Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

GLSLsmith: a Random Generator of OpenGL shader programs

Author:
Bastien Lecoeur

Supervisor:
Prof. Alastair Donaldson

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2021

Abstract

Compiler testing is a long-running practise which aims at finding bugs in existing compilers through the compilation of numerous programs. The GLSLsmith framework is a random program generator for GLSL, a graphics programming language. By generating a large set of programs and compiling them on multiple compilers, GLSLsmith has led to the identification of 15 bugs.

Inspired by Csmith, the famous random program generator for the C language, GLSLsmith supports specific features of graphic languages as well as floating-point testing: by generating literals in a controlled way, GLSLsmith is able to suppress the risk of round-offs in floating-point operations, ensuring strict equality in all output values.

To be useful for compiler development, programs need to be converted into simpler tests by a process called reduction. Program generation or reduction, however, may introduce undefined behaviours which suppress the meaning of the program. By postponing the suppression of such behaviours after generation, GLSLsmith demonstrates a new way to handle undefined behaviours both at generation-time and reduction-time. This technique enables to use generic reduction programs, diminishing the engineering effort needed to perform compiler testing. GLSLsmith has been tested with three different reducers (C-Reduce, PERSES and glsl-reduce) comparing the quality of the minimized programs and evaluating the benefits of this new technique.

Acknowledgments

First, I would like to thank my supervisor, Prof. Alastair Donaldson, for his availability and his suggestions throughout the project despite the complicated online conditions. His feedback and his experience have been valuable not only for the project but also to develop my skills in a broader way.

I want to thank Paul Thomson from Google for his help on the installation of Switshader, as well as the development teams of SwiftShader and of the Mesa project for their cooperation in the bug reporting process. I am especially grateful to Marcin Ślusarz and Timothy Arceri for their prompt fixings of the bugs in the Mesa project as well as the insight they offered of the inner workings of the compilers.

Finally, I want to dedicate this project report to my parents who have been a constant source of support and who have always answered the phone during this weird year.

Contents

Chapter 1

Introduction

1.1 Importance of Compiler Testing

Compilers are complex pieces of software that convert human-readable languages to machine-specific languages and apply optimizations to improve the efficiency of generated code. Like any software, they can exhibit bugs and errors. Moreover, as compilers are central in application development, their bugs have the potential to affect multiple programs. It is difficult for developers, being used to looking for bugs in their own application code, to find that the compiler is to blame. This results in a loss of time and unnecessary frustration for the programmers.

Compiler testing helps to identify such bugs and remove them from compilers ahead of time, so that the bugs never affect programmers and end users. As the problem arises for all languages, most compilers are tested against large conformance test suites, covering most areas of the underlying language. As a result, usual codes are likely to be compiled correctly. However, conformance suites are not perfect as they cannot check every instruction combination of any given language. Fuzzing or random testing (as described in section 2.1) based on the generation of programs is another efficient way to find residual bugs as it is more likely to produce less-used code constructions.

While multiple ways to produce such fuzzing inputs exist, the most common uses a so-called random program generator (subsection 2.1.1) to generate valid code. By then comparing results across multiple implementations (subsection 2.1.2), it is possible to spot discrepancies. Such discrepancy indicates either an undefined behaviour (a point not described by the specification) or a bug. To be effective, a generator, therefore, needs to get rid of all language specific undefined behaviours (the ones for GLSL are described in section 3.1).

1.2 Contribution

The contribution of this project is GLSLsmith, a complete framework to test graphics compilers presented in chapter 4. GLSLsmith builds random programs from scratch for both GLSL and ESSL, two shading languages usually used to convert data to images. While GLSL compilers are validated by conformance test suites and have been tested extensively through metamorphic testing (see GraphicsFuzz in subsection 2.2.3), GLSLsmith managed to identify 15 new bugs. Especially 11 bugs are miscompilation bugs, which do not create any warning from the compiler but change the program output. 9 of the identified issues have been fixed as summarized in subsection 5.1.2. GLSLsmith have been tested on 5 different platforms finding bugs on each of them, thanks to its high coverage of language features (vectors, built-in functions, control-flow).

