5M-03

関数型プログラムの条件式に関するバグの自動修正

松井健 † 佐藤亮介 ‡ 鵜林尚靖 ‡ 亀井靖高 ‡

†‡ 九州大学

1 はじめに

ソフトウェアのバグ修正の多くは手動で行われており，ソフトウェア開発において大きなコストがかかっている．そのため，自動バグ修正の研究が進められているが，現在行われている研究は手続き型のプログラムに関するものである [1]．

そこで本稿では，関数型プログラム自動バグ修正研究の足がかりとして，関数型プログラムに対する自動バグ修正手法を提案する．対象言語は，単純型付きラムダ計算に整数とブール型と再帰を含めたもので，修正対象のバグは，条件分岐の条件式における誤りである．入力としてバグを含んだプログラムを受け取り，誤った条件式を正しい条件式に変更したプログラムを出力する．

本手法では，まず，高階モデル検査に基づく関数型プログラムの検証器により修正対象のプログラムを検証し，検証器の出力する反例を利用して，プログラムのどの条件式を修正すればよいかを特定する．

次に，修正対象箇所の条件式が満たすべき制約を生成する．修正対象の条件式が評価されるときに取るべき値と，条件式を評価する際に使用可能な変数及びその変数の値を元に制約を生成する．制約を SMT ソルバで解消できれば条件式を修正し，解が見つからない場合は制約を変更して再度 SMT ソルバで解消を試みる．

最後に，修正されたプログラムを再検証する．プログラムが検証器により安全と判断された場合は修正が完了したとし，安全ではないと判断された場合は新たな反例が出力されるので，その反例に対しても修正を行う．最終的にプログラムが安全であると判断されるまで修正を繰り返す．

2 対象

本手法が対象とする言語は，単純型付きラムダ計算に整数とブール型と再帰を含めたもので，修正対象とするバグは，条件分岐の条件式における誤りである．誤った条件式を含む条件分岐は，プログラム中に 1 つのみ存在しているとする．また，非決定性を除くため，修正対象のプログラムに乱数は含まないものとする．ここでは実現可能性を調べるために，修正の対象範囲を制限している．

```
1    let f n = if n = 1 then 0 else 1
2    let main n =
3        assert (if n = 0 then f n = 0 else f n = 1)
```

図 1: 修正対象プログラムの例

プログラムの最後には，アサーションを含み引数を受け取る関数を定義し，アサーションには，プログラム中に定義された関数に対するテストを記述する．アサーションの中に含まれる条件式は必ず正しいとして修正対象にはならないものとする．

修正対象プログラムの例を図 1 に示す．この例では，main 関数内のアサーションの中に関数 f のテストが記述されている．この例では，関数 f に含まれる条件式 $n = 1$ が誤っており，正しくは $n = 0$ である．main 関数が 0 を入力値として受け取ると，関数 f の仕様は満たされずアサーションが失敗する．

3 提案手法

本章では，プログラム検証器と，修正手法について説明する．

3.1 プログラム検証器

本手法では，修正箇所の特定と，バグが含まれているかどうかを判定するためにプログラム検証器を利用する．検証器はプログラムを受け取ると，プログラム中のアサーションが失敗しないかどうかを自動で検証する．アサーションが失敗しない場合は，プログラムは安全であるとし，プログラム中にバグは含まれていないと判断する．アサーションが失敗する可能性がある場合は，プログラムは安全でないとし，検証器は反例（アサーションの失敗につながるプログラムへの入力値と簡約列）を出力する．ここで利用できる高階関数型の検証器には MoCHi[2] がある．

3.2 修正の流れ

本手法は，修正箇所の特定，修正パッチ生成，プログラムの再検証の 3 つの段階に分けられる．図 2 に修正のフローチャートを示す．

3.2.1 修正箇所の特定

本手法では，修正対象の条件式を特定するために，検証器の出力する反例を利用する．まず，修正対象の

Ken Matsui†, Ryosuke Sato‡, Naoyasu Ubayashi‡, and Yasutaka Kamei‡
†‡Kyushu University, Japan
†matsui@posl.ait.kyushu-u.ac.jp
‡{sato, ubayashi, kamei}@ait.kyushu-u.ac.jp

Copyright ©2019 Information Processing Society of Japan.
All Rights Reserved.

