Vol.2017-SE-195 No.18
2017/3/12

分岐カバレッジ向上を目的とした静的解析を用いたテストケース自動生成

山田 誠也[1,a)]　大久保 弘崇[2,b)]　粕谷 英人[2,c)]　山本 晋一郎[2,d)]

概要：ランダムテストは人的コストをかけずにテストケース数を増やすことを可能にするため注目を集めているが，テストは入力範囲に関して均一に行われ，欠陥の検出は確率的である．本研究では Haskell を対象として，プログラムの条件分岐に出現するリテラルに注目し，ここから境界値テストのテストケースを生成する簡易なテストケース自動生成手法を提案する．本手法をランダムテストツール QuickCheck と併用することで高い分岐カバレッジを効率的に達成できることを示す．

1. はじめに

1.1 背景

ソフトウェアの不具合による社会的影響が増加の傾向しつつあるが，ソフトウェアの信頼性は，ソフトウェアテストにより担保されているため，ソフトウェアテスト自身にも高い信頼性が求められている．一方，ソフトウェアテストで消費される開発コストは大きく，ソフトウェア開発工程の 50%を占めると言われている．ソフトウェアテストの信頼性と消費コストには密接な関係があり，できる限り少ないコストで高い信頼性のソフトウェアテストを可能にする技術が必要とされている．

ソフトウェアテストで消費されるコストを削減するための技術として，テストケース自動生成手法がある．テストケース自動生成手法の最も単純な方法は，ランダムテストと呼ばれるテストケースの入力をランダムに自動生成する方法である [1]．この方法を用いれば，テストケースの入力を手動で生成するコストを削減でき，テスト担当者の考慮に入らない思わぬ欠陥を発見することが可能になる．しかし，ソフトウェアの内部構造を考慮しないため，ランダムテストでは信頼性の高いソフトウェアテストを実施することはできない．

[1] 愛知県立大学大学院情報科学研究科
Graduate School of Information Science and Technology, Aichi Prefectural University
[2] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
[a)] yamada@yamamoto.ist.aichi-pu.ac.jp
[b)] ohkubo@ist.aichi-pu.ac.jp
[c)] kasuya@ist.aichi-pu.ac.jp
[d)] yamamoto@ist.aichi-pu.ac.jp

ソフトウェアテストの信頼性を示す指標にカバレッジがある．カバレッジとは，所定の網羅条件がソフトウェアテストによってどれだけ実行されたかを割合で表した指標である [2]．このカバレッジの 1 つとして，ソフトウェアのプログラムにおける分岐を，ソフトウェアテストにより通過した割合を用いることで信頼性を表す分岐カバレッジがある．手動で生成したテストケースは，条件分岐の境界値を網羅することが可能なため，高い分岐カバレッジを達成することができる．しかし，ランダムテストで生成したテストケースは境界値を網羅できず，プログラムによっては低い分岐カバレッジを計測する可能性がある．これが，ランダムテストの信頼性を低下させる一因となっている．

1.2 目的

本稿では，ソフトウェアの内部構造を利用して境界値を網羅するテストケースを自動生成し，ランダムテストの支援として分岐カバレッジを向上させることを目的とする．このために，プログラムを静的解析した結果から，分岐構文の条件式で現れるリテラルに対応したテストケースを自動生成する手法を提案する．

1.3 貢献

本稿の貢献は，以下の 3 つである．

- 静的解析を用いてプログラム中のリテラルを抽出し，テストケースを自動生成する手法を提案した．SAT ソルバ [3] などを使用することなく，より直接的なテストケース自動生成手法でも，分岐カバレッジの向上が可能であることを実証した．
- 提案手法を併用することで，QuickCheck の自動テス

トにおける分岐カバレッジの 4%向上を達成し，同じ分岐カバレッジを達成するテストで消費する時間を 100 分の 1 に削減した．

- 提案手法により，QuickCheck を用いた自動テストで，境界値を網羅するテストケースをテスト担当者が手動で生成する必要がなくなった．このため，手動でテストケースを生成するコストを削減し，テスト担当者にかかる負担を軽減可能にした．

2. ソフトウェアテスト

1.1 節で述べたように，ソフトウェアテストは，ソフトウェアの欠陥を検出し，その信頼性を高めるための 1 つの手段であり，ソフトウェアテスト自身に対しても高い信頼性が求められる．このソフトウェアテストでは，ソフトウェアの誤りを検出するための適切な入力とその入力に対して期待される正しい出力の組を用いてソフトウェアの信頼性を確保する．この入力と出力の組をテストケースと呼ぶ．