While floating-point values are often excluded from differential testing tools due to their imprecision (subsection 3.1.6), GLSLsmith is one of the first generators to support floating-point programs. Extending an idea from one of the Orange generator tools [1], GLSLsmith ensures that floating-point operations do not introduce round-off[1], GLSLsmith ensures strict equality between the results, effectively enabling differential testing (subsection 4.2.4).

GLSLsmith supports a fully automated pipeline to build and reduce tests for graphic compilers (section 4.4). By postponing the undefined behaviours suppression after generation, GLSLsmith can ensure that a program stays well-defined through reduction. Thanks to this property, any reducer can be used without the risk of introducing undefined behaviours, removing the potential engineering effort required by language-aware reducers. GLSLsmith have been tested with 3 different reducers: C-Reduce [2], PERSES [3] and Glsl-Reduce [4], providing a way to compare the quality of reducers without the problem of undefined behaviours and evaluating the impact of post-processing (section 5.3).

[1]The idea presented in the report was developed independently but matches the approach presented in [1], found in a late stage of the project.

Chapter 2

Background

In order to test compilers, the GLSLsmith framework relies on the random testing process whose main steps and difficulties are described in section 2.1. As multiple random program generators have been already designed, the section 2.2 concisely presents three of them (Csmith, YARPGen and GraphicsFuzz) which influenced the design of GLSLsmith, presented in chapter 4. The last section of the chapter (section 2.3) introduces the main existing graphics frameworks, their general principles and the tools which enable to test them efficiently. The chosen graphics framework, OpenGL, its specific features and undefined behaviours are then described in chapter 3, along with information on the tested compilers and reducers.

2.1 Random testing process

2.1.1 Program generation

The first step of randomized compiler testing involves generating programs. This generation step includes the code file for later compilation but can also generate extra files to provide information about the test. Randomized compiler approaches are usually divided into two categories: *generator-based* approaches and *metamorphic-based* approaches [5; 6]. While both methods generate randomized programs, they may exhibit different properties than can be used for bug identification (discussed in subsection 2.1.2).

Generator-based approaches generate test programs by strictly following the grammar of a language avoiding *undefined behaviours*. Programs are built from scratch following the language specification. While being particularly effective, generator-based programs cannot be transferred between languages, requiring to write a complete generator per language. Multiple generators have been proposed, such as Csmith [7; 8] and YARPGen [9], focusing on C and C++ compilers. Generators usually use context-dependent building rules to expand generated code and then dump it into a file. Even if generators produce a wide range of expressions, they often exhibit the same bugs. This problem is rooted in the probability distribution of the generation-procedure. As long as the distribution is identical, programs, while being different, show similar properties: even if new bugs exhibiting programs could in theory be generated, they have an unrealistically low probability to get produced. *Swarm testing* [10] therefore proposes to change the program generation configuration across time, leading to new bug discoveries.

Metamorphic approaches generate test programs from a seed (or pool) of already existing programs by applying a set of transformations, which do not affect the outputs of the program. Those transformations are said to be *semantic-preserving* [11]. Different types of mutations are possible depending on the objective of the generator. As they don't generate code from scratch, they don't necessarily need to understand it completely, making them more reusable. However, as they rely on a pool of existing programs, the seed of starting programs that exhibit interesting characteristics (such as a great feature diversity) impacts their capabilities. Interestingly, it is possible to seed such a program with the result of a random program generator to expand the capabilities of the underlying generator or target specific compiler features (such as optimizations) [12]. Depending on the applied transformations, a set of programs can exhibit the same result while using different code paths, for example, by replacing for loops by while loops. This characteristic can later be used for bug identification.

2.1.2 Bug identification

Some bugs can be found using only a random program generator when a valid code is given to a compiler but fails to compile, or code crashes at run time without an expected reason. Such errors are called *Internal Compiler Errors* or simply *compiler crashes* [12; 13]. However, even if a test program does not crash, the program is not necessarily correct. Compiled code might exhibit an actual output different from the expected one. Such bugs are often called *miscompilations* [5; 12]. In such a case, it is impossible to rely only on the execution of a single program on a single compiler to find the bug. Strategies have therefore been designed to identify those by comparing an expected program output with its implementation.

