Programming Language and Compilers Project

Deadline: Jue. 18, 2017

1 Project: Compiler Testing

Compilers are widely used software in today's world, the correctness of the compilers is very important for us. So in last years, people mainly proposed three compiler testing methods: Randomized Differential Testing (RDT) [1,2,3,4], a variant of RDT—Different Optimization Levels (DOL) [2,4], and Equivalence Modulo Inputs (EMI) [5]. So far, people have found a lot of compiler crash and compiler miscompilation problems in GCC,LLVM and other compilers.

In this optional project, you will design and implement a compiler test tool by leveraging the EMI method for cool programming language. This tool might be used to test cool compilers developed by other students during this course. Indeed, the mandatory project of this course is to design and develop a frond-end for cool language including a lexical analyzer, a syntax analyzer, a sematic analyzer and an immediate code generator which transforms cool programs into LLVM IR. Using LLVM back-end, we could compile cool programs into any target programs.

For the three compiler testing methods we have mentioned above, here we will give a simple description about them [4]. For RDT, we firstly need to assume that several comparable compilers are implemented based on the same specification, then for a given set of testing compilers, denoted as$\{C_1, C_2, ..., C_n\}$,where $n \geq 3$, the process for applying RDT to test these compilers is as follows. Each compiler C_i compiles a test program P and generates an executable E_i, where $1 \leq i \leq n$. For any given set of test inputs for P, denoted as I, these executables produce different results, denoted as $O_1, O_2, ..., O_n$ Because the compilers under test are designed to follow the same specification, their behaviors are expected to be the same. Therefore, RDT detects compiler bugs through the voting among $O_1, O_2, ..., O_n$. For DOL, We consider that a test program is respectively compiled under different optimization levels (e.g., -O0, -O1, -Os, -O2 and -O3 in GCC) and executed under the same set of test inputs, it may produce different results, thus we know that the compiler under test contains bugs. Through the introduction about RDT and DOL, we know that RDT need several compilers to compare, and because all of the compared compilers are under the same specification, they may produce the same wrong results that we can't find. DOL need to test different optimization levels, but for the cool language compilers, we don't design the optimization by ourselves, so it's infesible to use the RDT

and DOL methods. In this project, we mainly want you to try the EMI compiler testing method, the EMI compiler testing method is proposed by Zhendong Su(http://web.cs.ucdavis.edu/~su/), who is a professor in Department of Computer Science ,University of California, Davis.

EMI is a newly proposed compiler testing technique, which addresses the test oracle problem through comparison between a test program and its variants whose behaviors are regarded as equivalent under a set of test inputs for this test program. In particular, for a test program, EMI identifies a set of statements that affect its behavior given some test inputs, and constructs variants whose behaviors are equivalent to the behavior of the test program. EMI introduces a "Profile and Mutate" strategy on the statements of the given test program and uses this strategy to generate variants.

For a compiler under test (denoted as C), the process for applying EMI to test this compiler is as follows. For any given test program P and its test inputs I, EMI first generates some variants of P (denoted as $Q_1, Q_2, ..., Q_m$) through the following process. First, EMI identifies a set of statements in P unexecuted by I through dynamic analysis, and generates variants by randomly deleting statements within this set. Then the test program P and each variant Q_i are compiled by the compiler C, and thus the corresponding executables (denoted as E_p and E_i) are generated. Taking I as test inputs, these executables produce results, denoted as O_p and O_i . Because the variants should behave equivalently to given program P under test inputs I, EMI detects the bugs in C by comparing each pair O_p and O_i .

Now I will give an illustrative example for EMI [5], the Figure 1 shows two c language programs. From the program P_a, we know that none of abort calls in the function f should execute when the program P_a runs, and the coverage data confirms this. This allows us freely alter the body of the *if* statements in the function f or remove them entirely without changing this program's behavior. By doing so, we can transform the program P_a and produces many new test cases, one is like P_b we show in Figure 1. In fact, the program P_b is found to be miscompiled by Clang through Su's team in [5]. So, we know that when we compiler the two EMI programs and run at the same input, if we find a different output, we should know that there must be a miscompilation in the compiler.

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) abort();
  if (x.d != 20) abort();
  if (x.e != 30) abort();
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) abort();
  if (z.c != 12) abort();
  if (z.d != 22) abort();
  if (z.e != 32) abort();
  if (l != 123) abort();
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

(a)

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) /* deleted */;
  if (x.d != 20) abort();
  if (x.e != 30) /* deleted */;
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) /* deleted */;
  if (z.c != 12) abort();
  if (z.d != 22) /* deleted */;
  if (z.e != 32) abort();
  if (l != 123) /* deleted */;
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

(b)

Figure 1: Example for EMI, (a) is program P_a, (b) is program P_b

2 Guidelines

• The detailed EMI testing method you can find in Su's homepage, and we suggest you to read the following article:

-V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2014.

• For the data structure of Cool, you can use the code from the Cool Support Source Code we have listed in the class's homepage.

• If you have any problem about it, you can connect me through the email: zhangjun@shanghaitech.edu.cn

• No matter how much you have worked with this project, we will consider about the work you have done and give you a bonus grade. If you can solve this whole compiler testing project and find some bugs, you can get A or A+ in the final grade.

• If you have any other idea about this project, or you can discuss with me.

3 Delivery items

• A report that clearly describe the background, your design thoughts, the specific realization and the testing results.

- Source code and a user's guide.
- You need to prepare a final presentation to show what you have done.
- At last, all of these materials must be sent to zhangjun@shanghaitech.edu.cn

4 References

[1] W. M. McKeeman. Differential testing for software. Digital Technical Journal, 10(1):100–107, Dec. 1998.

[2]F. Sheridan. Practical testing of a C99 compiler using output comparison. Software: Practice and Experience, 37(14):14751488, 2007.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In Proceedings of the 32nd Conference on Programming Language Design and Implementation, pages 283294, 2011.

[4]J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, B. Xie, "An empirical comparison of compiler testing techniques", Proceedings of the 38th International Conference on Software Engineering. ACM, 2016.

[5]V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In Proceedings of the 35th Conference on Programming Language Design and Implementation, pages 216226. ACM, 2014.