等価変換に基づく C コンパイラのランダムテストにおける制御文およびデータ型の拡張

高倉 正悟[1]　岩辻 光功[1]　石浦 菜岐佐[1]

概要：本稿では, 等価変換に基づく C コンパイラのランダムテストにおける不具合検出能力の向上を目的として, プログラム生成を強化する手法を提案する．等価変換に基づくランダムプログラム生成法は, 生成規則のみに基づく生成法に比べて意味まで考慮したプログラムの生成が可能であるが, この方法で生成可能な構文はこれまで代入文, if 文, for 文に限られており, 変数もスカラ変数しか扱えなかった．本稿では, 制御文に関しては関数呼び出し, while 文, switch 文を生成可能にするとともに, データ型に関しても配列, 構造体, 共用体およびその任意回のネストを生成可能にする．本手法をランダムテストシステム Orange4 に追加実装した結果, 従来では検出できなかった不具合を検出できるようになり, GCC-8.0.0 および LLVM/Clang-6.0 (それぞれ 2017 年 12 月時点における最新バージョンの開発版) において新たな不具合を検出することができた.

キーワード：コンパイラ, ランダムテスト, 等価変換, 最小化, 信頼性

Enriching Generation of Control Statements and Data Structures for Random Test of C Compilers Based on Equivalence Transformation

Takakura Shogo[1]　Iwatsuji Mitsuyoshi[1]　Ishiura Nagisa[1]

Abstract: This article proposes a method of enriching program generation in random testing of C compilers based on equivalence transformation on test programs. While the conventional method based on equivalence transformation can only generate programs with scalar variables, assign statements, if and for statements, the proposed method enables generation of arrays, structs, unions, as well as while and switch andfunction calls. Orange4 C compiler test system extended with the proposed method has detected bugs in the latest development versions of GCC-8.0.0 and LLVM/Clang-6.0 which had been missed by the existing test methods.

Keywords: compiler, random test, equivalence transformation, minimization, reliability

1. はじめに

コンパイラはソフトウェア開発の基盤ツールであり, 高い信頼性が求められる．コンパイラの不具合はソフトウェアの品質に致命的な影響を与えるため, コンパイラには徹底的なテストが求められる.

コンパイラのテストは膨大な数のテストプログラムから成るテストスイートを用いて行われるが, テスト数が有限である以上, 不具合の見逃しは避けられない．テストスイートによるテストを補う手法として, ランダムに生成したプログラムを用いて対象のコンパイラをテストするランダムテストが行われる [1].

C コンパイラのランダムテスト手法はこれまでに様々なものが提案されているが, 傑出したものの一つが Csmith [2] である．Csmith は C 言語の広範な文法をカバーしており, 2010 年までの 3 年間に GCC および LLVM/Clang の不具合をそれぞれ 79 件と 202 件検出し, これらオープンソースコンパイラの品質向上に大きく貢献している.

Csmith は, 基本的に文法規則だけに基づいてランダムなテストプログラムを生成するため, 比較的自由に広範囲の構文を生成できる．しかし, 生成するプログラムの実行結果は考慮していないため, そのままではプログラムが実行中にゼロ除算や配列の領域外アクセス等の未定義動作

[1] 関西学院大学
Kwansei Gauin Uniersity, 2–1 Gakuen, Sanda, Hyogo, 669–1337, Japan

© 2018 Information Processing Society of Japan

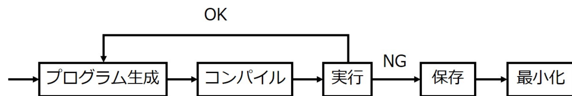

図 1: コンパイラのランダムテストの流れ

を引き起こす可能性がある. Csmith では, このような不正なプログラムの生成を回避するため, 例えば除算は必ず（B !=0 ? A/B : A）のような式に限定して生成する等, 文や式の形式に制限を課している. 広範囲の構文をカバーできる一方で, このような制約が生成できるプログラムを限定している.