Oracle testing refers to a strategy in which one has an oracle, which can decide the correctness of the output. It can be seen as the most convenient situation as it directly assesses if a given compiled program gives the correct result. While contract-based languages have used contracts as an oracle [14], most languages do not offer such functionalities. Thanks to their inner organisation, some generators can effectively be used as oracles [1; 9] by computing the program at generation-time. When available from generation, the result is likely to change through reduction (discussed in subsection 2.1.3): the generator would need to evaluate the new outputs for any reduced program (and therefore could work as an interpreter for any program). For similar reasons, generators that compute all values at generation time have limited control-flow support.

Differential testing [15] eliminates the need for an oracle by testing multiple compiler versions / implementations. By collecting the results from all the compilers, it is possible to see if they all return the same answers. A compiler that returns a different value from all other implementations can exhibit a bug and therefore the code of such program should be kept, as demonstrated on the left of Figure 2.1. This approach relies on the assumption that independent compilers are unlikely to exhibit the same bug. Differential testing is a commonly-used setting, as it has the benefit to find bugs in multiple compilers, increasing the chance to identify bugs with a single generator. This method is practical but introduces the problem of *undefined or compiler-dependent behaviours*. Undefined behaviour is a situation in which the specification of a language allows compilers to behave differently. They arise from various sources like the finite representation of numbers and the definition of arithmetic operations [8]. One of the challenges of random testing is to produce programs that are free of compiler-dependent behaviours while exploring as many parts of the language as possible [8; 9].

Metamorphic testing [5; 6] is another way to replace the oracle. While differential testing uses different compiler versions against the same program, metamorphic testing uses multiple program versions against a given compiler. The underlying assumption is that one can produce equivalent codes in terms of entries and outputs. If one of the two versions creates a different output on the compiler, then a possible bug has been found and the code should be kept, as demonstrated on the right of Figure 2.1. Two programs need to be similar only for the given inputs, leading to *equivalence modulo input* (EMI) testing [12]. EMI testing leverages that one program can have a different behaviour if such behaviour never propagates to the program results (never reached code path for a given input, equivalent instructions). EMI testing finds compiler defects as various codes lead to different internal optimization by the compiler [16]. It is possible to anticipate that controlled changes in program instructions should lead to similar results. By relaxing the condition of equivalence between programs, it enables to compare floating-point values that admit uncertainty. As metamorphic testing requires building a family of programs related to a given original one, it is more likely to be used in the context of mutation-based testing (given that the selected mutations do not affect the final code output) [11]. Multiple generators relying on metamorphic testing have been designed its creation such as Orion [12], Athena [16] and Hermes [**?**] as well as one of the Orange generator [17].

Both differential and metamorphic testing are built on the assumption that the output values will be identical across programs and compilers implementations. For integer-based numbers, a result is effectively uniquely defined. However, for floating-point based values, multiple values can be correct as floating points are only approximate finite representations of real numbers. Standards, therefore, accept that the performed operations are only approximate (see subsection 3.1.6 and subsection 4.2.4). These approximations can propagate to the final result, such that the results are unlikely to be equal even for correct programs. As differences between floating-point results are tolerated by the specification, a first approach is to exclude

