コンパイラのテスト結果から
アプリケーションコードの品質を改善する

概要

SuperTest は、多くのコンパイラサプライヤに使用される、C および C ++用のコンパイラのテストと検証スイートです。その主たる目的はアプリケーションコードがコンパイラによって正しく変換されるということを確かめることですが、アプリケーションの品質を向上させることにも、ある種の効果をもたらし得ることについて、この資料で紹介します。

コンパイラの品質について

コンパイラは非常に複雑なプログラムで、多くのオプションがあります。そしてコンパイラに起因するエラーは、思いも寄らないところでアプリケーションを破壊することがあり、デバッグも困難です。

もちろんコンパイラはサプライヤによりテストされますが、ユーザー側の特定のユースケース（環境オプションとコンパイラオプションの組み合わせ）でテストされることは殆どありません。そこでアプリケーション開発者には、コンパイラが特定のユースケース下でアプリケーションエラーを引き起こさないことへの確証が必要になります。

SuperTest のテスト

SuperTest のテストにはポジティブとネガティブの二種類があります。

ネガティブテストはプログラムに言語の構文や意味規則に反する誤りがあってコンパイラが診断メッセージを出すべきもので、コンパイルエラーが出されると予想されます。SuperTest はコンパイルエラーが出るとテスト成功と報告します。また、コンパイラ自身のクラッシュもできるだけ検出します。

ポジティブテストでは、エラーなくコンパイルされたコードの実行時の振る舞いが言語規格に則って正しいかどうかを検査します。例えば、配列サイズが初期化子で決定されることの動作をテストするために、int[]{1, 2, 3, 4} のサイズが sizeof(int)の 4 倍に一致するかの検査コードを実行します。

また最適化でのコード変換が正確であるかどうかを検証するテストもあります。

ネガティブテストの例

```
/*
 * (c) Copyright 2017 by Solid Sands B.V.,
 *     Amsterdam, the Netherlands. All rights reserved.
 *     Subject to conditions in the RESTRICTIONS file.
 */

#include "def.h"

#ifdef CVAL_STDC

typedef struct s {
    int *p;
    int a[3];
} ts;

ts observed;

MAIN
{
    ts  a, b;

    CVAL_HEADER("Cast operators: void type or scalar to scalar");

    a.a[0] = 3;
    a.a[1] = 2;
    a.a[2] = 1;
    a.p    = &(a.a[2]);


    b = (ts) a;
    observed = b;

    /*    ( type-name ) cast-expr
     *
     *    Constraints
     *
     *    [#2] Unless the type name specifies a void  type,  the  type
     *    name  shall specify qualified or unqualified scalar type and
     *    the operand shall have scalar type.
     */
```

このコードは、C 言語では構造体型（構造体型へのポインタではなく）へのキャストはできなくて、コンパイラが診断メッセージを出すべきであることをテストするものです。

例えば GCC 7 と clang では、このテストに失敗して診断メッセージは出力されません。

使用を検討するコンパイラがこのテストに失敗する場合、静的コード解析ツールを利用するなどしてキャストに関するアプリケーションのコーディング検査を十分に実施し、このようなコードがコンパイルされることを未然に防ぐといった対応が考えられます。

ポジティブテスト/実行時エラーの例（1）

```
/*
 * (c) Copyright 2015 by Solid Sands B.V.,
 *     Amsterdam, the Netherlands. All rights reserved.
 *     Subject to conditions in the RESTRICTIONS file.
 */

#include "def.h"

typedef struct {
    int  k;
    int  l;
    int  a[2];
} T;

typedef struct {
    int  i;
    T    t;
} S;

T x = {.l = 43, .k = 42, .a[1] = 19, .a[0] = 18 };

S y;

void
f(void)
{
    S    l = { 1, .t = x, .t.l = 41, .t.a[1] = 17};

    y = l;
}

MAIN
{
    CVAL_HEADER("Partial override");

    f();

    CVAL_VERIFY(y.i == 1);
    CVAL_VERIFY(y.t.k == 42);
```

出力ログ

```
----------------------------------------
C99/6/7/8/t7.c:

LOG:
        Partial override
        line 42: failed at:
        y.t.k == 42
        line 44: failed at:
        y.t.a[0] == 18
RESULT: C99/6/7/8/t7.c                                  FAILED
```

このテストは、構造体への部分的な上書きが正しく行われているかどうかをみるものです。出力ログに示されるように、このコンパイラ（x86-64 用の GCC 7）では正しく実装されていません。