2.1 ランダムテスト

ランダムテストとは，プログラムの誤りを検出する適切なテストケースの入力をテスト担当者が手動で生成する代わりに，数多くのテストケースを機械的に自動生成するソフトウェアテスト手法である．ランダムテストを用いることで，テスト担当者への負担を増加させることなく，テストの回数を多く実行することが可能になり，誤りを検出する可能性を増加させることができる．

2.1.1 QuickCheck

QuickCheck は，関数型言語 Haskell で開発されたランダムテストツールである．QuickCheck を用いたテストでは，テスト担当者は，プログラムの誤りを検出するために，テストの入力に対して期待される正しい出力ではなく，プログラムが満たすべき性質を関数として記述する．そして，ランダムに自動生成した入力をプログラムとその性質に与えることで，プログラムの出力結果が性質を満たすか調査し，プログラムの妥当性を確認する．満たすべき性質に反する結果が生じた場合，使用した入力を反例として出力することで，ソフトウェアの欠陥が特定可能である．

QuickCheck の使用例を，「入力されたリストを逆順にする」という仕様を持つプログラム `myReverse` を用いて説明する．その実装を**図 1** に示す．また，「入力されたリストを逆順にする」という仕様に対する満たすべき性質の例として以下を用いる．

- `prop_twice` : `myReverse` をリストに 2 回適用したら元のリストに戻る．

この性質を Haskell の関数として実装すると**図 2** になる．QuickCheck を用いたテストを行うと以下の結果が得られる．この出力は，ランダムに生成した 100 個の入力全てが，`myReverse` の満たすべき性質 `prop_twice` をパスしたことを示している．

```
1 myReverse :: [a] -> [a]
2 myReverse []     = []
3 myReverse (x:xs) = (myReverse xs) ++ [x]
```

図 1 myReverse

```
1 prop_twice :: [Int] -> Bool
2 prop_twice xs = xs == myReverse (myReverse xs)
```

図 2 prop_twice

```
1 myReverse :: [a] -> [a]
2 myReverse []     = []
3 myReverse [x]    = []
4 myReverse (x:xs) = (myReverse xs) ++ [x]
```

図 3 欠陥のある myReverse

```
*Main> quickCheck prop_twice
+++ OK, passed 100 tests.
```

次に，`myReverse` の実装を**図 3** で示すように，『長さ 1 のリスト `[x]` に対して空リスト `[]` を返す』と誤った場合について考える．図 3 のプログラムに対して，`prop_Twice` を用いたテストを行うと，以下の結果が得られる．以下の結果では，テスト 2 回目に生成した入力 `[0]` が，`myReverse` の満たすべき性質 `prop_Twice` に反する結果を出力したことを示している．

```
*Main> quickCheck prop_twice
*** Failed! Falsifiable
(after 2 tests and 2 shrinks): [0]
```

このように，QuickCheck を用いることで，テスト担当者に対して，性質を記述する負担は増加するが，欠陥を検出するための入力とその入力に対して期待される正しい出力の組であるテストケースを考える負担を取り除くことが可能になる．

2.1.2 問題点

1.1 節で述べたように，ランダムテストの問題点は，境界値を網羅するテストケースを生成できないことである．この問題点について，QuickCheck を用いて言及する．

例えば，「入力 x が `50` 以下のとき `100` を加算し，それ以外のときは入力 x を返す」という仕様を持つ関数 `add100` について考える．この関数が満たすべき性質は，**図 4** の関数 `prop_add100` となる．この際，**図 5** の関数 `add100` のように，“以下” を “未満” と誤った実装を行った場合，QuickCheck を用いたテストでは，以下の結果が得られる．

```
*Main> quickCheck prop_add100
+++ OK, passed 100 tests.
```

上記の結果では，QuickCheck は，プログラムの誤りを検

```
1 prop_add100 :: Int -> Bool
2 prop_add100 x =
3     if x <= 50 then add100 x == x + 100
4                else add100 x == x
```

図 4 add100 の満たすべき性質

```
1 add100 :: Int -> Int
2 add100 x = if x < 50 then x + 100 else x
```

図 5 欠陥プログラム

出できなかったことを示している．図 5 の add100 では，入力として整数 50 を与えたとき，実装の欠陥が明らかになる．これに対して，整数である Int 型の値が取りうる値の範囲は，-2147483648 から 2147483647 までの広大な範囲である．この Int 型の値をランダムに生成した場合，欠陥を特定できる入力である整数 50 をピンポイントで生成する可能性は限りなく低い．このピンポイントでプログラムの振る舞いを大きく変化させる境界値を QuickCheck では生成できない場合がある．