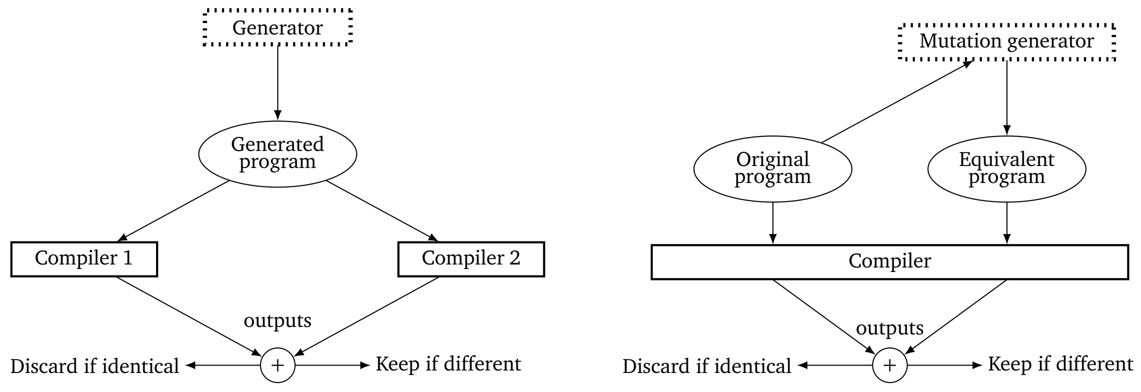

Figure 2.1: Differential testing (left) vs Metamorphic testing (right)

all floating-point values [8]. However, as this solution limits the testing capabilities on most languages, values can be considered to agree as long as they are within a given range of each other. Recent program generators, therefore, deal with floating-points restricting and checking their proximity with a threshold [9]. This approach can be generalised with different data types such as images, which also admit uncertainty [11].

A second approach to floating-point testing for C is offered by one of the Orange program generator [1]. By limiting the support of floating-point to the arithmetical operations and constraining literal generation to match integer values within range $[2^{-m}; 2^m]$ (where m is the fixed number of bits in the *mantissa* for a given type), the generator is able to compute all expected outputs including floating-point values. As rounding errors can still occur on the result of operations if they to escape those bounds, the generator checks and rewrites any possibly offending operation.

2.1.3 Test case reduction

A crashing program or one returning a miscompilation output (while being free of undefined behaviour) probably exhibits a compiler bug. However, such a program can be unreadable for a human due to its random nature. The given code can be very long, contain unused or irrelevant code which hides the defect. It is unlikely that such a program could be accepted as bug proof. It is therefore essential to identify the underlying code lines triggering the bug and remove all other unnecessary information. The process to perform this task is called *test case reduction*. It has two purposes: identifying unique bugs (multiple programs might exhibit the same underlying bug or a program could contain two bugs) and making the code easy to read for review (and fixing). The reduction can be done by hand or using so-called *reducer* programs to find shorter code versions with the same properties by performing operations on the original code while ensuring that chosen properties are satisfied by the shorter versions. Using a reducer usually requires writing an *interestingness test* which indicates if a given program matches a given property.

Reducers exist for common languages such as C (C-Reduce) [18; 2] or as language-independent tools such as Delta Debugging [19] and Hierarchical Delta Debugging [20]. They all rely on variations of the Delta Debugging practice. *Delta debugging* examines every possible suppression from the code to find if the resulting code still exhibits a given property (a bug). The resulting file is then *1-minimal*, meaning that any further suppression from the reducer would suppress the bug. Some reducers are completely *language-agnostic* and consider codes as a list of strings. Some are *grammar-aware*, meaning that that the resulting code is still syntactically valid and code is recognized as written in the given language. Other reducers are optimized against one language to perform further operations. An ideal reducer would make as few assumptions as possible on the underlying language while finding shorter examples. While reduction implies removing as much code as possible, it may be suitable to replace some existing code with longer but more readable code [18]. New reducers such as PERSES [3] use only a language grammar to perform syntactically valid reduction, improving reduction speed while not being *language-specific*.

A first challenge associated with test-case reduction is the management of undefined behaviour across reduction. While the generator and the original code is free from undefined behaviour, it is unlikely that code produced by deletion will have the same property. A common way to deal with such a situation is to exclude programs exhibiting compiler-dependent behaviour with one or multiple *sanitizers*, requiring such programs to be available in a given language. Sanitizers aim at preventing accesses to uninitialized memories, ensuring that loops terminate and detecting arithmetic exceptions. Such sanitizers include UBsan [21] or Frama-C [22] for the C language. They are usually executed as part of the interestingness function. Without such securities, a bug-triggering program will likely be converted to a program exhibiting an undefined behaviour giving the same value, therefore, destroying the whole interest of the test [23].

