VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Debugging 42

Aden Kenny

Supervisor: Dr. Marco Servetto

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

42 is a statically typed, high level programming language which allows the composition and cooperation of millions of libraries at once. The 42 compiler, like the vast majority of software products, is not bug-free and therefore requires significant testing. This project has produced tests that have located bugs in the compiler and has lowered the number of unknown bugs in the 42 compiler.

Acknowledgments

Contents

Figures

Chapter 1

Introduction

> *"The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software."*
>
> – Edsger W. Dijkstra

42 is a statically typed, high-level programming language designed to allow the use and composition of millions libraries simultaneously and [1]. The compiler is implemented in Java and is relatively untested. This project's aim is to design and implement a testing regime in order to identify bugs in the current implementation of the 42 compiler. Generally compiler bugs are often found through language usage but this does not mean that highly popular languages are free from them [2] [3] [4] [5]. This shows that we cannot guarantee bug free code in even the most used languages.

Compiler testing is generally considered difficult and time consuming, and testing by itself is unlikely to be adequate to guarantee correctness, [6] let alone being free of harmful bugs. This means that the best we can do without resorting to formal methods is simply finding as many bugs as possible with the resources available, which is what this project attempts to do. Therefore the overall goal is to create as many short 42 programs as possible and attempt to compile them. The programs that cause the compiler to throw an error or simply not behave as expected will be examined for the possibility of a bug. If a bug is found, a bug report will be created and sent to Dr Servetto where it will either be fixed or added to a to-do list of bugs to fix. Therefore this project has the singular goal of finding as many bugs as possible but it must be noted that goal of this project is not prove the absence of bugs in the 42 compiler, as testing cannot show or prove the absence of bugs [7].

Useful background reading is introduced in this report. Firstly a brief introduction to the most important parts of 42 is presented. We then review an example of a fully verified compiler. A verified compiler would be impractical for this project, but it does present the ultimate solution in terms of producing a bug free compiler and therefore provides a useful starting point for considering compiler correctness and testing. The concept of fuzzing is introduced and the potential usage of fuzzing to find bugs in programs is also discussed, and we discuss the ability of using fuzzing tools as part of this project. Finally we review a small sample of the current literature regarding compiler testing, and present possible usage of the techniques introduced and how they could be used as a part of this project.

2

Background and Related Work

2.1 An introduction to 42

42 has a set of guarantees that are key to the language and and its use in practically any situation. These guarantees are also useful, as a user language knows that they will always hold true, except in the case of bug interfering with the language guarantees. The following section will quickly introduce the most important guarantees provided by 42.

2.1.1 Language guarantees

Immutability

In 42 an immutable object, as the name suggests, is always immutable and will stay immutable for the rest of its life cycle [1]. A reference to an object is immutable by default which gives us some guarantees surrounding objects and makes it easier to reason about them when contrasted to objects that are mutable, or could become mutable in the future [8].

```
A: {
   S out = S"Hello, World!"
   out := S"Hello, 42!" // This is not allowed as out is immutable.

   var S out2 = S"Hello, World!"
   out2 := S"Hello, 42!" // This is allowed, out2 is mutable.
}
```

Encapsulation

For an encapsulated, mutable, object, a single ”capsule binding” to that object is the one and only means of accessing the encapsulated object and by extension its graph of mutable objects [1]. Immutable objects do not require this guarantee for obvious reasons.

Error safety

In 42 we have try blocks with catch clauses just like Java and C++. The main difference is that in 42 the any code that is ”guarded by a catch error” [1] (in a catch clause) cannot modify an external state. This allows much easier reasoning about code inside a catch clause, especially when combined with immutable references to objects.

