

分岐カバレッジ向上を目的 として静的解析を用いて テストケース自動生成

山田誠也 大久保弘崇 粕谷英人 山本晋一郎

情報処理学会報告研究所 2017/3/12

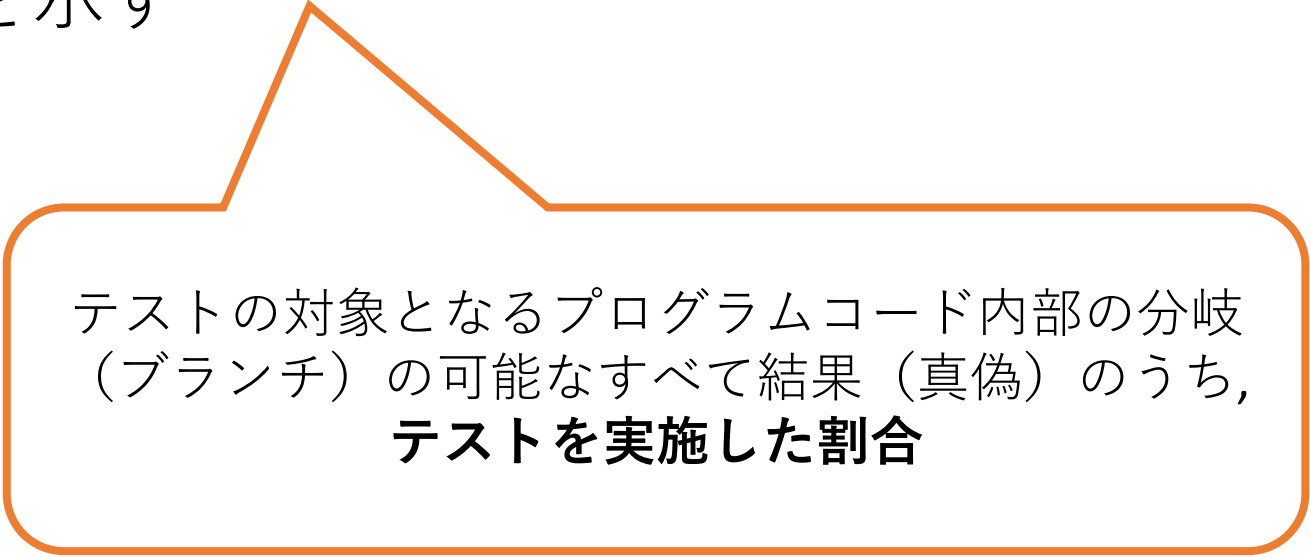
60216114 作島大智

目次

- 概要
- 背景
- ランダムテスト (Quick Check)
- 境界値テスト
- 提案アプローチ
- 提案手法
- 評価と考察

概要

- 簡易テストケース自動生成手法の提案
- 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



テストの対象となるプログラムコード内部の分岐
(ブランチ) の可能なすべて結果 (真偽) のうち,
テストを実施した割合

分岐カバレッジの例

テストの対象となるプログラムコード内部の
分岐(ブランチ)の可能なすべて結果(真偽)のうち
テストを実施した割合

```
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

各分岐の条件

```
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

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

※anyは任意の値

```
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

背景

- ソフトウェアの信頼はソフトウェアテストによって担保されている

- 工程の50%を占める大きい開発コスト

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

- ランダムテストはソフトウェアの内部構造を考慮しないため信頼性は高くない

手法

ランダムテスト

- QuickCheck

境界値テスト

- 列挙型・代数型データ群

ランダムテスト

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主導で
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手法のこと。

QuickCheck

関数型言語Haskellで開発されたランダムテストツール

- ・ プログラムが満たすべき性質を関数として記述
- ・ 関数の性質を検査するもの

入力は自動生成ランダムな値,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.

反する結果が出た場合,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.

使用例

myReverse 「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

prop_twice : **myReverse**をリストに2回
適用したら元のリストに戻る

```
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

出力結果

- ランダムに生成した**100**個の入力すべてが **myReverse** の満たすべき性質 **prop_twice** をパスしたことを示している

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

欠陥のあるプログラム

「長さ 1 のリスト [x] に対して空リスト [] を返す」
という誤ったコードの場合

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

図 3 欠陥のある myReverse

出力結果

- 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す.