これに対し, Orange3 [3], Orange4 [4] では, プログラムの意味情報を併用してテストプログラムを生成する. プログラム生成時にプログラム中の変数や式の値を把握することにより, 文や式に Csmith のような制約を加えることなく不正なプログラムの生成を回避する. 特にコンパイラの算術最適化をテストするための複雑な式を生成可能であり, Csmith では検出できなかった GCC や LLVM の不具合を検出している. しかしこの方法では, プログラム生成時に構文木だけでなく意味に関するデータ構造も構築する必要があるため, プログラム生成処理が複雑になる. このため, Orange3, Orange4 では, 変数はスカラ型のみであり, 制御文も if 文と for 文しか生成できていなかった.

本稿では, Orange4 のプログラム生成方法を拡張することにより, 意味情報を併用した C コンパイラのランダムテストにおいて, Csmith と同程度の制御文とデータ型を含むプログラムを生成する手法を提案する. これは, プログラム中の式の値が一意に定まるような制約を加えるという方針により実現する. 制御に関しては, if 文, for 文に加えて while 文, switch 文, 関数呼び出し文を生成可能にする [5]. また, データ型に関しては, 配列, 構造体・共用体およびその任意回のネストを生成可能にする.

本手法に基づくランダムテストシステムを Orange4 に追加する形で実装した結果, Csmith と同等の範囲の構文を生成可能になり, Csmith よりも不具合検出能力を向上させることができた. さらに, 実験当時の最新版の C コンパイラである GCC-8.0.0, LLVM/Clang-6.0 において, これまでの手法では検出できない不正コード生成を 2 件検出することができた.

2. 関連研究

2.1 コンパイラのランダムテスト

コンパイラのランダムテストの流れを図 1 に示す. ランダムに生成したテストプログラムをテスト対象のコンパイラでコンパイルし実行するという処理を, 繰り返し実行する. もし, コンパイラがクラッシュしたり, プログラムの実行結果が正しくない等のエラーが生じれば, そのプログラム (エラープログラムと呼ぶ) を保存して分析する. エラープログラムは数千行規模に及ぶことがあるため, エラーの原因を分析するためにプログラムを縮約し, エラーが起こるできるだけ小さなプログラムを求める「最小化」の処理が行われる.

コンパイラのランダムテストでは, ランダムに生成したプログラムのコンパイル結果の正誤判定ををいかにして行うか, ゼロ除算等の未定義動作の生成をいかにして避けるかの二点が課題になる.

コンパイル結果の正誤判定の方法は差分法と期待値計算法に大別される. 差分法は, テストプログラムを二つ以上の異なる (あるいは異なるバージョンの) コンパイラでコンパイルし, 実行結果を比較するというものである. Csmith [2] は差分法に基づいている. 期待値計算法は, プログラムの計算結果 (期待値) を生成時に把握し, これに基づいて正誤判定を行うものである. 関数呼び出しのテストを行う Quest [6] や算術最適化のテストを行う Orange3 [3], Orange4 [4] は期待値計算法に基づいている. 差分法では未定義動作の回避が課題になる. Csmith では, 文法に制限を加えることにより未定義動作を回避しているが, 生成できるテストプログラムが未定義動作に関して悲観的なものに制限されてしまう. 期待値計算法では未定義動作の回避は比較的容易に行えるが, 複雑なプログラムを生成する際には, その意味情報をどのようにして保持するかが課題となる.

一方, プログラムの生成に関しては, 生成規則に基づく方法と等価変換に基づく方法がある. Quest, Csmith, Orange3 は前者に基づいている. 後者は既存のプログラムに対して計算結果を変えないような変換を行って新たなプログラムを生成する手法である. Proteus [7], Athena [8], Hermes [9] は Csmith で生成したプログラムから新たなテストプログラムを生成している. Orange4 は, 自明なプログラムから等価変換の繰り返しにより長いランダムプログラムを生成する.

2.2 Csmith

Csmith [2] が生成するテストプログラムの例を図 2 に示す. データ型はスカラ変数, 配列, 構造体・共用体, ポインタが生成される. 制御文に関しては if 文, for 文, 関数呼び出し, goto 文など広範囲の構文が生成される.