Subtyping

Java code
In Java it is always possible to create a subtype of a class assuming the class is not declared final.

```
class A {
   A() {
      A a = B.b(new B()); // Won't work in 42. An exact type was not passed.
      B c = B.b(new A()); // Won't work either. Return type is not exact.
      A b = B.b(new A()); // This will work.
   }
}
class B extends A {
   static A b(A a) {return a;}
}
```

42 code
In 42 we cannot create a subtype of a class through extending it, and subtyping is significantly more restrictive [1]. The only way to get a subtype is through an interface, as all classes must be of an exact type therefore subtyping can only be accomplished through inheriting an interface. This means that we know that any class reference is of an exact type with one important exception to this is through interfaces where a reference to an interface type could be a type of any implementing class.

```
A : {interface
   {...}
}
B: {implements A
   class method // class keyword is equivalent to static.
   B b(B that) {return that}
}
C: {implements A
   class method
      A c(B that) { // We're returning an A.
         B known = that.b(that) // This must be of exactly type B.
         return known // This is okay as A is an interface, not a class.
      }
   }
}
D: {extends B // This won't work. We can't extend classes.
   {...}
}
```

2.2 A note on compiler correctness

A compiler is an extremely large and complex piece of software; it is highly likely that bugs exist in the implementation of practically any language [9] and 42 is no exception to this.

Complier testing can be a difficult and time consuming process and Leroy [10] states that compiler-introduced bugs are widely considered to be extremely difficult to find and track down. Below we will explore some of the current approaches to compiler testing and debugging that we consider most important to this project, and explore how these ideas introduced in these approaches can best be applied to this project.

2.3 CompCert

2.3.1 Introduction

The only way to truly prove a piece of software is bug free is through formal verification and the best way to make a systematic statement about the correctness of the code is to use formal methods [11] Leroy [10] presents a compiler called CompCert for a large subset of the C language [12] [13]. CompCert is a verified compiler that provides some semantic preservation and has "a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program" [14].In theory this rules out the possibility of bugs being introduced by the compiler and this means that any bug free code written will produce a bug free program.

2.3.2 Conclusions

The CompCert compiler is an interesting project and contains useful information for this project, but to carry out similar project for 42 language is unrealistic for multiple reasons. First and foremost is that the writing of a formally verified compiler is significantly out of scope of this project, especially considering the CompCert compiler has had over 6 man years of effort spent on it. Additionally if the approach used by Leroy was to be applied to a language different than the C subset introduced by Blazy [13] some issues may arise. The CompCert compiler, as of publication of Leroy's paper [10], only produces code for PowerPC, a RISC ISA. This is in direct contrast with 42 which targets the x86 ISA, and targeting x86 may be more difficult due to it being a CISC ISA. Finally, the 42 language could prove to be more difficult to produce a verified compiler for due to difficult to verify features such as metaprogramming and large numbers of libraries in a 42 project.

2.4 Fuzzing

2.4.1 Introduction

Fuzzing is the technique of supplying random input into a program or application. This technique is quite often considered to be quite simple [15] but can be effective as it is fairly easy to implement and can find a large number of bugs[11] [15] [16] [9]. An example of a modern, well used fuzzing tool is *american fuzzy lop* (AFL) [17]. AFL differs from other older and less advanced fuzzing tools such as *fuzz* [11] by the use of compile-time instrumentation and genetic algorithms. This allows AFL to "discover clean, interesting test cases that trigger new internal states in the targeted binary" [17]. AFL has been used to find bugs in major programs such as *Apple Safari*, *llvm*, *Adobe Reader*, and *VLC* [17].

2.4.2 Potential fuzzer usage

A potential issue with using AFL is that the interface to run it on Java programs [18] is less used and less well documented than the standard AFL version. While fuzzers are a useful tool that have worked well on similar projects such as *llvm*, using a fuzzer on the 42 compiler may present some issues. The most pressing issue would perhaps be compilation time of a 42 program using the 42 compiler. Compiling a 42 program currently takes a significant period of time (around 45 seconds on fairly standard consumer hardware) and if compiling large numbers of programs, this compilation time would quickly add up to a large period of time. If an attempt was made to fuzz the compiler by providing random 42 programs it would require writing a fuzzer from scratch which is almost certainly beyond the scope of this project, as well as still having the previously mentioned disadvantage of significant compilation times. These issues do not preclude the use of a fuzzer but a more careful approach would have to be taken perhaps with the use of a different fuzzer [19] [20] focusing more on fuzzing distinct parts of the compiler rather than fuzzing the entirety of the compiler.