A second challenge is known as *bug slippage* [24]: across reduction, a different bug can be triggered by a given program. While it can lead to the opportunistic discovery of a new bug, it can fall back on a commonly experienced bug or create a bug in a different compiler, leading to an unplanned situation for the reducer. Similarly to the problem of undefined behaviours through reduction, one way to reduce the risk of bug slippage is to look for the most precise cause of the error as possible (by differentiating errors for each compiler or affected values in the program result, for example).

In both cases, the quality of the interestingness test is crucial to the correct execution of the reducer. The way compilers report specific defects and the tools available in a given language are also criteria that determine the capabilities of the test.

2.2 Existing random program generators

2.2.1 Csmith

Csmith [2; 18] is one of the most famous, adapted and used program generators for the C language. It uses probability tables to add new instructions in the given context, adding new instruction/sub-instruction one at a time and backtracking if the new code is problematic. It supports a range of C features such as loops and branch control. It deals with arithmetic-based undefined behaviours by adding *safe wrappers* around all math operations, testing for unacceptable values, and defaulting if an undefined behaviour is reached. Wrappers, generated as either functions or preprocessor macros, are necessary to remove undefined behaviours (one of them is shown in Figure 2.2). They are, however, criticized for the pattern they introduce in the code, reducing the number of optimizations that the compiler might apply, for example, to group arithmetic operations together. An extension named `CsmithEdge` [25] has been proposed to remove unnecessary wrappers. It runs the program twice, checking wrapper values on a first run to remove unnecessary ones from the code before recompiling it. This extension demonstrates the possibility to perform metamorphic testing by applying semantic-preserving transformation on randomly generated programs. While Csmith permits the *non-termination* of programs (which are caught back by time outs), it deals with pointer-based undefined behaviours as well as with evaluation-order based undefined behaviours (see subsection 4.3.6 Parameter order control for the adaptation in GLSLsmith) by performing an analysis of the variables which are used into any given context. Csmith greatly inspired generators for other languages such as CLsmith [26] for the OpenCL language, VeriSmith [27] for the Verilog language, and GLSLsmith.

```
static uint64_t safe_div_uint64_t (uint64_t ui1, uint64_t ui2)
{
  return (ui2 == 0) ? ui1 : (ui1 / ui2);
}
```

Figure 2.2: Example of wrapper for division operations with unsigned integers, testing explicitly for 0 values before performing the division

2.2.2 YARPGen

YARPGen [9] is a recent generator for the C and C++ languages relying on a recursive generation function. The generation function being aware of the current context only proposes valid generation. It does not use wrappers for arithmetic-based undefined behaviours but instead computes the values of the different variables during generation. Knowing every value in advance, YARPGen can remove undefined behaviours by rewriting instructions containing problematic values in a similar way to the one of the Orange generator [1]. It does not use a complete interpreter for C

/ C++ and limits its generation to function call-free programs. As YARPGen computes all variables affection including the output values, it provides an oracle of the program and the result of the generation could be used to check the values that a compiler would produce. However, YARPGen does not use the computed values to check the program outputs. It instead relies on differential testing, enabling reducers to change the output values. It only provides limited loop support and does not deal with pointer arithmetic, two critical features of the C / C++ languages. YARPGen instead focuses on finding compiler bugs that appear through erroneous arithmetic optimizations unlikely to be triggered by Csmith due to its wrappers.

Another feature of YARPGen is its ability to use *generation policies* like the swarm-testing principle to randomly skew the generation probabilities of one feature compared to others. Swarm-testing (and generation policies) has been designed to experiment with more diverse code paths in a similar execution time. For example, bugs might only appear if a long series of unsigned integers was to be declared in a row, however choosing uniformly between integers, unsigned integers and boolean, code exhibiting the bug would appear under extremely low probability. Randomly skewing the distribution would increase this probability and, therefore, the durability for which a generator can be used.