ポジティブテスト/実行時エラーの例（2）

```
/*
 * (c) Copyright 2015 by Solid Sands B.V.,
 *     Amsterdam, the Netherlands. All rights reserved.
 *     Subject to conditions in the RESTRICTIONS file.
 */

/*
 *     test the rint() function.
 */

#include    "def.h"
#include    <math.h>
#include    <fenv.h>


#pragma STDC FENV_ACCESS ON

MAIN
{
    int  i;
    double    d;
    double    res;
    double    diff;

    CVAL_HEADER("Test the rint function (rounding direction)");

#ifdef FE_DOWNWARD
    CVAL_SECTION("Downward");
    if (fesetround(FE_DOWNWARD) == 0) {
        d = -.00147;
        for (i = 0; i < 40; i++) {
            CVAL_CASE(i);
            res = rint(d);
            CVAL_VERIFY(rint(res) == res);
            diff = d - res;
            CVAL_VERIFY(diff >= 0.0);
            CVAL_VERIFY(diff < 1.0);
            d = d * -1.532;
        }
    } else {
        CVAL_NOTEST();
    }
#endif
```

出力ログ

```
----------------------------------------
C99/7/12/9/4/t10.c:

LOG:
        Test the rint function (rounding direction)
        line 40 case 0: failed at:
        diff >= 0.0
        line 40 case 2: failed at:
        diff >= 0.0
        line 40 case 4: failed at:
        diff >= 0.0
```

このテストは丸めの設定に伴うもので、コンパイラ（x86-64 用の GCC 7）の最適化-O2 でエラーとなります。（-O0 では正しく実行されます。また、LLVM では-O2 でも正しく実行されます。）

これらのテスト結果は直接的にはコンパイラの不具合の指摘ですが、コンパイラのユーザーであるアプリ開発者から見ると、そのコンパイラの弱点（欠陥）を認識して、自身のソースコードでそれを回避するような対応をとることができます。

例えば、ポジティブテスト/実行時エラーの例(１)によれば、構造体への部分的な代入を行っているテストプログラムの内容と「Partial override」というテスト内のキーワードから、自身のアプリケーションコード内で、構造体への部分的な代入があるかどうかを主として構造体に対する操作を見直し、必要な修正を行うことが考えられます。これには静的コード解析ツールを利用することもできます。

また、同じく例(２)によれば、整数値への丸め演算を行っているコードに対して演算の仕様や処理方法の確認による見直しを行う、不具合が発生したコンパイラ最適化レベルを使用しない、または、自身のコードと最適化レベルの組み合わせで動作確認を行う、などの対応策を実施することが考えられます。

まとめ

SuperTest はアプリケーションコードがコンパイラによって正しく変換されるということを確かめることが主たる目的ですが、上記のようにしてアプリケーションの品質を向上させることにも、以下のようにある種の効果をもたらし得ます。

・ネガティブテストで必要な診断が得られていないと分かった場合、ユーザープログラムでその誤ったプログラム要素を使用しているかどうかを検出し避けることで、そのアプリケーションの移植性を向上させることにつながります。

・SuperTest にはオブジェクトコードや実行ファイルのサイズをレポートする機能があり、これを用いれば、非効率な要素を発見できる可能性があります。また、コンパイルオプション指定とサイズ報告の組み合わせのテスト結果を調査して、ユーザープログラムに対してより適したオプションを選んだり、複数のコンパイラやオプションによる性能比較に利用したりすることもできます。(実行時間に関しては、デバッガやシミュレータとの連携が必要になるかもしれません。)

・生成されたコードの実行時エラーを検出した場合、そのエラーに対するプログラム要素を確認し（ワークアラウンドとして）避けることができます。

・実行時のエラーが最適化レベルやその他オプションの設定で検出される場合、それらの組み合わせの結果を調査することで、ユーザープログラムに対してより適したオプションを選ぶようにもできます。

また、ユーザーのアプリケーションコードの向上の観点以外にも以下のような SuperTest の利用が考えられます。

・レガシーでリファクタリング困難なサイズの大きいコードが新しいコンパイラでうまくコンパイルできるかというような点に関して、SuperTest には 1 関数で何千行にもなるような関数など限界テストも多数もっており、あらかじめ限界値に対する検討も可能になります。

・新しいバージョンのコンパイラやライブラリを入手するたびに、コンパイルされたコードが旧バージョンと一致することをチェックする必要があります。アセンブラレベルでのコード比較を人手で行うには膨大な労力と時間を要しますが、バージョンやライブラリの更新ごとに SuperTest を実施し同じ結果を得ることで、それらの振る舞いが同じである確証が得られます。これはターゲット CPU が変更される場合にも同様です。

SuperTest について、以下の資料も参考いただけるかと思います。

・SuperTest の概要を説明する Q&A

https://www.fuji-setsu.co.jp/files/SuperTestFAQs.pdf

・ポジティブテストによるコンパイラバックエンド（ジェネレータ）のテストについて

https://www.fuji-setsu.co.jp/files/JP_Code_generator_validation.pdf

・C、C++コンパイラの tool qualification の効用

https://www.fuji-setsu.co.jp/files/QualificationBenefitsSuperTestf_JP.pdf

FUJI SETSUBI

富士設備工業株式会社 電子機器事業部　www.fuji-setsu.co.jp