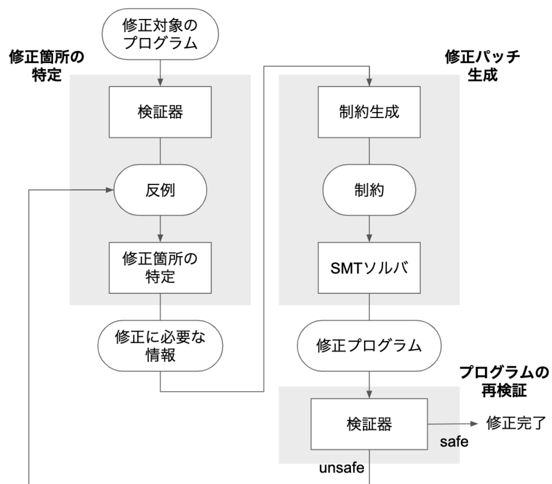

図 2: 修正フローチャート

プログラムの中から条件式を 1 つ選択して t_{cnd} とする．ここで条件式とは，$if\ t_1\ then\ t_2\ else\ t_3$ における t_1 のことである．また，修正対象のプログラムに反例の入力値を与えたものを t とする．反例の簡約列の中に，条件式 t_{cnd} の評価が現れている場合,【操作】t を $E[if\ \sigma t_{cnd}\ then\ t'_1\ else\ t'_2]$ となるまで簡約する．ここで E は評価文脈であり，σ は変数への値の代入である．

$E[if\ not\ \sigma t_{cnd}\ then\ t'_1\ else\ t'_2]$ を t' として，t' を検証器で検証した際に t' が安全であると判断された場合，条件式 t_{cnd} が修正対象となる．t' が安全でないと判断された場合は反例が出力されるので，反例の簡約列に再び t_{cnd} の評価が現れているかを調べる．もし現れていない場合は，条件式 t_{cnd} は修正対象ではないと判断し，他の条件式を対象にする．簡約列に t_{cnd} の評価が現れている場合は，t を t' として【操作】を再度行う．また，条件式 t_{cnd} の評価が反例の簡約列に複数回現れている場合は，すべてのパターンに対して上記の操作を行う．

図 1 のプログラムを検証器で検証した際の反例を図 3 に示す．図 1 の 1 行目に含まれる条件式 $n = 1$ を t_{cnd} とすると，t_{cnd} は図 3 の 6 行目において $false$ と評価されている．ここで図 3 の 6 行目までプログラムの簡約を進めた $E[if\ false\ then\ t'_1\ else\ t'_2]$ に対し，条件式の評価を反転させた $E[t'_1]$ が安全かどうかを検証する．この例では安全であると判断され，修正すべき条件式は $n = 1$ となる．

3.2.2 修正パッチ生成

修正箇所を特定した後，修正のための制約を生成する．まず，修正対象の条件式を，条件式のテンプレート P に置き換える．P は，プログラム中に存在し，かつ修正対象箇所において利用できる変数を用いて構成され，σP の制約が線形となるような式である．例えば，修正対象箇所において変数 n, m が利用可能な場合，$a_1 \times n + a_2 \times m + b = 0$ がテンプレートの 1 つとなる．

次に，修正対象のプログラム t に反例の入力値を与えたものを t' とし，t' を簡約する際の P の i 回目の代入を

```
1    Input for main:
2        0
3    Error trace:
4        main 0 ->
5        …
6        f 0 ->
7            if false then … ->
8        assert false ->
9        FAIL!
```

図 3: 入力プログラムの例に対する反例

σP_i，t' のアサーションを成功させるために σP_i が評価されるべき値を b_i とすると，P の変数係数が満たすべき制約は $\wedge_{i=1}^{n}(\sigma P_i \Leftrightarrow b_i)$ となる．この制約を SMT ソルバで解消し，見つかった解を元に条件式を修正する．解が見つからない場合は，解が見つかるまでテンプレートを変更して SMT ソルバで解消を試みる．

図 1 の例では，1 行目の条件式 $n = 1$ が修正対象箇所であり，利用できる変数として n が存在するので，$n = 1$ を $a \times n + b = 0$ にテンプレート P として置き換える．また，$\sigma P_1 \Leftrightarrow true$ が制約となる．修正対象の条件式が評価される時，n の値は 0 であるから，制約は $a \times 0 + b = 0 \Leftrightarrow true$ となり，$a = 1, b = 0$ が解として存在するので，対象の条件式を $n = 0$ に修正する．

3.2.3 プログラムの再検証

修正したプログラムを検証器で再検証し，安全だと判断された場合，修正完了となる．安全ではないと判断された場合，別の反例が出力されるので，その反例に対しても修正を行う．

4 今後の課題

本稿では，関数型プログラムの条件式に関するバグ修正手法を提案した．直近の課題としては，手法の形式化と実装があげられる．また，現在の手法では修正箇所の特定に時間がかかる可能性があるため，効率のよいバグ修正箇所の特定方法に関しても研究を進めたい．

謝辞

本研究は，18H04097 による助成を受けた．

参考文献

[1] Luca Gazzola, Daniela Micucci, Leonardo Mariani. Automatic Software Repair: A Survey. In IEEE Transactions on Software Engineering, vol.45, no.1, pp.34-67, 2019.

[2] Naoki Kobayashi, Ryosuke Sato, Hiroshi Unno. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In PLDI ’11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.222-233, 2011.

Copyright ©2019 Information Processing Society of Japan.
All Rights Reserved.