このように，テストケースの入力として，Int 型のような取りうる値が広大な範囲の代数データ型の値をランダムに生成する場合，境界値を生成できない可能性があることが QuickCheck の問題であるといえる．

2.2 境界値テスト

境界値テストとは，プログラムの分岐条件式中に記述される境界値をテストケースの入力として使用するソフトウェアテスト手法である．

境界値テストの例として，前節の「入力 x が 50 以下のとき 100 を加算し，それ以外のときはそのまま入力 x を返す」という仕様を持つ関数について考える．このプログラムの仕様から，境界値は 50 であると判断できる．このため，テストケースの入力となる値として 50 が挙げられる．また，この入力 50 に対して期待される出力は 100 となる．このテストケースをプログラムで実装すると**図 6** のようになる．この図 6 のテストケースを実行すると，図 5 の add100 の欠陥をテストにより発見することができる．この一連の流れが境界値テストの流れになる．

境界値テストでは，境界値に加えて境界値の前後の値をテストケースの入力として使用する．例えば，Int 型の値である 10 の前後の値は 9 と 11 になる．また．Char 型の値である'b' の前後の値は'a' と'c' である．このように，前後の値が一意に定まる代数データ型のことを列挙型と呼ぶ．

これに対して，リストの前後の値は一意に定めることができない．リストとは，特定の代数データ型の要素を格納する n 次元ベクトルの代数データ型であるため，前後の値として様々な形が考えられる．以下にその例を 4 つ挙げる．

```
1 testCase :: Bool
2 testCase = add100 50 == 100
```

図 6 境界値テストケース

```
1 altExam :: Int -> Int -> Int
2 altExam 0 y = y               -- alt1
3 altExam x 0 =
4     if x >= 100 then x        -- alt2, alt3
5                 else x + 100  -- alt4
6 altExam x y = x + y           -- alt5
```

図 7 分岐を持つプログラム

```
1 testCase1 = altExam   0 1 ==   1 -- alt1
2 testCase2 = altExam 100 0 == 100 -- alt2,alt3
3 testCase3 = altExam  99 0 == 199 -- alt2,alt4
4 testCase4 = altExam   1 1 ==   2 -- alt5
```

図 8 分岐カバレッジを 100%にするテストケース

表 1 altExam の分岐条件

	第 1 引数	第 2 引数
alt1	0	any
alt2	any	0
alt3	100 以上	0
alt4	100 未満	0
alt5	any	any

- 先頭要素を前後の値に 1 つ転置したリスト
- 全ての要素を前後の値に 1 つ転置したリスト
- 長さを前後に 1 つ変更したリスト
- 辞書順で前後のリスト

以上のように，前後の値が一意に決定できない代数データ型も存在する．このような代数データ型の値が分岐条件式中で境界値として記述されるプログラムに対して境界値テストをおこなう場合，テストケースを生成するために前後の値として様々な値を用意する必要がある．

2.3 分岐カバレッジ

分岐カバレッジとは，プログラムにおける全ての分岐の内，テストにより実行された割合を用いてテストの信頼性を測定するカバレッジである．境界値テストは，この分岐カバレッジを高い水準に保つことが可能である．一方，ランダムテストでは，低い分岐カバレッジを計測することがある．

分岐カバレッジの例として，**図 7** のプログラムに対する分岐カバレッジを考える．図 7 のプログラムでは，分岐が 5 つ存在する．以下にその分岐を示し，各分岐の条件を**表 1** に示す．表 1 の any は，どのような Int 型の値でもよい場合を表している．

alt1 ：2 行目の等式のトップレベルのパターンマッチ
alt2 ：3 行目の等式のトップレベルのパターンマッチ
alt3 ：3 行目の if 式の True 部

alt4 ：4 行目の if 式の False 部
alt5 ：5 行目の等式のトップレベルのパターンマッチ

分岐カバレッジを 100%にするためには，これら 5 つの分岐を全て網羅するテストケースが必要になる．分岐カバレッジを 100%にするためのテストケースの例を，**図 8** に載せる．`testCase1` は，alt1 のパターンマッチを通り，`testCase2` は，alt2 のパターンマッチを通り，alt3 の if 文の True 部を通る．同様に，`testCase3` は，alt2 のパターンマッチを通り，alt4 の if 文の False 部を通る．そして，`testCase4` は，alt5 のパターンマッチを通るため，図 8 のテストケースを全て用いたテストを行えば，図 7 の関数 `altExam` に関する分岐カバレッジは 100%になる．