2.5 Literature on compiler testing

In a paper presented by Chen et al. the topic of automated compiler testing[21] is introduced. They note that traditional automated testing techniques such as fuzzing have performance issues, with many tests and a significant amount of time needed to find bugs. They then introduce the idea of *learning to test* and propose LET, an approach to compiler testing that allows the prioritisation of test programs through a learning process, where a model is trained. This model is then used to prioritise test programs based on their likelihood of finding a bug and their performance costs. The results are quite impressive with around 60% of test cases being sped up by at least 25%. While the results are significant, the usage of the findings in regards to this project seems unlikely. This is primarily due to the approach being cutting edge, and seemingly difficult to replicate or even use if it were to be available.

In another paper Chen et al. present a comparison of compiler testing techniques [22]. They compare three different compiler testing techniques, Randomized Differential Testing (RDT), Different Optimization Levels (DOL) (a variation of RDT), and Equivalence Modulo Inputs (EMI). The results show that DOL is effective at finding bugs relating to optimisation, and RDT is generally more effective at detecting other types of bugs, but all three techniques can be used together to complement each other to a certain extent. The results presented here are not particularly useful to this project. Firstly regarding DOL, since 42 does not have a heavily optimising compiler or different levels of optimisation, DOL would not be particularly useful if applied to this project. RDT is used with two different compilers implementing the same specification and 42 only has a single complier implementing the specification therefore RDT is not useful to us. EMI on the other hand could be useful to this project. EMI is fairly similar to LET presented in another paper by Chen et al. [21] therefore the viability of using EMI and LET should be explored for this project.

3

Work Done

3.1 Initial steps

The initial stages of the project consisted of familiarising myself with the 42 language and as I had no previous knowledge of the language this would prove to be extremely important. I started by reading the 42 tutorial [1]. After this I started to get some hands on experience by writing very basic and small 42 programs. Writing these programs uncovered a few small bugs but the primary value of them was learning how to use the language which would allow me to create more complex programs and push the boundaries of the language later on.

3.1.1 Bug areas

Once I had familiarised myself with the language, Dr Servetto suggested a few areas of the language where he thought it likely that it would be possible to find bugs.

- Parser
- Type system
- Metaprogramming
- Reduction
- Native interfacing

3.1.2 Bug types

Dr Servetto also laid out the criteria for what constituted a bug and these criteria are fairly simple and straightforward;

- A Java error is produced
- 42 behaviour does not match specifications

Java errors

The first criterion is fairly self-explanatory. If stderr reports a Java exception or error we classify it as a 42 bug as all errors or exceptions should be handled by the 42 compiler. This not true in all cases, and in sometimes a Java error being produced reveals a deeper problem. An example of this is in Figure 3.1. We would expect the code in Figure 3.1 to

throw some sort of type error as *s* is an immutable object as it is not prefaced with the *var* keyword. However instead of this expected behaviour, this code snippet will throw a Java AssertionError. If we look at the Java stack trace we can see that an assertion is violated in the type system. The purpose of this assertion is to make sure that a state that we should never reach (when behaviour is as expected) is not reached. Therefore we can come to the conclusion that since this assertion is violated we have reached a state that should not be reached. We can then come to the conclusion that we have an error that is not just 42 failing to handle its own error, it also shows there is an implementation error in the compiler.

```
C: {
   s = 12Num // s is immutable as variables are immutable by default.
   s := 13Num // Note Pascal style for updating variables.
   return ExitCode.normal()
}
```

Figure 3.1: The update immutable bug. This bug throws a Java AssertionError and is on the TODO list to be fixed.

Behaviour mismatch

This criterion is significantly more broad than the previously mentioned one. This includes 42 behaviour that does not match with the specifications laid out for 42 (Figure 3.2 is an example of this) [1], and behaviour that is not mentioned in the specifications (Figure 3.3 is an example of this).

```
C: {
   s = return ExitCode.normal()
}
```

Figure 3.2: Assigning an exit code to a variable. According to the specifications this should work correctly

```
C: {
   Num n = 1Num / 0Num // n has the value of 1/0.
   Debug(n) // This will print 1/0 to the console.
} // No error is thrown by this code.
```

Figure 3.3: 42 division by zero. This should not be allowed and we would expect it throw some sort of math error.