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

QuickCheckの効果

膨大な量であるテストケースを手動で作成する時間コストを軽減できる.

必要な負担	軽減される負担
性質を記述する	テストケースを考える負担

ランダムテストの問題点

「入力Xが50以下の時100を加算し,
それ以外の時は入力Xを返す」関数add100

```
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

問題点

- 境界値を網羅するテストケースを生成できない
- “**以下**”を“**未満**”と誤った場合も結果は変わらない

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

改善できない理由

- 整数50を与えたとき実装の欠陥が明らかになる

整数であるInt型(4byte)の値が取りうる値の範囲は
-2147483648～2147483647

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

手法

ランダムテスト

- QuickCheck

境界値テスト

- 列挙型・代数型データ群

境界値テスト

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

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

入力 出力

図 6 境界値テストケース

列挙型

境界値に加えて前後の値をテストケースの入力として使用する

型	境界値	前後の値
Int	10	9,11
Char	b	a,c

代数型

リストの前後の値は一意に定められない

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

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

提案アプローチ

分岐条件内のリテラルの調査

リテラル：プログラム内で使用される数値や文字などを示す定数.**変数の対義語**.

分岐条件式中の境界値として扱われることも
→境界値テストの効率を向上できるかも



Haskellプログラム内で出現する割合を調査

調査結果

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%

リテラルに着目すれば境界値をテストすることが可能となり分岐カバレッジの向上を期待できる

分岐カバレッジの例

テストの対象となるプログラムコード内部の
分岐(ブランチ)の可能なすべて結果(真偽)のうち
テストを実施した割合

```
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

各分岐の条件

```
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

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

※anyは任意の値