2.4 分岐条件式中のリテラルの調査

リテラルとは，プログラム内で使用される数値や文字などを表す定数のことである．2.2 節で述べたように，境界値テストでテストケースとして使用する前後の値は，列挙型の値ならば一意に決定することが可能である．この列挙型の値としてリテラルが存在する．2.1 節の図 5 におけるプログラムのように，リテラルは分岐条件式中の境界値として扱われることがある．このリテラルが分岐条件式中の境界値として Haskell プログラム内で出現する割合を調査し，その調査結果について考察を行う．

2.4.1 Haskell プログラムに対する調査

Haskell プログラムにおける，リテラルが分岐構文の条件式とパターン中の境界値として出現する割合を調査する．その調査方法について以下で述べる．

調査対象 Hackage は，Haskell のパッケージ管理システムとして広く使用されている．この Hackage が管理している 9,987 個のパッケージ [4] 中の Haskell プログラムを調査対象として，リテラルが分岐構文の条件式とパターン中の境界値として出現する割合を計測する．

分岐構文 Haskell プログラムには，条件式を用いた分岐構文として if 式とガードの 2 つが，パターンを用いた分岐構文として case 式と関数のトップレベルにおけるパターンマッチの 2 つの計 4 つが存在する．これらの分岐構文における条件式とパターン中のリテラルの数を計測し，各分岐構文における条件式とパターンの総数に対する割合を計測する．

リテラル 調査で計測するリテラルは，整数である `Int`，浮動小数点数である，`Float`，`Double`，`Rational`，そして，文字である `Char` の 5 つの型の値とする．これは，調査で使用した Haskell の静的解析パッケージ haskell-src-exts[5] におけるリテラルの定義と同様である．これらのリテラルが分岐構文における条件式とパターン中に出現する数を計測する．

上記の調査方法で，リテラルが分岐構文の条件式とパターン中の境界値として出現する割合を計測した．この調査結果を**表 2** に示す．

2.4.2 考察

表 2 から，Haskell プログラムにおいて分岐構文における条件式とパターンの 31%が境界値としてリテラルを含んでいることがわかる．このことから，リテラルに着目すれば，境界値をテストすることが可能になり，分岐カバレッジの向上を期待できることが理解できる．

3. 境界値テストケース自動生成手法

ランダムテストの問題を解決するための本稿のアプローチと，提案手法である境界値テストケース自動生成手法について述べる．そして，提案手法の各 Step について詳細を述べる．

3.1 概要

本稿のアプローチと，提案手法の流れについて述べる．

3.1.1 本稿のアプローチ

2.1 節と 2.2 節では，ランダムテストツール QuickCheck と境界値テストに以下の特徴があることを述べた．

- QuickCheck では，入力を自動生成し，プログラムの満たすべき性質を用いてプログラムの妥当性を確かめることができる．しかし，分岐条件式中の境界値を生成不可能であることが，QuickCheck における信頼性低下の一因となっている．
- 境界値テストでは，境界値を基にテストケースを生成するため，信頼性の高いテストケースを生成できる．

上記の知見と，2.4.1 節の調査結果から，ランダムテストにおける問題点を解決するアイデアを考えた．

- 境界値テストで用いるテストケースの入力さえ自動生成することが可能になれば，QuickCheck の利点を生かすと共に信頼性の高いテストができる．
- リテラルは分岐構文の条件式とパターン中の境界値として用いられるため，境界値の自動生成に使用できる．

本稿では，上記のアイデアから，ランダムテストの問題を解決するために，以下のアプローチを取る．

- 境界値抽出のために分岐構文の条件式とパターン中のリテラルに着目する
- リテラルを基に境界値テストケースを自動生成する

3.1.2 提案手法の流れ

本稿の提案手法では，プログラムの分岐構文における条件式とパターンに含まれるリテラルを静的解析を用いて検出し，このリテラルとその前後の値を基にテストケースの入力を自動生成する．提案手法の流れを**図 9** に示す．提案手法では，テストケース生成のために，以下の 2 つの定義を行う．

- リテラルの再定義
- 前後の値の定義

そして，この定義を用いた以下の 3 つの step でテスト

表 2　Haskell プログラムに対する調査結果

構文の種類	if 式	ガード	case 式	パターンマッチ	合計
条件式, パターン	36,259	84,193	180,846	592,703	894,001
リテラル	7,074	11,835	77,275	177,547	273,731
割合	20%	15%	43%	30%	31%