The behaviour for the snippet of code in figure 3.3 is not specifically defined in the 42 specifications [23]. Therefore we must err to our best judgment on a case by case basis. In this specific case it is clear that this is not how the code should behave, it is nonsensical to allow division by zero therefore this should result in both the specifications and implementation being changed.

```
class A {
   static void a() {
      int n = 1 / 0;
      System.out.println(n);
   }
}
```

Figure 3.4: Java division by zero. The equivalent code of figure 3.3. This will throw an ArithmeticException which is the behaviour we would expect in 42.

3.2 Testing

For this project two main types of tests were written. Firstly, various general tests that did not target any specific area of the compiler were written. This kind of test would generally be implementing simple, well known program in 42. Examples of this include finding the maximum value in an array or finding if an array contains a value (see Appendix A). These tests found roughly half the bugs (6) but made up around 70% of the total tests written (196). This means that compared to targeted tests which found 8 bugs, the more general tests have a lower chance of exposing a bug. Though it must be noted that this is skewed by the fact that many of the general tests were written when initially familiarising myself with the language.

After a potential bug was found I would attempt to minimise the code required to replicate bug then a bug report (an example of a bug report can been seen in Appendix B) would be sent to Dr Servetto who would give feedback on the bug, such as if it was already know, the likely cause, and how difficult it would be to fix. If a bug was of high priority Dr Servetto would fix it and the bug report could be closed. If it was not of a high enough priority it would be put in the "TODO list". This meant the bug would then be known and could be fixed in the future.

3.3 Conclusion

A lesson that has been learnt over the course of this project is that compiler testing is very different when contrasted to more normal testing. Compiler bugs are considered to be difficult to find [10], and attempting to find bugs in the 42 compiler was no exception. Some bugs were found by programming fairly simple programs, but the majority were found through pushing the boundaries of what one might expect a programmer to do in the normal usage of a language. Examples of this include the "underscore bug". In 42 an underscore character is a valid variable name, and if an object that was passed to a method as a parameter was called "_" and "_" was not in scope when the method was called, a Java NullPointerException would be thrown (see Figure 3.5 for full code snippet). This is clearly unintentional behaviour and a bug in the 42 compiler.

This project has shown, that while writing simple programs may find some fairly simple bugs, a far larger quantity of bugs will be found by specifically targeting different areas of the compiler or by writing unusual code. An additional reason to focus on these two methods is that many (but most certainly not all) of the bugs that could be found by writing simple or standard programs have already been found and fixed.

An advantage of working with the 42 compiler versus a C or C++ compiler is that a significant number of compiler bugs can be attributed to aggressive compiler optimisation

```
A: {
   class method Any m(Any that) that // Any is supertype of all objects,
      like Object in Java.
}

B: {
   a = a.m(_) // Underscore is a valid variable name but there is no
      variable with this name in scope.
   return ExitCode.normal() // We need to declare an exit status on any
      program.
}
```

Figure 3.5: The underscore bug. This code would throw a NullPointerException in Java and it was subsequently fixed to produce the correct 42 error.

[3][24][25][26]. At the current time optimisation is not a priority for the 42 compiler at the present time. A major reason for this is that some language features are still being implemented and numerous bugs are still being found, as can be seen by the results of this project. The lack of aggressive compiler optimisation meant that some fairly common classes of compiler bugs have not found at all. This mainly includes bugs that are present at higher levels of optimisation but not at lower levels. This is clearly seen in bug reports for *gcc* where bugs are found when a higher levels of optimisation flag is passed e.g. *-O3* and then the bugs are not found at lower level e.g. *-O0* or *-O1*.

After looking over the bugs found so far, Dr Servetto and I came up with eight different categories for the classification of bugs found so far. These categories are roughly equivalent to the areas where Dr Servetto thought it likely I would find bugs in the compiler as per the list at the start of this chapter. This list of categories can be seen in the list below.