```
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

提案手法

提案手法

- 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

提案手法の流れ

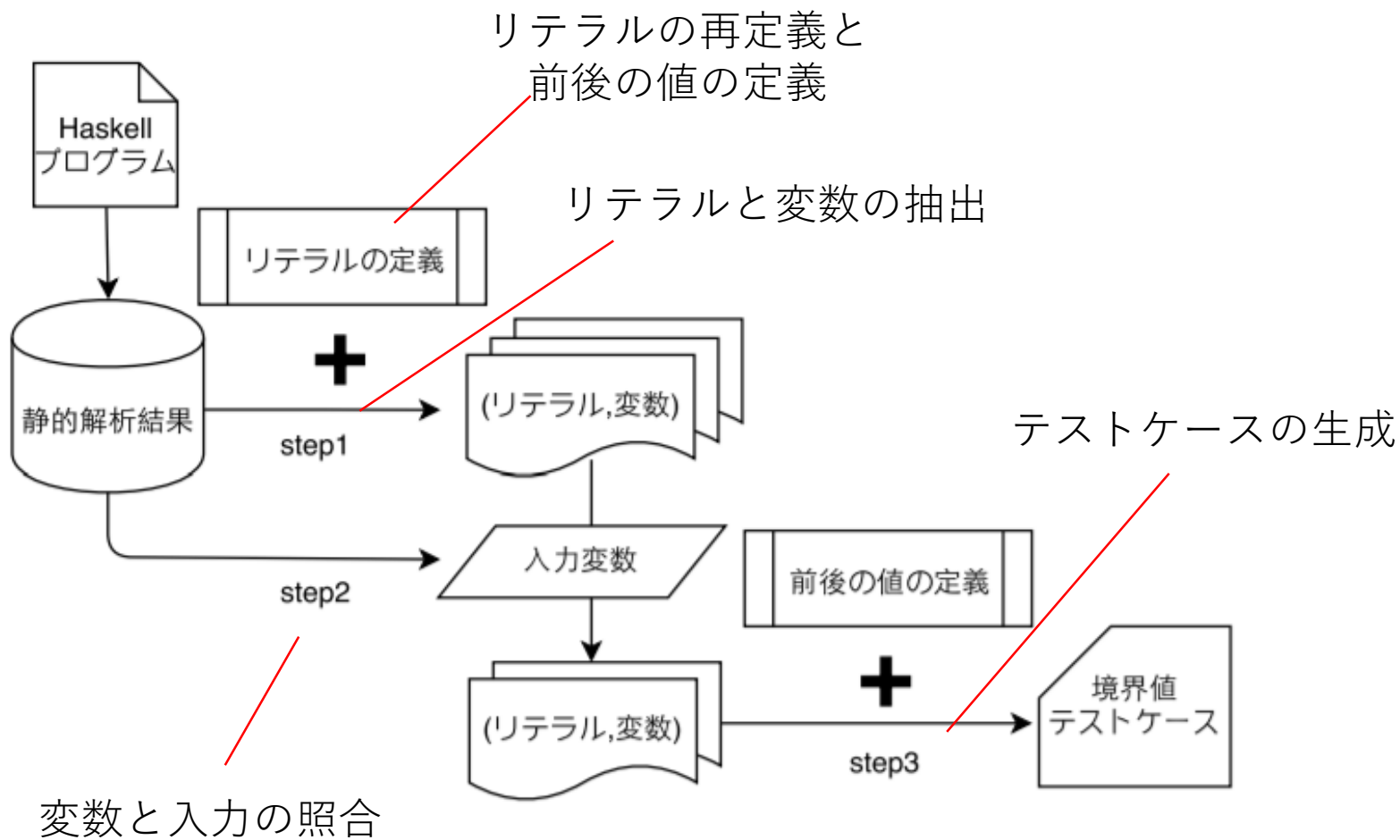


図 9 提案手法の流れ

リテラルの再定義

- 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.
- 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.

数値・文字の再定義

数値・文字は一般にリテラルとして
定義される値に再定義する

種類	型名
文字	Char
整数	Int, Integer
浮動小数点数	Float, Double, Rational

代数の再定義

リストは要素[1][2][3]すべてがリテラルなので
変域を持たない定数として扱える



リテラルのリスト[1,2,3]はリテラルであると定義

パラメータを持たない 値コンストラクタの再定義

代数データ型 **Color** の値コンストラクタ **R, G, B, K** は
パラメータを持たない。

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

図 10 代数データ型 Color

変域の存在しない定数である。



パラメータを持たない値コンストラクタを
リテラルと定義する。

数値・文字・代数の前後の再定義

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]