Csmith ではプログラム生成時に変数や式の実行時の値はわからないため, プログラムが未定義動作を引き起こさないように工夫されている. 例えば, 26 行目の `safe_mul_func_uint8_t_u_u` は乗算を行うマクロであるが, オペランドを検査してオーバーフローが起きない場合には乗算結果を, そうでなければ第 1 オペランドを値とするようになっている. 20 行目では, for 文が無限ループに陥らないよう, バウンドは定数のみとしている. 配列の添え字は, 22 行目のように範囲が既知のループ変数か, 24 行目のように値が既知の変数に定数を加えたものに限っている. また, 式の値が制御できないと無限ループに陥る可能性のある while 文や case 節が一つも選択されない可能性が高くなる switch 文は生成されない.

Csmith で検出したエラープログラムの最小化には, 汎用の最小化ツールである C-Reduce [10] が使用される. C-Reduce は C プログラムのソースコードとエラーを判定するコマンドラインを与えると, エラーが起こる縮約されたプログラムを出力する. C プログラムを縮約する種々の変換を実装しているが, その適用によって未定義動作が発生することがあるため, 静的解析ツールを用いてそのような変換の適用を排除している. 縮約による未定義動作が頻発する場合には最小化に長時間を要し, 最小化に失敗するこ

© 2018 Information Processing Society of Japan

```
 1: #pragma pack(push)
 2: #pragma pack(1)
 3: struct S1 {
 4:   volatile int32_t f0;
 5:   struct S0 f4;
 6:   volatile uint8_t f7;
 7: };
 8: #pragma pack(pop)
 9: union U3 {
10:   const uint32_t f0;
11:   const volatile int64_t f1;
12:   int32_t f2;
13: };
14: ...
15: static uint32_t * func_1(void)
16:
17: {
18:   ...
19:   if (func_2(g_31[3])) {
20:     for (i = 0; i < 1; i++)
21:       l_96[i] = &g_97;
22:     (****g_1658) = ((((safe_mod_func_int32_t_s_s(
23:     g_450[(g_133.f6 + 2)][(g_133.f6 + 3)], g_62)) >=
24:     l_172), 0x5318L), (g_1725, l_1726));
24:     (*p_61) = ((*g_821) |= (safe_lshift_func_uint8_t_u_u(
25:     ((safe_mul_func_uint8_t_u_u(g_22.f1, l_1371)) |
26:     ((++l_1664) > l_1667)), l_1336)));
27:   }
28:   ...
29: }
30: ...
31: int main ( int argc, char* argv[])
32: {
33:   ...
34:   func_1();
35:   ...
36: }
```

図 2: Csmith が生成するテストプログラムの例

ともある.

2.3 Orange4

Orange4 [4] が生成するテストプログラムの例を図 3 に示す. Orange4 では全ての変数や式の値をプログラム生成時に把握しているため, 式に未定義動作を回避するガードを設けることなく複雑な式を生成している. また 12 行目のように for 文のバウンドにも式を生成している. さらにプログラムの実行結果の正誤判定は, 22–24 行目のように期待値との照合により行う. しかし制御文は for 文 と if 文のみであり, 変数もスカラ型しか生成することができない.

Orange4 では, `return 0;` だけを含む自明なプログラムからプログラムの等価変換 (文の追加と式の複雑化) を繰り返し適用することにより複雑なテストプログラムを生成する. 算術式は, 図 4 に例を示すように, 等価変換により定数から生成する. まず最初に式の値を決定し, それを値が同じになるような式に展開するという変換を繰り返すことにより式を生成する. この際, オペランドを適切に選択することにより未定義動作が発生しないようにできる. また, オペランドに境界値を生成することも可能である.

Orange4 では, プログラム中の各文は高々 1 回しか実行されないという制約を設けている. この制約により, プログラム中で参照される変数および式の値は一意に定まる. Orange4 はプログラムの解析木表現と共に, 全ての変数, 式, および部分式に対してこの値をプログラムの意味情報として保持している.