- Syntax is too liberal
- Syntax is too restrictive
- Type system is too liberal
- Type system is too restrictive
- Metaprogramming is too liberal
- Metaprogramming is too restrictive
- Reduction
- Miscellaneous

Most of these categories are fairly self-explanatory but two of the categories require a brief explanation. Firstly a bug that came under the ”Metaprogramming is too liberal” category would generally mean that the code produced unexpected behaviour as a result of metaprogramming. An example of this would a metaprogramming operation *O* that takes well-typed code and produces non-well-typed code. Correct behaviour would result in *O* taking

well-typed code and producing well-typed code. A further category that requires some clarification is the ”Reduction” category. This includes the 42 compiler producing wrong Java behaviour and the compiler producing non-well-typed code. Finally, the ”Miscellaneous” category includes all the bugs that did not fit nicely under any other category. Note that these categories are not final and could change in the future. Additionally some of the bugs uncovered over the course of this project do not fit nicely into a single category or would be hard to categorise accurately so are generally placed in the miscellaneous category.

3.4 Bug overview

Table 3.1: The categories and quantity of bugs found

Category	**Quantity**
Syntax (too liberal)	2
Syntax (too restrictive)	1
Type system (too liberal)	2
Type system (too restrictive)	1
Metaprogramming (too liberal)	0
Metaprogramming (too restrictive)	0
Reduction	3
Miscellaneous	5
Total quantity	**14**

Some comments can be made about the quantity of bugs (see table 3.1) found over the course of this project. The first thing that stands out is that the total quantity of bugs found is not as high as hoped. This is primarily due to the absence of automated testing of the compiler, all testing was done manually. On average around 10 tests have been written for each bug found (142 tests written, 14 bugs found). An additional interesting point to note includes the total lack of metaprogramming bugs discovered. This is due to no tests having been written that use the 42 metaprogramming system. Reasons for this include not having very much experience with metaprogramming and not being needed in writing simple programs. Future testing of metaprogramming in 42 will require specifically tailored tests and more in depth research into metaprogramming. Another topic worthy of discussion of the high quantity of bugs in the ”Miscellaneous” category. This category, as described earlier, includes all the bugs that were easily classified into the other categories or simply did not fit. Examples of this include the division by zero bug discussed earlier in this chapter (see figure 3.3). Therefore one could probably expect a high number of this category of bugs, especially due to the lack of automated testing so far.

4

Future Work

A significant amount of work has been accomplished so far in finding bugs in the 42 compiler. Although the quantity of bugs found is prehaps not as high as hoped, but the quality of bugs found, if all factors are considered, is high. One of the mains reason the quantity of bugs found is quite low is due to the fact that no automated testing tools have been thus far. Which means that the overall quantity of tests written so far is not very large. All tests have been written manually, and the majority of these manually wrritten tests have been fairly gerneral programs which have been written in a black-box testing style. These black-box tests did not take into account the compiler code or previous knowledge of the compiler. A minority of programs were written in a grey-box testing style, and these programs proved to be the most effective at finding bugs. They had a probability of 9.2% of finding a bug compared to the black-box style tests which had a probability of 3.1%. We can therefore conclude that the grey-box testing style was more effective when contrasted to the black-box testing style. A conclusion that can be drawn from these results is that a focus should probably be placed upon a grey-box testing style. It must be noted that this would not exclude the possibility of black-box tests being written in the future, as they could still be useful and black-box testing has found bugs so far.

The next step for this project is to incorporate automated testing tools such as fuzzers. These tools may potentially uncover a large amount of bugs by virtue of allowing far more tests to be created than would be possible with purely manually creating tests. It is a possibility that automated test may not be as effective as hoped in this scenario due to various factors discussed in Section 2.4.2 of this report. The possibilty of this will be mitigated by still writing manual tests. Automated testing will take place in parallel with manual testing. The future manual testing will almost certainly take on a more grey-box testing strategy, as so far, it has proven to be both more effective at finding bugs on the whole, and more effective per test written.

Overall, the future work on this project will consist of more testing, specifically targeted tests with knowledge of the code of the compiler, as well as the use of automated testing tools and techniques [15] [18] [20] [21] [22] to increase the quantity of tests produced. The combination of high quality targeted testing, and more general automated tests will provide a significantly more rounded testing strategy than has been used so far over the course of this project.