値コンストラクタ前後の再定義

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



Rの前後の値は**GBK**の3つ

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

図 10 代数データ型 Color

提案手法の流れ

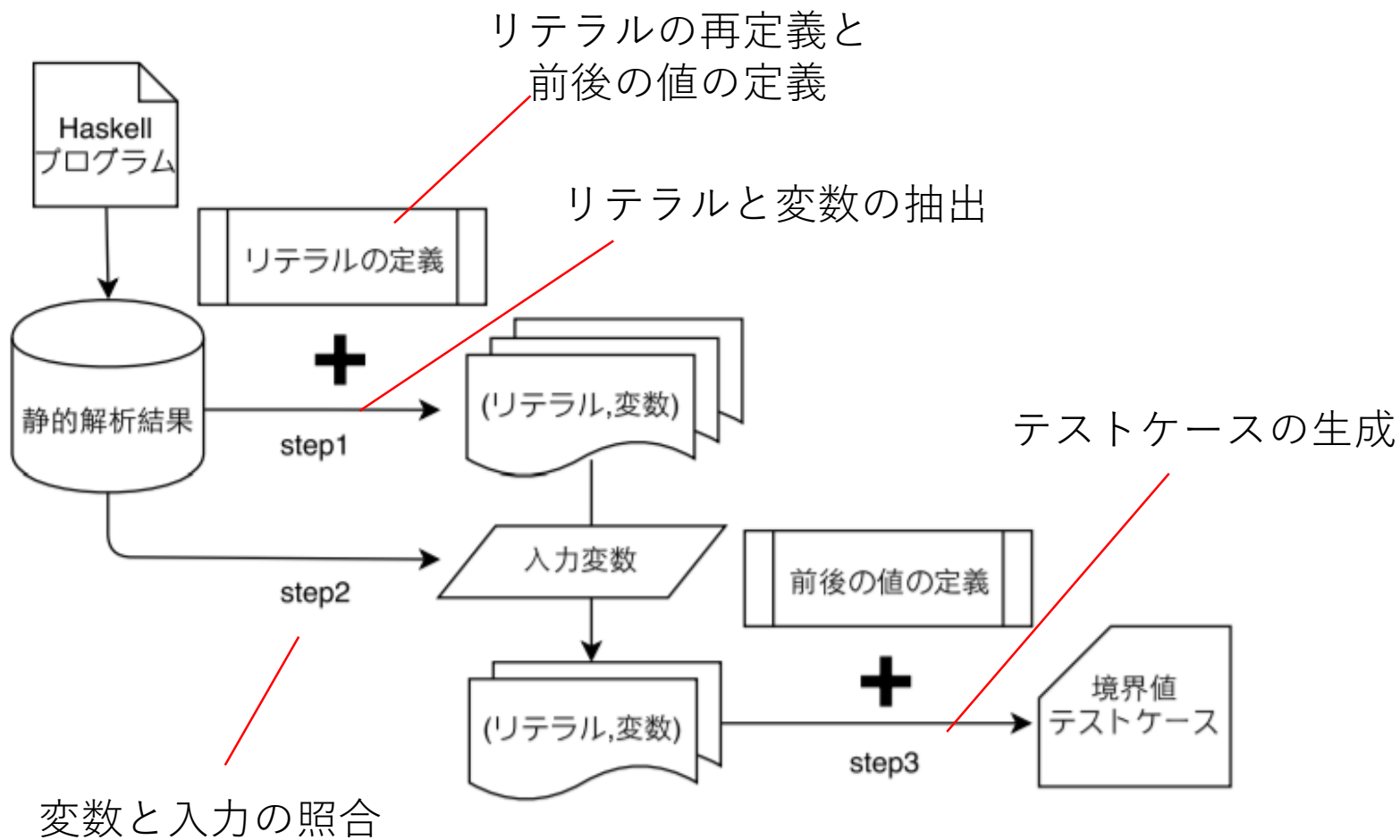


図 9 提案手法の流れ

Step1: リテラルと変数の抽出

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

Step2: 変数と入力の照合

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

入力x,出力y

```
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

Step3: テストケースの生成

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

ランダムに生成

評価

平均分岐カバレッジACは以下の数式で測定