この制約を満たすために, for 文の生成においては, ループ回数は 0 回または 1 回になるように制御変数のバウンドの式の値を設定している. コンパイラは, 式の値が定数に畳み込める場合にはループを削除する最適化を行うが,

```
 1: #define OK()
 3: #define NG(fmt,val) __builtin_abort()
 3: const volatile signed int x9 = -59;
 4: int main (void)
 5: {
 6:   static unsigned long long x0 = 7LLU;
 7:   static const volatile signed long x1 = 0L;
 8:   ...
 9:   int t0 = 46;
10:   signed long t1 = 297271L;
11:   ...
12:   for( i = x9*x6-x8; i < x5+x5; i -= x7+x3 ) {
13:     t0 = x3|x0*x2-x4;
14:     t1 = x5*x5-x6+x2/x7;
15:     if( x1<<x1 ) {
16:       t2 = x8>>x4/t0+x10*x10+x11;
17:     }
18:     else {
19:       t3 = x14|x16;
20:     }
21:   }
22:   if (t0 == 120) { OK(); } else { NG("%d", t6); }
23:   if (t1 == 220) { OK(); } else { NG("%d", t6); }
24:   if (t3 == 22) { OK(); } else { NG("%d", t6); }
25:   return 0;
26: }
```

図 3: Orange4 が生成するテストプログラムの例

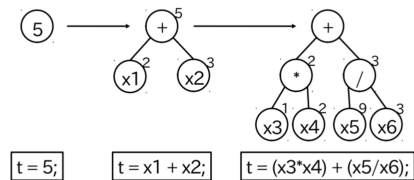

図 4: Orange4 における式の s 生成

式中に volatile 変数が生成されていて式を定数に畳み込めない場合には, ループ回数が 2 回以上の場合と同じコード生成を行うため, この制約が不具合検出能力を著しく損なうことはないと考えられる.

Orange4 では, エラープログラムの最小化機能を組み込みで実装している. エラープログラムに対する縮約変換を, それ以上どの変換を適用してもエラーが消失する直前まで繰り返す. この変換は, 意味情報 (変数と部分式の値) を持った解析木を入力として行われるため, プログラムの縮約に際して未定義動作を引き起こすことがない.

一方で, Orange4 では, プログラム生成のために解析木と共に意味情報を構築しなければならないため, Csmith に比べるとカバーできる文法の範囲が狭くなっている.

3. Orange4 の制御文およびデータ型の拡張

本稿では, Orange4 が生成するテストプログラムの構文について, 制御文およびデータ型の拡張を実現する手法を提案する. 制御文に関しては while 文, switch 文, 関数呼び出しを [5], データ型に関しては配列, 構造体, 共用体を任意回ネストしたものを新たに生成できるようにする.

3.1 データ型の拡張

3.1.1 概要

配列, 構造体, 共用体を導入しても, 式中で参照されるのはその一要素なので, Orange4 の式生成法をそのまま適用できる. 図 6 に拡張した式生成の例を示す. 定数 5 を加算で展開する際に, スカラ変数だけでなくネストされた配列・構造体の要素 (`x2.m1[2][3].m4`) を参照できるようにする. 次に, 配列の要素参照を演算 (`[][]`) とみなして, 添

© 2018 Information Processing Society of Japan

```
 1: int func0(int a0[5], long a1) {
 2:   ...
 3:   t50 = a0[2] + x43;
 4:   if (t50 == -59000) {OK();} else {NG();}
 5:   return t50 + x51 - a0[4];
 6: }
 7: int main(void) {
 8:   int x6[5] = {1227,4,-59433,-106,24};
 9:   int t42[3][5] = { {-3290,559,23085,26668,-38567},
      {358148,8,4,35388,85600},
      {12594474,-293697,979,-125774,1138055} };
10:   ...
11:   t42[1][2] = (x4 - x3) - func0(x6, x11 * x12) ;// = 100
12:   unsigned int t48[x6[1] + t42[1][2]];
13:   t48[0] = x6[1] * x11;
14:   t42[x12][t4[4]] = (x5 && t48[x16 + t1]) + t2;
15:   ...
16:   if (t42[1][2] == 100) {OK();} else {NG();}
17: }
```