2.2.3 GraphicsFuzz

GraphicsFuzz [11; 4] is a metamorphic-testing framework applied to graphics shading languages. It uses a pool of original programs to which it performs a series of semantic-preserving transformations to create variants. Such changes include *dead-code injection*, *program-workflow and arithmetic rewriting*, and *code injection* from another program. As the fuzzer generates new entries to the program, it can generate unreachable code that the compiler can't optimize. Such code can therefore trigger different paths in the compiler and result in crashes or miscompilation behaviours. To inject live code, the fuzzer ensures that the new text will not interact with the behaviour of the original code, changing variable names and rewriting functions. One of the advantages of this strategy is that the resulting program is made of code found in real-life applications. Errors found by GraphicsFuzz are therefore more likely to be of interest to compiler developers.

Images produced by graphics shaders are, as floating-point values, likely to differ from one implementation to another while they all remain valid. To compare pictures and spot differences across variants, GraphicsFuzz uses dedicated metric computing the distance between the colour histograms of two pictures. If the distance is above a threshold, pictures are considered as different, and a bug has possibly been found. Even if false negative or false positive cases are possible, human experimentation has proved that the measure was accurate enough to be used as a proxy for similar images.

GraphicsFuzz proposes a new way to deal with the reduction problem. As produced shaders are related to an original one by a set of semantic-preserving transformations, it is possible to apply the principal of Delta debugging to find the minimal transformations to apply to a shader to reproduce a bug. While it is likely that the resulting shader is not of a minimal size, the bug-exhibiting shader is however of interest to report if the original shader is well understood. Moreover, this approach helps with the problem of new bug identification. Two bugs are likely to be distinct if they are produced by different transformations. On the contrary, two bugs which share many transformations may be duplicate. These ideas have been adapted to spirv-fuzz [28; 29], a fuzzer directly targeting the SPIR-V representation of Vulkan, and integrated in the set of tools published in official Khronos repositories. The GraphicsFuzz framework also includes a more traditional language-aware reducer (discussed in subsection 3.2.3).

2.3 Overview of Graphics Programming

3D images are inherently expensive to produce as they require numerous computations. However, those computations can mostly be effected in parallel and largely benefit from the dedicated hardware support offered by graphic cards (GPU). As the GPU is separated from the CPU, *Graphics programming frameworks* offer a unified way to communicate and execute programs on the GPU. Graphics programming frameworks, therefore, offer at least two capabilities: GPU / CPU communication and GPU-side program execution.

To perform the communication, Graphics programming frameworks offer a common API that is implemented by graphic card makers. The API is then exposed to the CPU-side computations through a low-level library available in multiple languages. To build a graphic program, it is necessary to write code in a language that will perform CPU computations and communicate with a GPU. Such language is known as the *host language*.

The GPU part of the program is then written in a dedicated language, commonly known as a *shading language*. As multiple steps are necessary to create a picture, programs are written as multiple code pieces known as *shaders* with dedicated inputs and outputs, the last output being the expected image. It is known as the *graphic pipeline*. Across the pipeline, shaders are expected to match their outputs to the inputs of the next one, a process that is ensured at linking time to create a complete *graphic program*. Most Graphics programming frameworks associate a unique shading language to an API.

2.3.1 Available languages

As Graphics programming frameworks are interacting with low-level hardware, their functions partly depend on the OS. Some are therefore tied to specific OS / hardware. For example, DirectX is only supported by Windows and Metal is only supported by macOS / IOS. Standards developed by Khronos, OpenGL, OpenGL ES, WebGL and Vulkan are on the contrary available on most platforms. As Khronos only provides the specifications of the API, multiple compilers and drivers exist for some platforms. Some are directly provided by graphic cards manufacturers, while others are developed as open-source projects or independent companies. Except for Vulkan, which uses a binary format, all the other languages support a text-based shader language: (HLSL for DirectX, GLSL for OpenGL, ESSL for OpenGL ES and MSL for Metal).