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

- 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

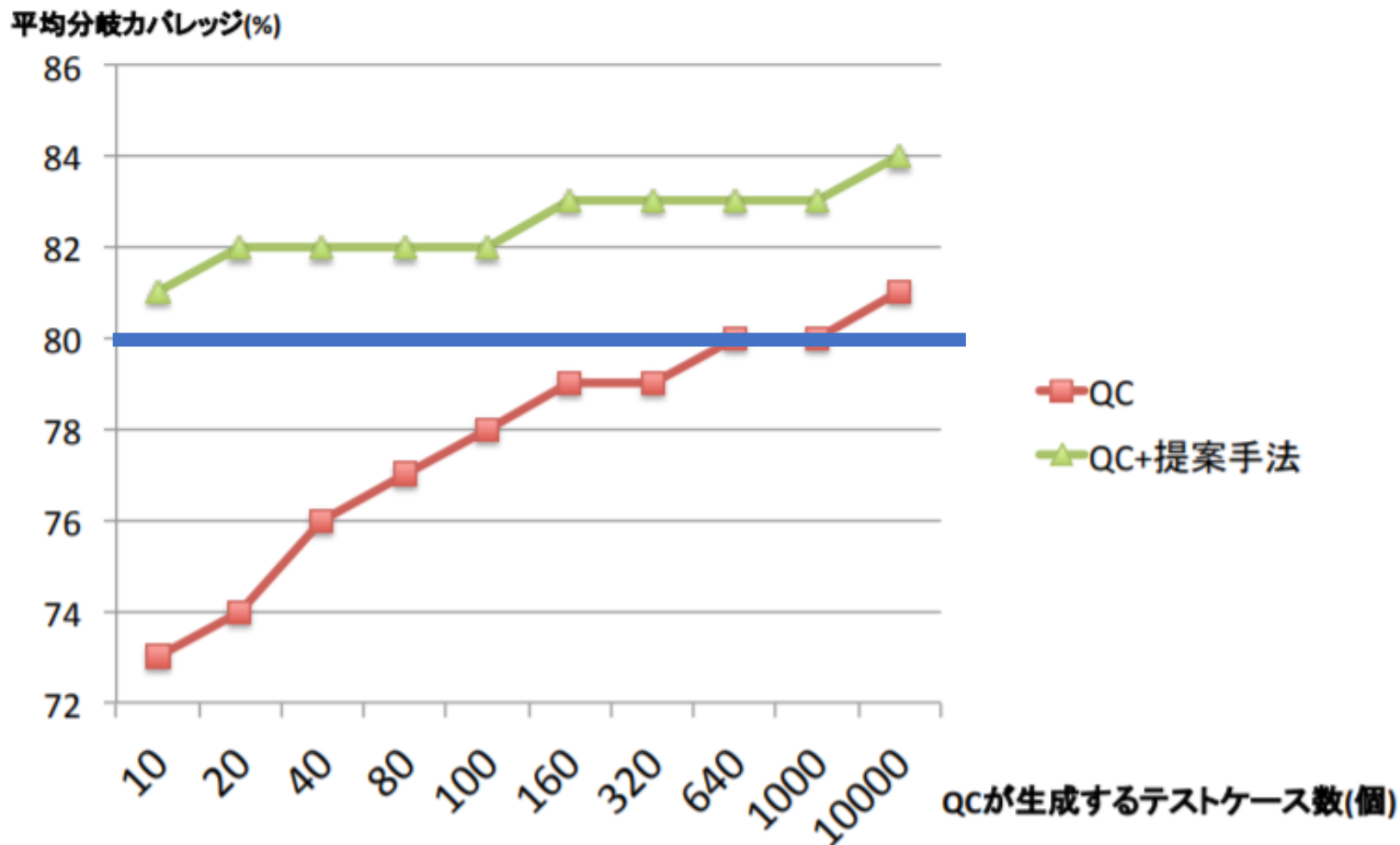
テストケース

(1)QuickCheckのみで生成

(2)提案手法とQuickCheckを併用して生成

最終的に対象とする関数は969個

平均分岐カバレッジの推移



考察

テストカバレッジの**80%**達成において
テストケース数の削減や時間的コストの軽減

	QuickCheck のみ	提案手法との 併用
81%達成までの テストケース数	10000個	10個
消費時間	803.17秒	8.17秒

今後の課題

- テストケース数の増加がAC向上につながる
→ 木構造や配列などもテストケース生成可能
- 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
→ どのようなデータ型に対しても判別可能に

所感

- パワポ見るだけで理解できるように
まとめるの大変
- 実際のコードを見て状況を整理することが
時間かかった
- パワポが大変