図 5: 配列を含むプログラムの例

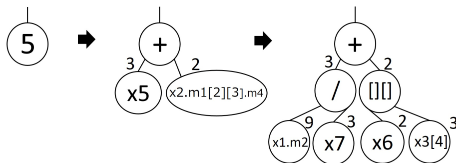

図 6: 添え字の式展開を含む式生成の例

え字の定数 (2, 3) を式に展開する．この式中に再帰的に配列や構造体・共用体が現れても良い．この方法により, 配列添え字に境界を超えることなく複雑な式を生成できる．また, 式の値が既知であることを活かして, 可変配列を宣言することもできる．

また, 後述する関数呼び出しの引数には, 配列や構造体・共用体の変数を渡せるようにする．

3.1.2 配列

本手法で生成する配列を含むプログラムの例を図 5 に示す．配列の基本型, 次元数, 要素数, 初期値はランダムに決定する．従来のスカラ変数と同様に, 配列の要素は, 11–14 行目のように式中で参照することができ, `x6[1]` のように同じ要素を複数回参照することもできる．また, 配列要素に代入を行って, その後の算術式内で参照することもできる．添え字には任意の式を生成することができる．この式は図 6 のように添字の値から式展開を行い生成するので, 配列の境界外へのアクセスが起こることはない．

本手法では 12 行目のように可変配列を生成することもできる．可変配列の要素は, 13–14 行目のように, 代入された後, 通常の配列と同様に代入や参照が可能である．式の値が既知であるため, サイズが負数や大きすぎる数になることがない．

さらに, 本手法では 11 行目のように配列を関数に渡すことも可能である．渡された配列は, 3 行目のように関数内で参照することができる．

要素への代入や参照は, 配列の各要素にスカラ変数と同等の意味情報を持たせ, 式生成で使用する変数表にそれぞれの要素を入れ, スカラ変数と同じように扱えるようにすることで実現する．また, 配列を関数に渡すのは, 関数呼び出し文を生成するとき, その関数を生成するときに決めた引数の値と同じ値の配列を, 呼び出し元の関数のローカルに宣言することで実現する．

```
 1: struct s0 {
 2:   signed int m0;
 3:   unsigned int m1;
 4: };
 5: union u0 {
 6:   signed int m0;
 7:   struct s0 m1;
 8: };
 9: struct s1 {
10:   signed int m0[2][2];
11:   union u0 m1;
12:   struct s0 m2[2];
13:   union u0 m3[5];
14: };
15: void func0(struct s1 a0, long a1) {
16:   t50 = a0.m0 + x43;
17:   if (t50 == -59000) {OK();} else {NG();}
18: }
19: int main(void) {
20:   struct s0 t0 = {64, 33};
21:   union u0 x1= {16};
22:   struct s1 x2 = {{{0, 3}, {2, 8}},
      {66}, {{125, 33}, {64, 666}}};
...
23:   func0(x2, x33 + x1.m0);
24:   t0.m1 = x3 + x1.m0 - x2.m2[x3-x4].m1 - x15;
25:   if (t0.m1 == 647){}
26:   ...
27: }
```

図 7: 構造体・共用体を含むプログラムの例

3.1.3 構造体・共用体

本手法によって生成する構造体・共用体を含むプログラム例を図 7 に示す．1–14 行目のように, メンバー変数にはスカラ変数, 配列, 構造体・共用体, 構造体・共用体の配列を任意に生成することができる．ただし, 共用体は 21 行目のように 1 番目のメンバ変数で初期化し, 1 番目のメンバ変数のみ算術式中で参照できる．

3.2 制御文生成の拡張

3.2.1 概要

関数呼び出しと while 文に関しては, いくつかの制限を課すことによりプログラム中で参照される変数の値が一意に定まるようにする．逆に, プログラム中の式の値が生成時にわかることを活かし, 無限ループを引き起こす while 文や case 節が選択されない switch 文の生成を避ける．

3.2.2 関数呼び出し

プログラム中で参照される変数の値が一意に定まるようにするため, 本稿の手法で生成する関数には「呼び出し時の引数の値は毎回同じであり, 関数の返り値も毎回同じ」という制約を課す．

