算術式の解析木の導出に基づく C コンパイラのランダムテスト

RANDOM TESTING FOR C COMPILERS BASED ON DERIVATION OF SYNTAX TREES OF ARITHMETIC EXPRESSIONS

由良駿　中橋昌俊 *　石浦菜岐佐
Suguru Yura　Masatoshi Nakahashi　Nagisa ishiura

関西学院大学理工学部
School of Science and Technology, Kwansei Gakuin University

1　はじめに

コンパイラのランダムテストは, ランダムに生成したプログラムによりコンパイラの不具合を検出する手法である.

文献 [1] は, 期待値計算に基づく算術最適化の C コンパイラのランダムテストを提案している. しかし, この手法では算術式をランダムに生成した後に, 式中の変数の値をランダムに決定するため, 式中の演算のオペランドの値に偏りが生じるという課題が存在する.

そこで本稿では, 算術式とその部分式の期待値を並行して生成することにより, 式中の値を任意に決定できるテストプログラムの生成手法を提案する.

2　期待値計算に基づく C コンパイラのランダムテスト

2.1　算術式の導出

コンパイラのランダムテストには, (1) ランダムに生成したプログラムの正しい実行結果 (期待値) をいかに計算するか, (2) 未定義動作 (零除算や符号付整数演算のオーバフロー等) を含まないプログラムをいかに生成するか, という課題がある. 文献 [1] は, 期待値計算に基づき, C コンパイラの算術最適化のランダムテストを行う手法を提案している. この方法では, 図 1 のように算術式を含むプログラムを生成してコンパイラをテストするが, (1) まず式と変数の初期値をランダムに決定し, (2) 次に各式の期待値を計算し, (3) その過程で未定義動作を検出すると, これを回避するように式を修正する (例えば, 零除算を検出すれば, 除数にランダムな数を加算する演算を追加する). これによって多くの演算子を含む長い式を生成できるため, GCC-4.9.0 等の最適化不具合を検出している.

しかし, この方法では, 式中のオペランドの値に偏りが生じるという問題がある. これは, 比較演算の結果が 0 か 1 であることが原因である. これにより, ランダムな式中の部分式の値は, 絶対値が小さい値, あるいは絶対値が非常に大きい値 (絶対値が小さい符号なし整数同士の減算で, ラップラウンドが行われる結果生じる) に偏り, 演算の最適化を多様な値でテストすることができない.

3　解析木の導出に基づくテストプログラムの生成

本稿では, 算術式とその部分式の期待値を並行して生成することにより, 演算のオペランドの値の偏りをなくす手法を提案する.

```
01: #include <stdio.h>
02: #define OK()  printf("@OK@\n")
03: #define NG(fmt,val)  printf("@NG@ (test = " fmt ")\n",val)
04:
05: signed char x0 = 11;
06: const volatile unsigned char x2 = 6U;
07: signed int t0 = 44490973;
08: static unsigned long t1 = 851211932LU;
09: signed int t2 = 5218232;
10: signed long long t3 = -1524120658162LL;
11: unsigned long t4 = 45304606LU;
12: signed int t5 = 97;
13: signed int t6 = 1;
14:
15: int main (void)
16: {
17:  const volatile unsigned short x1 = 3863U;
18:  const signed char x3 = -2;
19:  const volatile signed long long x4 = -200036528799435LL;
20:  signed int x5 = 0;
21:  unsigned short x11 = 2U;
22:  const volatile unsigned long x12 = 31LU;
23:  const volatile signed int x15 = 31;
24:  signed int k16 = 1397218534;
25:
26:  t0 = (x2>>x12);
27:  t1 = (x1&x12);
28:  t2 = ((x3+k16)>>x15);
29:  t3 = (x4&x11);
30:  t4 = (x12+x0);
31:  t5 = (x2!=x2);
32:  t6 = (x5*x1);
33:
34:  if (t0 == 0) { OK(); } else { NG("%d", t0); }
35:  if (t1 == 23LU) { OK(); } else { NG("%lu", t1); }
36:  if (t2 == 0) { OK(); } else { NG("%d", t2); }
37:  if (t3 == 0LL) { OK(); } else { NG("%lld", t3); }
38:  if (t4 == 42LU) { OK(); } else { NG("%lu", t4); }
39:  if (t5 == 0) { OK(); } else { NG("%d", t5); }
40:  if (t6 == 0) { OK(); } else { NG("%d", t6); }
41:
42:     return 0;
43: }
```

図 1　文献 [1] のテストプログラムの例

本手法では, ランダムに生成された定数に対し, これが演算の結果の値となるように式を決定するという操作を繰り返すことにより算術式を生成する. 式生成の例を図 2 に示す. まず, 式の計算結果 (期待値) の型と値 (この例では int 型の 12) をランダムに決定する. 次に, 演算子 (−) と左オペランド (21) をランダムに決定する. 左オペランドの値は, この演算で結果を 12 にすることが可能な範囲の数から選択する. この後, 右オペランドの値を計算により求める ($21 - 12 = 9$). 定数は, その値を持つ変数に置き換えても良いし, さらに $9 \to 3 \times 3$ のように導出を繰り返しても良い.