Appendix A

Short 42 programs

```
Nums: Collections.vector(of: Num) // Define a vector of Num.
// Gets the max value in an array.
C: {
   class method Num max(Nums arr) {

      var Num max = Num"-9999" // Set a low number. Note no Num.Min_VALUE

      with e in arr.vals() ( // Like for(int e : arr)
         if (e > max) (
            max := e
         )
      )
      return max
   }
}
```

```
Nums: Collections.vector(of: Num)

C: {
   class method Bool contains(Nums arr, Num target) {
      with e in arr.vals() (
         if (e.equals(target)) (
            return Bool.true()
         )
      )
      return Bool.false()
   }
}
```

Appendix B

Sample bug report

VectorBlankVariable.L42

```
reuse L42.is/AdemTowel02
CacheAdamTowel02:Load.cacheTowel()

Nums : Collections.vector(of: Num)
Main: {
   Nums arr = Nums[_;_;_]
}
```

Expected behaviour: Some sort of 42 error stating that variable "_" is not in scope or cannot be found.

Actual behaviour: java.lang.NullPointerException

```
Caused by: java.lang.NullPointerException
        at newTypeSystem.AG.g(TIn.java:92)
        at newTypeSystem.TIn.g(TIn.java:1)
        at newTypeSystem.TsOperations.tsX(TsOperations.java:46)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:137)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$X.accept(ExpCore.java:318)
        at newTypeSystem.Impl.type(TypeSystem.java:135)
        at newTypeSystem.TsMCall.tsMCall(TsMCall.java:73)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:141)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$MCall.accept(ExpCore.java:50)
        at newTypeSystem.Impl.type(TypeSystem.java:135)
        at newTypeSystem.TsBlock.dsType(TsBlock.java:139)
        at newTypeSystem.TsBlock.dsType(TsBlock.java:163)
        at newTypeSystem.TsBlock.tsBlockBase(TsBlock.java:69)
        at newTypeSystem.TsBlock.tsBlock(TsBlock.java:28)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:142)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$Block.accept(ExpCore.java:408)
        at newTypeSystem.Impl.type(TypeSystem.java:135)
        at newTypeSystem.TsMCall.tsMCall(TsMCall.java:73)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:141)
        at newTypeSystem.Impl$1.visit(TypeSystem.java:1)
        at ast.ExpCore$MCall.accept(ExpCore.java:50)
```

[h]

Appendix C

Project Proposal

1. Introduction

42 is a programming language "developed to allow the transparent cooperation and composition of millions of libraries at the same time". An implementation of 42 requires significant testing to attempt to minimise implementation bugs and this project aims to test the implementation of 42 for bugs.

Testing of the 42 implementation will be done in two different ways. Firstly a large number of 42 programs will be written to attempt to find any bugs and to informally prove the correctness of the implementation in certain cases. Thirdly, programs will be written to target specific parts of the implementation in a blend of the previously two described methods.

Evaluation of testing is difficult in nearly any scenario especially when testing a project with the size and complexity of the 42 implementation. Testing cannot show the absence of bugs therefore the evaluation of the project will be based on the number and severity of bugs found in the 42 implementation.

2. The Problem

The implementation of a programming is so large scale and complex that is highly likely that bugs exist in the implementation of practically any language [9] and 42 is no exception.

Bugs in the compiler can lead to incorrect machine code being generated from a correct source program.

Bugs in a language implementation can be extremely costly and adversely affect programmers working with that implementation as implementation bugs can cause the language to be harder to use, or as a worst case scenario, be impossible to use. Therefore this project aims to find and demonstrate bugs in the language implementation with the aim of improving the language for programmers working with it.

The aim of this project is not to prove the absence of bugs or deficiencies in the implementation of 42, as testing cannot show or prove the absence of bugs [7].

3. Proposed Solution

The testing of 42 will consist of two different approaches to testing. Firstly small programs will be written and their expected behaviour will be compared to the actual behaviour. Any differing behaviour is indicative of a bug in the language implementation. These will be general programs written without consulting the source code of the compiler. This is a

form of ”Black-box” testing as the test is built around strict specifications and the expected behaviour is compared to actual behaviour. If the intended and actual behaviour match the test has passed, and if they do not match the test has failed.

The second approach will consist of small programs written to target different areas of the compiler. These tests will be targeted as knowledge of the compiler will be used to write them. Potential flaws will be found in the compiler with different tools and approaches (static analysis, etc.) and these potential flaws will be exploited with programs written to target the found flaws.