図 8 に関数の定義と呼び出しのコード片の例を示す．8 行目で `f1` の返り値は定数 5 から展開したものであり, 値は必ず 5 になる．13–14 行目の `f1` の呼び出しでは引数の式はいずれも (5, 9) から展開したものであり, 2 回とも同じ値で呼び出される．式中に volatile 変数が含まれていれば, コンパイラは同じ引数で呼び出されていることや, 同じ値を返していることは判定できない．6–7 行目では, 引数の値が生成時に既知であることを利用して渡された引数の値を正しいかどうかを検査しているが, これは差分法ではできなかったことである．

3.2.3 while 文

ループ内で参照される変数の値を一意にするため, while 文でも for 文と同様にループの繰り返し回数を 0 か 1 に限定する．

繰り返しの回数が 0 の while 文は, 図 9 (a) のように, 継

© 2018 Information Processing Society of Japan

```
 1: int f1 ( unsigned short a1, long a2 ){
 2:     t1 = ((x23 * a1) >> x2 & x8);
 3:   t2 = (t1 | x5);
 4:   ...
 5:   t99 = (x15 * x2) - (x10 - a1);
 6:   if( a1 == 5 ){OK();} else{NG();}
 7:   if( a2 == 9 ){OK();} else{NG();}
 8:   return (a1-x3);
 9: }
10: int main(void){
11:   int x1 = 34;
12:   ...
13:   int t3 = ((x2 + f1((x2*x7), (x1-x5))) & x8)
14:   int t4 = (t3 < x7) >> f1((t3/x9), (x8-x7)) / x5;
15:   ...
```

図 8: 関数宣言と呼び出しの例

```
while( 0 ){           →   while( ((x3<<t7)&(a8%x5)) ){
  文リスト                   文リスト
}                         }
```

(a) ループ回数が 0 の while 文

```
while( 1 ){             →   while( (t2*x4)!=(x1>>x10) ){
  文リスト                     文リスト
  if ( 1 ){break;}            if( (a1-x11)-(x2/t5) ){break;}
}                           }
```

(b) ループ回数が 1 の while 文

図 9: while 文の生成例

続条件が 0 の while 文の原型から開始して, 継続条件を等価な式に展開することにより生成する. 文リストにはランダムな文のリストを再帰的に生成する. 同様に, 繰り返し回数が 1 の while 文は, 図 9 (b) のように, 継続条件と中断条件を 1 とした while 文の原型から生成する. 継続条件や中断条件の式が定数畳み込みで 0 や 1 に縮約される場合はループを削除する最適化が行われる (これもテスト対象) が, 条件式中に volatile 変数が含まれている場合には, 繰り返し回数が 2 以上の場合と同じ最適化が行われる.

3.2.4 switch 文

本手法で生成する switch 文の例を図 10 に示す. まず, switch 文中に生成する case 節のラベルの集合 (この例では {3, 4}) を決定し, switch 文の選択式 (この例では 4) をこの集合および他の適当な値 (default 節への分岐用) からランダムに選択する. そこから, 式展開と文の挿入を繰り返して switch 文を生成する.

3.3 最小化

エラーを検出したプログラムの最小化は, 本稿で追加した制御文やデータ型に関する縮約の変換を追加することにより実現する. 例えば, 関数呼び出しに関しては, 関数呼び出しの返り値での置き換え, 一度も呼ばれていない関数の定義の削除等を行う. while 文では, ループ本体が空の while 文の削除, 繰り返し回数 0 の while 文の削除, 繰り返し回数 1 の while 文のループ本体内文リストへの置換等を行う.