ケースを生成する．次節以降で，この 2 つの定義と，3 つの step についての詳細を述べる．

step1　：リテラルと変数の抽出

step2　：変数と入力の照合

step3　：テストケースの生成

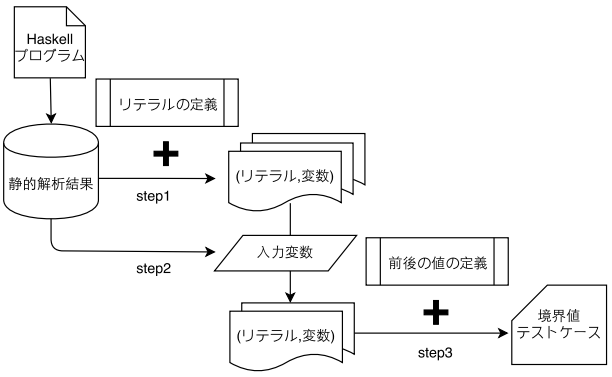

図 9　提案手法の流れ

3.2　リテラルの再定義

リテラルとは，プログラム内で使用される数値や文字などを表す定数のことを表す．しかし，条件分岐で境界値として現れる値は，数値や文字以外の代数データ型も存在する．また，その代数データ型は，ユーザ定義型としてプログラム毎に固有で生成可能である．

上記の問題に対応するために，提案手法で扱うリテラルの再定義を行う．

3.2.1　数値と文字

提案手法では，プログラムの静的解析として，Haskell の静的解析パッケージである haskell-src-exts[5] を用いる．このパッケージでは，プログラムの静的解析を行う際，以下の 3 つをリテラルとして扱う．これらは，一般にリテラルとして定義される値である．提案手法では，このパッケージと同様に，これらの値をリテラルとして定義する．

文字　Char

整数　Int,Integer

浮動小数点数　Float,Double,Rational

3.2.2　リテラルのリスト

2.2 節で述べたように，リストは，特定の代数データ型の要素を n 個持つ n 次元ベクトルの代数データ型である．リストの例として，Int 型のリスト `[1,2,3]` について考える．このリストは，要素が全てリテラルであるため，変域を持たない定数として扱える．このため，提案手法では，

```
1  data Color = C Int Int Int | R | G | B | K
```

図 10　代数データ型 Color

表 3　数値と文字の前後の値

	前値	元の値	後値
Char	'a'	'b'	'c'
Int	0	1	2
Float	2.14	3.14	4.14

表 4　リストに対する前後の値

	前値	元の値	後値
パターン 1	[0,1,2]	[1,2,3]	[2,3,4]
パターン 2	[1,2]	[1,2,3]	[1,2,3,7]

リテラルのリストはリテラルであると定義する．

3.2.3　パラメータを持たない値コンストラクタ

図 10 で示す代数データ型 `Color` の値コンストラクタ `R`，`G`，`B`，そして `K` は，パラメータを持たない．このような値コンストラクタは，パラメータが存在しないため，変域の存在しない定数であると考えられる．このため，提案手法では，このパラメータを持たない値コンストラクタを，リテラルとして定義する．

3.3　前後の値の定義

2.2 節で述べたように，境界値テストでは，テストケースの入力として境界値に加えて，その前後の値を用いることでテストの信頼性を高い水準に保っている．このため，境界値テストケースを自動生成するためには，前後の値の定義が必要になる．このため，3.2 節で再定義したリテラルに対する前後の値を定義する．

3.3.1　数値と文字

本稿でリテラルとして定義した，数値と文字は列挙型であり，前後の値が Haskell の標準ライブラリで定義されている．このため，前後の値はこの値を用いる．

数値と文字に対する前後の値の例を，**表 3** に示す．

3.3.2　リテラルのリスト

2.2 節で述べたように，リストの前後の値は一意に決定できない．このため，提案手法ではリテラルのリストに対して，以下の 2 つのパターンの前後の値を定義する．

パターン 1　全ての要素を前後の値に転置したリスト．

パターン 2　長さを 1 加減算したリスト．長さを 1 減算したときには，末尾の要素を削除する．長さを 1 加算した場合は，末尾にランダムに生成した値を追加する．

リストに対する前後の値の例を，**表 4** に示す．

表 5 抽出したリテラルと変数

	リテラル	変数
データ 1	50	x
データ 2	10	z

```
1 func :: Int -> Int
2 func x y
3   | x <= 50 = x
4   | z <= 10 = y
5   | otherwise = z + 200
6     where z = x + y
```

図 11 func

3.3.3 パラメータを持たない値コンストラクタ

ユーザ定義型の代数データ型は，列挙型ではない場合がある．このため，パラメータを持たない値コンストラクタの前後の値を一意に決定できない可能性がある．この問題を解決するために，本稿では，ユーザ定義の代数データにおけるパラメータを持たない値コンストラクタの前後の値は，同じユーザ定義型の，パラメータを持たない値コンストラクタ全てであると定義する．例えば，図 10 の代数データ型 `Color` のパラメータを持たない値コンストラクタ `R` の前後の値は，他のパラメータを持たない値コンストラクタである `G`,`B`, そして，`K` の 3 個全てとなる．