As OpenGL, OpenGL ES, WebGL and Vulkan are developed by Khronos [30], they share a part of their APIs. Some projects, therefore, support the different shading languages of Khronos coherently. For example, any OpenGL ES shading language (ESSL) specification is based a subset of an OpenGL (GLSL) specification (some operations on arrays and precision qualifiers are distinct). GLSLangValidator [31], the reference code validator can be used for OpenGL and OpenGL ES codes as well as to translate code to the binary format of Vulkan shaders (SPIR-V).

If the coexistence of multiple similar Graphics programming frameworks can be surprising, their existence is rooted in the hardware support offered by different execution platforms. As OpenGL ES and WebGL are lighter versions of OpenGL, there are especially interesting for 3D rendering in Smartphones (OpenGL ES) and web browser (WebGL). On computers, most driver implementations can accept either ESSL or GLSL as a valid shader. Vulkan has then been developed as a way to unify Graphics programming frameworks (as discussed in the subsection 2.3.2).

Framework	Supported Platforms	Shading Language	Textual
Direct X	Windows	HLSL	✓
Metal	MacOs / IOS	MSL	✓
OpenGL	Windows / Linux	GLSL	✓
OpenGL ES	Windows / Linux / Android	ESSL (GLSL ES)	✓
Vulkan	Windows / Linux / Android MacOs / IOS[1]	SPIR-V	x

Table 2.1: Summary of the most-common Graphics programming frameworks

2.3.2 Conversions between graphics programming frameworks

While it is common for OpenGL drivers to support OpenGL ES shaders (ESSL format), there is no direct support to use a different shading language in a given framework. It is an important issue to port a program to different platforms. The Vulkan specification has been conceived as a way to solve that issue.

[1]No native support from Mac and IPhone, but a libary performs the conversion

Vulkan provides the specification for a binary shader language (SPIR-V) which can be used to target Vulkan-compatible back-ends. It offers a common *intermediary representation* for shader code which can be generated from all textual languages except Metal Shading Language (MSL). The representation can then be optimized independently from the back-end and can be converted to any shading language (including MSL) or executed on a compatible back-end.

While shaders can be effectively converted from one language to another, it is not sufficient to fully perform the conversion from one Graphics programming framework to another. Especially, the communication of the data to the shader is not converted and data can be loaded incorrectly in the API. Some projects, such as ANGLE [32] or Zink [33], try to close that gap by providing a way to port an API to the other available back-end APIs per platform. Specifically, ANGLE from Google ports OpenGL ES and WebGL to Vulkan, and Direct3D, while Zink from the MESA project performs the conversion from OpenGL to Vulkan.

2.3.3 Scripting languages for shader testing

While real-life applications must provide features unrelated to graphics, traditional host languages such as C or Java are not well-suited to test shaders. Introduction codes to draw a single triangle usually contains 100 lines for the OpenGL API [34] and 1000 lines for the Vulkan API [35]. They require to write a large number of low-level instructions to perform repetitive tasks (such as passing the correct data or compiling the shader) and need to be compiled to execute programs. It is therefore necessary to largely rewrite the host language code. Moreover, as the final binary equivalent of a shader is largely dependent on the hardware, shaders are only compiled from their text form during execution and are often stored in separate files.

Scripting languages have been designed to provide an abstraction to those low-level instructions and to test shaders in an easier and isolated way. They provide operations only related to graphics to build a single image/output and to test the result values. They also offer a convenient way to execute programs in a similar environment across multiple platforms. Moreover, as the tests are written in a single textual file, it is easy to read and to share to other people, especially for bug reporting (see subsection 5.1.1).

Multiple scripting languages exist and are supported by different projects to test specific APIs. It is usual for projects to ask that bug triggering shaders are enclosed in their preferred scripting language. Scripting languages are often named by their testing program. For example:

- `shader_runner` is the test command for the Mesa / piglit [36] project, it deals with the OpenGL API and is centred on the writing of self-contained test shaders with pre-determined results. Equivalent versions of the tool exist for the OpenGL ES APIs.