*現在 新明和テクノロジ株式会社

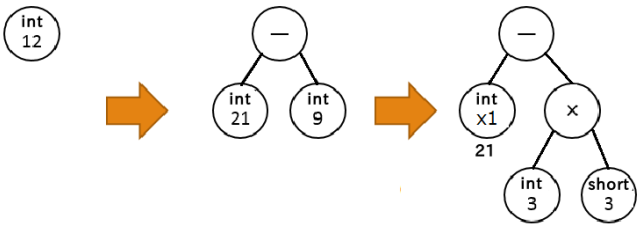

図 2　算術式の導出

演算子に乗算を選択した場合のオペランドは, 結果の値の約数からランダムに選択する. ただし, 素因数分解を行うと時間がかかるので, 定数個 (25 個程度) の素数でのみ除算を試みるようにする.

演算のオペランドの型はランダムに決定するので, 例えば int 型の演算のオペランドに int 型以外の型を指定することもできる. ただし, int 型よりも上位の型を指定する場合には, キャスト演算を挿入する.

定数の値を単純に一様乱数で決定すると, 値が 0 や 1 になる確率が低くなるため, 演算子に比較演算を選択できる機会が低くなってしまう. また, 演算子に左シフト演算を選択した場合, 結果の値が 2^k になる確率は k が大きくなるにつれて小さくなるので, 右オペランドに大きな値を選択できる機会も低くなってしまう. この問題を解決するため, 定数の値をランダムに決定する前に, 次の導出で用いる演算子を先に決定し, この演算子に適した値をランダムに生成するようにする.

定数を変数に置き換える場合には, 事前に定めた確率で既存の変数と新しい変数を選択する. ただし, その値を持つ既存の変数がいつも存在するとは限らないので, 定数の値をランダムに選択する際に, 既存の変数の値の中から可能な範囲に入っているものを選択するようにする.

4　実験結果

本稿で提案するランダムテスト手法を文献 [1] のテストシステム Orange3 に組み込んだ. このテストシステムは Perl で実装しており, Linux, Mac OSX, Windows 上の Cygwin 等の上で動作する.

表 1 は, 文献 [1] の手法と提案手法について, 演算子数 1000 のテストプログラムをそれぞれ 100 本生成し, プログラム中に現れる算術式の期待値の出現頻度を比較したものである. 文献 [1] の手法では期待値が 0 と 1 に大きく偏っていたが, 提案手法ではこれを修正できている.

表 1　値の出現頻度の比較

期待値	0	1	2〜8	9〜64	65〜max
文献 [1]	37.8%	12.4%	4.0%	6.6%	38.4%
本手法	5.0%	7.6%	11.4%	15.5%	60.2%

表 2 は, いくつかのコンパイラについて, 文献 [1] の手法と提案手法によるランダムテストを 24 時間実行した結果である. CPU は Intel Core i7 3540M 3GHz (RAM 12GB), OS は Ubuntu 14.04 であり, テストプログラム中の演算子数は 1000 に設定した. 「テスト数」はテストプログラムの数, 「エラー数」はエラーを検出したプログラムの数である. 提案手法のテスト数が少ないのは, [1] よりもテストプログラムの生成に時間を要するためである. 2 つの手法で検出エラー数に変化が見られることから, プログラム中に出現する値の分布がエラーの検出に影響を与えていると考えられる. GCC-4.5.0 および Clang-3.3 では, 提案手法はより多くのエラーを検出できている.

表 2　検出能力の比較

コンパイラ	文献 [1]		本手法	
	テスト数	エラー数	テスト数	エラー数
GCC-4.5.0	57,786	155	27,005	168
GCC-4.7.4	52,977	10	24,799	2
Clang-2.8	57,810	216	27,969	4
Clang-3.3	62,455	306	28,260	970

5　まとめ

本稿では, 算術式の解析木の導出に基づく C コンパイラのランダムテスト手法を提案した. テストプログラム中の演算のオペランドの値の分布とコンパイラの不具合検出能力の関係を詳細に調べることが今後の課題として挙げられる.

謝辞

本研究を行なうにあたり, 多くの御助言や御協力を頂きました関西学院大学理工学部 石浦研究室の諸氏に感謝いたします. 本研究は一部科学研究費補助金 (25330073) による.

参考文献

[1] 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. vol. 7, pp. 91–100 (Aug. 2014).