3.4 提案手法の各ステップ

前節で定義したリテラルとその前後の定義を用いて，テストケースの生成を行うための各 step の詳細を述べる．

3.4.1 step1:リテラルと変数の抽出

step1 では，3.2 節で拡張したリテラルの定義を基に，4 つの分岐構文，if 式，ガード，case 式，そして，トップレベルのパターンマッチにおける条件式とパターン中のリテラルと，そのリテラルと比較されている変数を抽出する．これらの抽出したデータは，リテラルと変数の組として取り扱う．

例えば，**図 11** の関数 `func` からは，**表 5** に示す 2 つのリテラルと変数の組を抽出する．

3.4.2 step2:変数と入力の照合

step2 では，step1 で抽出したデータの持つ変数が入力変数であるか，その変数同士の識別子を用いて確認する．この確認の結果，入力変数と識別子が等しい変数を持つデータのみがテストケースの生成のために残されることになる．

例として，図 11 の `func` について考える．`func` の入力変数は，`x` と `y` の 2 つである．これに対して，step1 では，データ 1 で変数 `x` をデータ 2 で変数 `z` を抽出する．このため，入力変数に存在しない変数 `z` を持つデータ 2 は破棄され，入力変数と照合できた変数 `x` を持つデータ 1 のみが残される．

3.4.3 step3

step3 では，step2 で照合した結果，残されたデータを基

表 6 抽出したリテラルと変数

	入力変数 x	入力変数 y
テストケース 1	49	0
テストケース 2	50	5
テストケース 3	51	13

にテストケースの生成を行う．テストケースの生成では，3.2 節で定義した前後の値を基に，残されたデータ中のリテラルと，このリテラルに対する前後の値を，対応する入力変数への入力として用意する．これに対して，対応する変数がデータに存在しない入力変数には，QuickCheck の生成器を用いてランダム生成した値を入力として用意する．

例として，step2 で抽出されたデータ 1 に対するテストケースを**表 6** に載せる．このテストケースを用いて，QuickCheck では網羅できない境界値のテストを行う．

4. 評価

提案手法の適用実験として分岐カバレッジの測定を行い，生成したテストケースの実例を確認し，これらについて考察する．

4.1 分岐カバレッジ測定

対象プログラムとして，Github で開発されている 38,805 個の Haskell プロジェクトにおける 596,371 個の Haskell ファイルを対象に適用実験を行う．適用実験では，この Haskell ファイルに対して，以下の絞り込みを行う．

(1) QuickCheck を import している Haskell ファイル
(2) 1. のプログラムの内，haskell-src-exts を用いて静的解析可能な Haskell ファイル

596,371 個の Haskell ファイルの内，6,446 個の Haskell ファイルが QuickCheck を import していた．そして，6,446 個のプログラムの内，haskell-src-exts を用いて静的解析可能なファイルは 3,213 個だった．

静的解析が不可能な理由は，以下の 2 つである．

制作者の記述ミス 誤ったコメントの記述方法などの単純な記述ミスが見られた．

GHC 拡張の問題 Haskell におけるパッケージインストールで cabal ファイルを使用する場合，制作者は，GHC 拡張を Haskell ファイルで宣言しないことがある．しかし，haskell-src-exts では，1 つのファイル毎に解析を行うため，GHC 拡張が Haskell ファイルで宣言されていない場合，その拡張に対応できず，解析失敗になってしまう．

例えば，cpp 拡張を用いてプログラムが実装されている場合，Haskell ファイル内で，`#if` といった記述をする．しかし，これは，標準の Haskell の構文ではないため，誤った記述だと判断され，haskell-src-exts では，解析失敗になる．

今回の適用実験では，テスト対象を，この 3,213 個の

Haskell ファイルとする．この 3,213 個プログラムに，関数は 20,419 個存在する．この 20,419 個の関数に，提案手法を適用してテストケースを生成し，そのテストケースから得られる平均分岐カバレッジを計測する．