Preliminary schedule

- Produce literature review (est: 1 week)
- Produce testing plan (est: 5 days)
- Black-box testing (est: 1 month (concurrent with gray-box testing))
- Gray-box testing (est: 1 month (concurrent with black-box testing))
- Preliminary report (est: 14 days)
- Evaluate progress (est: 4 days)

4. Evaluating your Solution

Evaluation of the testing of software is difficult as testing cannot prove the absence of bugs or faults therefore the success of this project cannot be based on the implementation of 42 being bug free. This means that the only real option for the evaluation of the project is by the quantity of bugs or faults found, and the quality of the found bugs.

Bibliography

[1] "42 - Metaprogramming as default - Tutorial." `http://l42.is/tutorial.xhtml`, 2018. [Online; accessed 19-May-2018].

[2] N. Amin and R. Tate, "Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 838–848, ACM, 2016.

[3] "Bug List." `https://gcc.gnu.org/bugzilla/buglist.cgi?chfield=\%5BBug\%20creation\%5D&chfieldfrom=7d`, 2018. [Online; accessed 07-June-2018].

[4] "The Unsound Playground." `https://lwn.net/Articles/342330/`, 2018. [Online; accessed 06-June-2018].

[5] "Known Issues for JDK 8." `http://www.oracle.com/technetwork/java/javase/8-known-issues-2157115.html`, 2018. [Online; accessed 08-June-2018].

[6] J. Harrison, "Formal verification at intel," in *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pp. 45–54, June 2003.

[7] E. W. Dijkstra, "Notes on Structured Programming - 249." circulated privately, apr 1970.

[8] "Mutability & Immutability." `http://web.mit.edu/6.005/www/fa15/classes/09-immutability/`, 2015. [Online; accessed 05-June-2018].

[9] F. Arnaboldi, "Exposing Hidden Exploitable Behaviors in Programming Languages Using Differential Fuzzing," p. 19.

[10] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[11] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM*, vol. 33, pp. 32–44, Dec. 1990.

[12] S. Blazy, Z. Dargaye, and X. Leroy, "Formal Verification of a C Compiler Front-end," in *Proceedings of the 14th International Conference on Formal Methods*, FM'06, (Berlin, Heidelberg), pp. 460–475, Springer-Verlag, 2006.

[13] S. Blazy and X. Leroy, "Mechanized Semantics for the Clight Subset of the C Language," *Journal of Automated Reasoning*, vol. 43, pp. 263–288, Oct 2009.

[14] "CompCert." `http://compcert.inria.fr`, 2018. [Online; accessed 26-May-2018].

[15] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," tech. rep., 1995.

[16] J. E. Forrester and B. P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 2000.

[17] "american fuzzy lop." `http://lcamtuf.coredump.cx/afl/`, 2018. [Online; accessed 27-May-2018].

[18] "Afl-based fuzzing for Java." `https://github.com/isstac/kelinci`, 2018. [Online; accessed 28-May-2018].

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 75–84, May 2007.

[20] "Javafuzz is a java classes fuzzer based on the the Java Reflection API." `https://github.com/cphr/javafuzz`, 2016. [Online; accessed 27-May-2018].

[21] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to Prioritize Test Programs for Compiler Testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 700–711, May 2017.

[22] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An Empirical Comparison of Compiler Testing Techniques," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 180–190, May 2016.

[23] "42 - Metaprogramming as default." `http://l42.is.xhtml`, 2018. [Online; accessed 19-May-2018].

[24] R. Seacord, "Dangerous Optimizations and the Loss of Causality." `http://www.eng.utah.edu/~cs5785/slides-f10/Dangerous+Optimizations.pdf`, 2010.

[25] "Can compiler optimization introduce bugs." `https://stackoverflow.com/questions/2722302/can-compiler-optimization-introduce-bugs`, 2010. [Online; accessed 07-June-2018].

[26] "Fun with NULL pointers, part 1 [LWN.net]." `https://stackoverflow.com/questions/2722302/can-compiler-optimization-introduce-bugs`, 2009. [Online; accessed 07-June-2018].