配列・構造体・共用体を含むプログラムによってエラーを検出した場合, エラーの原因が配列・構造体・共用体によるものであるか特定する必要がある. そこで, 式中の配列・構造体・共用体をスカラ変数に置換する処理を行う. 配列の最小化の例を図 11 に示す. 5 行目の式の右辺に配列 `t0` が参照されているが, これらをスカラ変数に置換する. 構造体・共用体も同様の手法で最小化する.

```
switch ( 4 ) {       →   switch ( (t2*x4)!=(x1>>x10) ) {
  case 3:                  case (a0-x13)-(x21/t3):
    文リスト                   文リスト
    break;                     break;
  case 4:                  case (x3-a3)*(t6*x12):
    文リスト                   文リスト
    break;                     break;
  default:                 default:
    文リスト                   文リスト
    break;                     break;
}                        }
```

図 10: switch 文の生成例

```
1: int t0[3] = {0, 1, 2};
2: x0 = 5;
3: int main(void)
4: {
5:   t1 = t0[0] + x0 + t0[1 * t0[1]];
6:   if (t1 == 6) {OK();} else {NG();}
7: }
                    ↓
1: int t0_0 = 0; int t0_1 = 1;
2: x0 = 5;
3: int main(void)
4: {
5:   t1 = t0_0 + x0 + t0_1;
6:   if (t1 == 6) {OK();} else {NG();}
7: }
```

図 11: 配列の最小化の例

表 1: 実験結果

compiler	Csmith		Orange4	
	time[h]	#error	time[h]	#error
GCC-4.4.7	46.4	5	54.5	25
GCC-4.5.4	30.2	1	56.7	30
GCC-4.6.4	31.3	0	56.4	10
GCC-4.7.3	35.6	0	62.1	10

(Ubuntu 14.04 LTS, Xeon, 3.60GHz, RAM 16GB)

4. 実装と実験結果

提案手法に基づくランダムテストシステムを Orange4 に追加する形で Perl 5.20.2 で実装した. 本システムは Ubuntu Linux や Mac OSX 等の Unix 環境で動作する.

提案手法と Csmith で GCC のテストを行った結果を表 1 に示す. "#error" は検出したエラーの数である. それぞれのバージョンで 100,000 件のランダムテストを実行したところ, 全てのバージョンで提案手法の方がエラー検出率が高くなった. なお, テストプログラムの平均サイズは Csmith が 15.3 KB, Orange4 が 8.9 KB であった.

表 2 は提案手法が生成することのできる構文を, Csmith と比較したものである. 表中 "*" は本稿で追加した構文である.

最新版のコンパイラの実験を行ったところ, 2 件のエラーを新たに検出した. 検出したプログラムを最小化したものを図 12 に示す. いずれのプログラムも添字が式になっている配列を含んでおり, Csmith や従来の Orange4 では検出できないものである. 添字の式を数値に置き換えたり配列を数値に置き換えるとエラーは消失する. (a) では, 変数 `t` の期待値は 1 なので 10 行目の if 文の本体は実行されず, 11 行目の `return 0;` に到達するはずだが, -O3 オプションでコンパイル・実行すると, `t` の値は 0 になり, `__builtin abort();` が実行されてプログラムは異常終了する. (b) では, -O1 オプションでコンパイルすると linker

表 2: Csmith との比較

	Csmith	Orange4 (本手法)	備考
if 文	○	○	
for 文	○	○	Csmith のループ境界は定数のみだが, 本手法では一般の式, 本手法の繰り返し回数は 0 回 か 1 回
関数	○	○*	本手法では渡された引数の値が正しいか関数内で検査
while 文	×	○*	
switch 文	×	○*	
goto 文	○	×	
配列	○	◎*	本手法では配列添え字に複雑な式を指定したり, 可変配列も使用可
構造体	○	◎*	本手法ではメンバーに構造体の配列も使用可
ビットフィールド	○	×	
ポインタ	○	×	

```
 1: int a[2] = {0,1};
 2: int x = 129;
 3: int main (void)
 4: {
 5:   volatile int v = 0;
 6:   int t = x;
 7:   for (int i=0; i < (1+v+v+v+v+v+v+v+v+v+a[a[0]]); i++) {
 8:     t = a[(signed char)(130-x)];
 9:   }
10:   if (t != 1) __builtin_abort();
11:   return 0;
12: }
```

(a) GCC-8.0.0 で検出したエラープログラム

```
1: #define INT_MAX 0x7fffffff
2: char a[1][1] = { {1} };
3: int x = 0;
4: int y = -INT_MAX;
5: int main (void)
6: {
7:   if (a[x][INT_MAX+y] != 1) __builtin_abort();
8:   return 0;
9: }
```

(b) LLVM/Clang-6.0 で検出したエラープログラム

図 12: 最新版コンパイラで検出したエラープログラム

command failed となりコンパイルに失敗する．これらの不具合は Bugzilla を通して開発チームに報告した [*1] [*2].