平均分岐カバレッジ AC は以下の数式 1 で測定する．

$$AC = \frac{\sum_{i=1}^{n} p_i}{\sum_{i=1}^{n} b_i} \quad (1)$$

- n：テストケースを生成できた関数の数
- p：テストケースで分岐を通過した数
- b：関数の分岐の数

テストケースは，以下の 2 つを用意する．

(1) QuickCheck のみで生成したテストケース
(2) 提案手法と QuickCheck を併用して生成したテストケース

20,419 個の関数に対してテストケースを生成した結果，テストケースが生成できた関数の数は 969 個であった．この理由は，以下の 3 つである．

提案手法の問題　テスト対象の関数が，提案手法で自動生成不可能な入力の型を持つ場合発生する問題である．例えば，条件分岐の条件式でパラメータを持つ値コンストラクタが現れた場合，提案手法ではテストケースが生成できなかった．

QuickCheck の問題　テスト対象の関数が，QuickCheck で生成不可能な入力の型を持つ場合，発生する関数である．ユーザ定義の代数データ型に対して，QuickCheck の生成器である `Gen` 型クラスのインスタンス宣言を行っていない場合に発生する．

実行環境の問題　テスト対象の関数で使用する外部パッケージが，今回の実験で使用する Haskell Platform[6] を用いて構築した Haskell の初期の実行環境に，インストールされていない場合，発生する問題である．必要な外部パッケージを全てインストールするためには，パッケージの依存関係を全て解決する必要がある．

提案手法と QuickCheck の問題が発生する関数は，提案手法と QuickCheck のテスト対象外の関数である．また，実行環境の問題に対応することは難しいため，今回は，この 969 個の関数に対して生成したテストケースの平均分岐カバレッジを測定する．

この際，1 つのテスト対象の関数に対する QuickCheck のテストケース生成数を，QuickCheck の既存の設定値である 100 に対して，10,20,40,80,160,320,640,1000,10000 個の 9 パターンを加えた 10 パターンで平均分岐カバレッジを計測する．この 10 パターンのテストケース生成数毎に QuickCheck を用いて生成したランダムテストケースから得られる平均分岐カバレッジを計測した結果と，これらに提案手法で生成したテストケースを加えた，平均分岐カバレッジを計測した結果を**図 12** に載せる．

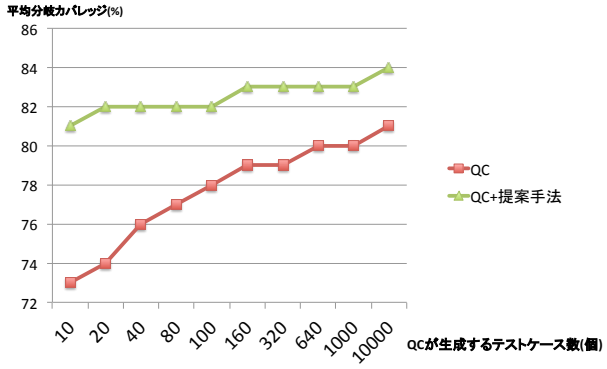

図 12　生成したテストケースの平均分岐カバレッジ

```
1 additionDec size | size <= 2 = ...
```

図 13　関数の整数を用いた分岐

```
1 additionDec 1
2 additionDec 2
3 additionDec 3
```

図 14　提案手法で生成した整数のテストケース

```
1 lexer ('i':'f':cs) = ...
```

図 15　QuickCheck では通過しない関数の文字列を用いた分岐

```
1 lexer ('h':('e':['c']))
2 lexer ('i':('f':['d','f','z']))
3 lexer ('j':('g':['d','m','c']))
```

図 16　提案手法で生成した文字列のテストケース

4.2　生成したテストケースの実例

整数の生成例として，Github のプログラム Xor.hs に存在する `additionDec` 関数の分岐を**図 13** に，この分岐に対して提案手法で生成したテストケースを**図 14** に示す．

文字列の生成例として，Github のプログラム Week4.hs に存在する QuickCheck でランダム生成したテストケースでは通過しない `lexer` 関数の分岐を**図 15** に，この分岐を通過する提案手法で生成したテストケースを**図 16** に示す．

4.3　考察

平均分岐カバレッジの測定結果と，生成したテストケースの実例について考察する．

4.3.1　分岐カバレッジの向上

図 12 に示した実験結果から，1 つのテスト対象の関数に対して QuickCheck のテストケース生成数を標準の 100 個に設定し，QuickCheck のみでテストケースを生成した場合と提案手法と併用してテストケースを生成した場合を比較すると，テストケースが達成する平均分岐カバレッジは，78%から，テストカバレッジで達成すべきといわれる 80%[7][8] を達成する 82%まで向上した，この理由は，図 15 のようにリテラルのリストを分岐条件に持つ関数に

対して，図 16 で示すように，提案手法が分岐を通過するテストケースを生成できるためだと考えられる．これらの結果から，提案手法は，QuickCheck を用いたテストの分岐カバレッジを向上させることで，テストの信頼性を高められることがわかる．

