
      unmatched method
4:    for each pair (m, m') in M do
5:       change m, m' to a series of hammock
6:       declare current hammock h and h', next hammock nh and nh, next
         next hammock nnh and nnh'
7:       while h.nonEmpty and h'.nonEmpty do
8:          if h, h' is matched then
9:             Matching Algorithm(h, h', L)
10:         end if
11:         if h, nh' is matched then
12:            Matching Algorithm(h, nh', L)
13:            keep a record of h'
14:            move to nnh, nnh'
15:         end if
16:         if nh, h' is matched then
17:            Matching Algorithm(nh, h', L)
18:            keep a record of h
19:            move to nnh, nnh'
20:         end if
21:      end while
22:      if (num of matched nodes / max num of node) > S then
23:         set label is Unchanged
24:      else
25:         set label is Changed
26:      end if
27:   end for
28: end for
29: return  C, O, T, M, Statistics
```

その後、二つの *Hammock* がマッチされると *Matching Algorithm* を呼び出して、ノードのマッチングに探索する。

図 2 は *Matching Algorithm* の擬似コードである。もし二つのノード *n* と *n'*がマッチされると次のノードに移動する。マッチされない場合には、お互いに(*p-L,　p+L*)範囲内にノードを探索する。そして、相関な情報を記録する。

最後はノードのマッチング比率が相似性閾値 *Threshold* より大きくと「*Unchanged*」、小さくと「*Changed*」とい

Differencing Algorithm for Scala Programs
Haichen Wen, Jianjun Zhao
Faculty of Information Science and Electrical Engineering, Kyushu University

図２： *Matching Algorithm* の擬似コード

Algorithm 2 Matching Algorithm
Input: *Hammock h in Original version*
Hammock h′ in Modified version
Lookahead L

```
1: declare current node n and n', current node position p
2: while n.nonEmpty and n'.nonEmpty do
3:   if n, n' is matched then
4:     keep a record of nodes
5:     continue
6:   else
7:     looking for node each other in ( p - L, p + L)
8:     keep a record of nodes
9:   end if
10: end while
```

うラベルをメソッドにつける。

2.2 Atomic Change

差分解析アルゴリズムはステートメントレベルで比較するが、プログラムが修正された後の行為を評価するために *Atomic Change*[4]を利用してプログラムの変化を分析し、評価する。

表１で示した通り、*AC*、*DC*、*AO* などを通して二つバージョンのプログラム間に *class*、*object*、*trait* と *method* の追加、削除情報を表示する。特に、*CM* はメソッドが変更を捕獲するが、その変更は次の三種類である。

1. 抽象メソッドにコードを追加すること。
2. コードを非抽象メソッドから削除し、抽象メッドになること。
3. メソッドのボディーに複数のステートメントレベルの変更があること。

表１：*Atomic Change*

略称	略称の説明
AC DC	Add an empty class Delete an empty class
AO DO	Add an empty object Delete an empty object
AT DT	Add an empty trait Delete an empty trait
AM DM CM	Add an empty method Delete an empty method Change body of method

3.実験

Apache Ant[5]を利用し、*fimpp*[6]という *Scala* プロジェクトの二つバージョンを全体的にコンパイルして、*CFG* プラグインからデータを獲得する。それらを差分解析アルゴリズムに入れる。表２はアルゴリズムの *Lookahead* と *Threshold* が各自１と 0.1 にする際の結果である。

表 2 は二つバージョンのプログラムの統計情報である。

表２： *Statistics for programs*

統計の対象	数
Class count in original version	41
Object count in original verison	53
Trait count in original version	5
Enclosing def in original version	179
Class count in modified version	41
Object count in modified version	53
Trait count in modified version	5
Enclosing def in modified version	181

表３： *Atomic Change* の結果

Atomic change	数	Atomic change	数	Atomic change	数
AC	0	AO	0	AT	0
DC	0	DO	0	DT	0
AM	2	DM	0	CM	50

オリジナルとモディファイドバージョンの *class, obejct* と *trait* の数値が同じである。しかし、それらがお互いに全部マッチされたので、表３で示した *AD*、*DC*、*AO*、*DO*、*AT* と *DT* の値がゼロになる。また、表２でオリジナルのメソッドとモディファイドバージョンのメソッドが全部マッチされ、それで表３で *AM* が２になった。

表４は *Threshold* が *CM* に対する影響の実験の結果である。示した通り、*S* が大きいほど *CM* の数値が大きくなる。その中、*S*=0.01 と *S*=1 は極端の例である。*S*=0.01 の場合はメソッドのボディーが全部変更されるメソッド数であり、*S*=1 の場合は１個の変化もある全部のメソッドの数である。

表４： *Threshold* が *CM* に対する影響

Threshold S	CM の数	Threshold S	CM の数
S = 0.01	3	S = 0.1	50
S = 0.3	72	S = 0.5	87
S = 0.7	104	S = 1	119

そして、*Lookahead* が *CM* に対する影響を明確するために、*Lookahead*=2 にして表４と同じ *CM* 数で実験をやった。実験結果は表４と同じである。その原因は：

1. 一個 *Hammock* の中にノード数が少ない。
2. ソフトウェアの更新する際ノードの順序を交換するケースが少ない。

しかし、*Lookahead* の存在意義としてはノード削除、追加することでノードのポジションがずれる可能性があるので、このような状況が結果に対する影響を減らせることである。

4.まとめ

本研究では、オブジェクト指向と関数型指向を融合する *Scala* 向けの差分解析アルゴリズムを提案した。

今後の課題としては、*Scala* プログラムの連続のバージョンに対して変化状況を分析、評価する差分解析アルゴリズムに改善する。プログラムの連続変化情報を通して、プログラムの進化方向が予測でき、メンテナンスタスクにも一層有用になる。

参考文献

[1] Martin Odersky, The Scala Language Specification Version 2.9, Technical Report, EPFL, 2011.
[2] T.Apiwattanapong, A.Orso and M. J. Harrold, “A Differencing Algorithm for Object-Oriented Programs”, Proc.ASE2004, pp.2-13, USA, 2004.
[3] Martin Th. Görg, JianJun Zhao, “Identifying Semantic Difference In AspectJ Programs”, Proc.ISSTA 2009, pp.25-36, USA, 2009.
[4] B.G. Ryder and F. Tip, “Change Impact Analysis for Object-Oriented Programs”, Proc.PASTE 2001, pp.46-53, USA, 2001.
[5] http://ant.apache.org/
[6] https://github.com/KarosS/fimpp