表 3 は, GCC-4.8.5 に対して提案手法でテストを行った際に検出した 11 件のエラープログラムを Orange4 と C-Reduce で最小化するのに要した時間である．Orange4 は全てのケースで C-Reduce よりも速く最小化を完了した．これは, プログラムの意味情報を用いて, 未定義動作を引き起こさない縮約のみを適用しているためと考えられる．

5. むすび

本稿では, C コンパイラのランダムテストシステム Orange4 において, 制御文およびデータ型生成を強化する手法を提案した．Csmith と同等の構文を生成することができるようになり, 不具合検出能力も向上した．実験の結果, GCC-8.0.0, LLVM/Clang-6.0 において不具合を検出し, 開発チームに報告した．今後の課題としては, 引き続き最新版のコンパイラにおいてテストを続けエラーを検出することや, 表 2 にあるような未対応の C の構文への対応, C 言語以外の言語のテストへの対応が挙げられる．

なお, Orange4 は 2016 年 12 月 22 日 から GitHub で一般に公開している [*3] .

謝辞　本稿の研究にあたり, 御助言を頂きました関西学院大学石浦研究室の諸氏に感謝いたします．

本研究は一部 JSPS 科研費 25330073 の助成による．

表 3: 最小化の実行時間の計測

	CPU time[sec]	
	C-Reduce	Orange4
#01	328.2	6.7
#02	155.0	24.8
#03	990.6	15.7
#04	106.0	13.9
#05	254.3	31.3
#06	76.1	4.2
#07	1697.3	5.7
#08	3888.2	23.2
#09	1521.5	20.4
#10	106.8	4.5
#11	72.8	6.6

(Ubuntu 14.04 LTS, Xeon, 3.60GHz, RAM 16GB)

参考文献

[1] 石浦菜岐佐: “コンパイラのファジング,” 電子情報通信学会 Fundamentals Review, vol. 9, no. 3, pp. 188–196 (Jan. 2016).

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr: “Finding and Understanding Bugs in C Compilers,” in *Proc. PLDI 2011*, pp. 283–294 (June 2011).

[3] E. Nagai, A. Hashimoto, and N. Ishiura: “Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions,” *IPSJ Trans. SLDM*, vol. 7, pp. 91–100 (Aug. 2014).

[4] K. Nakamura and N. Ishiura: “Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation,” in *Proc. APCCAS 2016*, pp. 676–679 (Oct. 2016).

[5] 岩辻光功, 石浦菜岐佐: “等価変換に基づく C コンパイラテストシステムにおける制御文生成の強化,” 信学技報, *VLD2017-87* (Jan. 2018).

[6] C. Lindig: “Find a Compiler Bug in 5 Minutes,” in *Proc. ACM International Symposium on Automated Analysis-Driven Debugging*, pp. 3–12 (Sept. 2005).

[7] V. Le, C. Sun, and Z. Su: “Randomized Stress-Testing of Link-Time Optimizers,” in *Proc. ISSTA 2015*, pp. 327–337 (July 2015).

[8] V. Le, C. Sun, and Z. Su: “Finding Deep Compiler Bugs via Guided Stochastic Program Mutation,” in *Proc. ACM OOPSLA 2015*, pp. 386–399 (Oct. 2015).

[9] Gergo Barany: “Finding Compiler Bugs via Live Code Mutation,” in *Proc. OOPSLA 2016*, pp. 849–863 (Oct. 2016).

[10] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison and X. Yang: “Test-Case Reduction for C Compiler Bugs,” in *Proc. PLDI 2012*, pp. 335–346 (June 2012).

[*1] https://bugs.llvm.org/show_bug.cgi?id=35159
[*2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=83580
[*3] https://github.com/ishiura-compiler/Orange4

© 2018 Information Processing Society of Japan