4.3.2 欠陥検出率の向上

図 13 のプログラムでは，「入力が 2“以下” のとき」という分岐が存在する．この実装に対して，「入力が 2“未満” のとき」という分岐が本来の仕様である場合，QuickCheck を用いたテストでは，欠陥の検出は確率的である．これに対して，図 14 に示す提案手法で生成したテストケースを用いれば，必ず欠陥を検出できる．このように，提案手法を用いれば，境界値付近に混入する欠陥を検出できる可能性を高められる．

4.3.3 テストケース数の削減

提案手法では，1 つの関数に対して平均して 6 個のテストケースを生成した．そして，QuickCheck のみでテストケースを生成した場合，1 つの関数に対するテストケース生成数を 10,000 個に設定したとき，平均分岐カバレッジが，カバレッジの基準となる 80%を越えた 81%を達成することに対し，提案手法と QuickCheck を併用した場合，テストケース生成数を 10 個に設定したときに 81%を達成した．この結果から，提案手法で生成したテストケース 1 個が，QuickCheck で生成したテストケース 1,665 個に相当することが確認できる．このため，提案手法を用いることで，効果のないテストケースを削減できることがわかる．

4.3.4 消費時間の削減

969 個のテスト対象の関数に対して，提案手法でテストケースを生成する時間と，提案手法で生成するテストケースと QuickCheck で生成するテストケース 10 個を用いたテストに消費した時間の合計は，8.17 秒であった．一方，QuickCheck で生成する 10,000 個のテストケースを用いたテストに消費した時間の合計は，803.86 秒であった．このため，提案手法を用いれば，約 100 倍の時間効率で，同じ分岐カバレッジを達成するテストが行えることがわかった．

5. おわりに

5.1 まとめ

本稿では，ランダムテストが持つ問題点へ対応するために，プログラム中のリテラルに関する調査を行い，境界値テストケースを自動で生成する手法を提案した．提案手法では，リテラルの再定義，リテラルに対する前後の値の定義を行い，プログラムの静的解析結果を基に，分岐条件式中のリテラルと変数を抽出し，抽出した変数と入力変数の照合を行うことで，境界値テストケースを生成する．提案手法を用いることで，ランダムテストツール QuickCheck で生成するテストケースでは網羅できない分岐を網羅し，分岐カバレッジを 4%向上できること，境界値付近に混入する欠陥を検出する可能性を高めること，提案手法のテストケース 1 個が QuickCheck が生成するランダムテストケース 1,665 個に相当すること，そして，100 倍の時間効率で同じ信頼性のテストが行えることを確認した．

5.2 今後の課題

今後の課題として以下の 2 点が挙げられる．

(1) `Traversable` 型クラスへの対応

`Traversable` 型クラスは，データを格納するコンテナの役割をする代数データ型の集まりであり，リストが最も基本的なインスタンスになる．本稿では，境界値としてのリストをテストケースとして生成可能にし，分岐カバレッジを大きく向上させた．

この知見から，`Traversable` 型クラスのインスタンスである，木構造や配列などもテストケース生成可能にすれば，更に分岐カバレッジを向上させることが可能だと考えられるため，この機能を提案手法に追加することが今後の課題といえる．

(2) パラメータを持つ値コンストラクタへの対応

現在は，複雑なデータ型の値について，その値が定数でもリテラルとして抽出することができない．このため，分岐条件式中に複雑なデータ型の定数が境界値として現れた場合，テストケースが生成できず，分岐カバレッジが低下した．

この知見から，どのようなデータ型の値に対しても定数かどうか判別可能にすれば分岐カバレッジの向上に繋がるため，この機能を提案手法に追加することが今後の課題といえる．

謝辞 本研究は科研費 24300006 の助成を受けたものである．

参考文献

[1] Yogesh Singh. *Software Testing*. November 2011, 2011.
[2] カバレッジ (網羅率) 分析とは. https://www.techmatrix.co.jp/t/quality/coverage.html.
[3] オープンソースと関数型言語によるプログラム解析とカバレッジテスト自動実行, 2010.
[4] Hackage. http://hackage.haskell.org/.
[5] The haskell-src-exts package. https://hackage.haskell.org/package/haskell-src-exts-1.13.5.
[6] Haskell Platform. https://www.haskell.org/platform/.
[7] テストカバレッジ. http://bliki-ja.github.io/TestCoverage/.
[8] コードカバレッジを使用した, テストされるプロジェクトのコード割合の確認. https://msdn.microsoft.com/ja-jp/library/dd537628.aspx.