- `VkRunner` [37] is a scripting framework based on the test language of `shader_runner` to execute Vulkan code. It supports the same shading language (GLSL) but converts it to SPIR-V to execute it. SPIR-V code can also directly be used. It can be used on Android as well as on Linux and Windows.

- `Amber` [38] is a scripting framework developed by Google. It is used as a runner for the Vulkan Conformance Test Suite (Vulkan-CTS). It supports Android and can be used to probe the difference between two images using histogram distance (As designed by the GraphicsFuzz project and explained in subsection 2.2.3). Tests can be written either in the same format as for `VkRunner` or using an independent scripting language `AmberScript`.

- `ShaderTrap` [39] is a scripting framework developed to support OpenGL and OpenGL ES APIs. It supports similar functionalities as the previously cited scripting languages, uses an independent scripting language (close to AmberScript). It offers the possibility not only to compare values and images at run-time but to dump results for a-posteriori analysis.

Chapter 3

Technology choices

GLSLsmith focuses on the testing of the shader compilers. It does not stress framework APIs and relies on a scripting framework to perform the necessary calls to the back-end. As OpenGL and OpenGL ES are the most supported frameworks and can be executed on Vulkan targets easily thanks to ANGLE, GLSLsmith focuses on the testing of GLSL / ESSL shaders. The main features of the two languages are therefore described in section 3.1). As multiple shading languages versions exist, the generator is focused on the most recent one ported to all the tested target platforms (see section 3.3). These standards are GLSL 4.5 [40] and ESSL 3.1 [41].

While any scripting framework could be supported by the shader generator of GLSLsmith to build shaders, the choice of the scripting framework influences the number of platforms being tested. As the OpenGL ES APIs calls can be converted to Vulkan through ANGLE, it is possible to test both Vulkan and OpenGL compilers through a scripting language supporting the OpenGL ES API (`shader_runner`, `ShaderTrap`). `shader_runner` does not offer the capability to dump values to files and is, therefore, more difficult to use for differential testing. As ShaderTrap also offers Android support, the GLSLsmith framework could be ported to Android as part of future works. GLSLsmith therefore dump shaders into `ShaderTrap` files.

One of the the objectives of GLSLsmith is the independence towards the reducer used in conjunction with it. As GLSLsmith does not make any assumptions on the used reducer, three different types of reducers have been used (see section 3.2) and their results compared (see section 5.3). While other reducers exist, they have been chosen to represent the different reducers that can be available for an arbitrary language. The tested reducers are `C-Reduce` as a language-independent reducer, `PERSES` a grammar-aware reducer and `Glsl-reduce` a language-aware reducer developed for GLSL as part of the GraphicsFuzz project. Their functionalities are describes in section 3.2.

3.1 Interesting GLSL / ESSL features

The GLSL and ESSL languages are based on C. While a full description of the two languages is unnecessary for this report, some features create specific generation and undefined behaviour challenges and are therefore presented here. As some features might have different support across GLSL and ESSL and as ESSL is a subset of GLSL, all features are presented for ESSL 3.1 and are supported by GLSL 4.5.

3.1.1 Shader communication

Shaders are usually assembled into a full graphics pipeline and act on different steps of the rendering to produce a picture. However, OpenGL offers a second pipeline called the *compute pipeline*.

In a compute pipeline, a single *compute shader* is used to perform multi-threaded operations on the GPU. Communication to such shader is achieved differently from the normal rendering pipeline by using *buffer* objects. The values of buffers can be populated from the API and they can be used and updated during shader execution. As the operations on a compute shader can be multi-threaded, buffers are shared across all the threads of a single compute shader.

From the point of view of the API, buffer data is considered in binary format. Their interpretation and typing are then inferred by the types declared in the shader. To use such buffer in a shader it is required to indicate its location (through a binding attribute), a name and the description of its different components. By default, compilers are free to arrange data in the buffer in any layout format as they like, which can be queried by the host language. However, it is possible to ask for a specific layout through an attribute such as the `std140` and `std430` which describe the stride and the alignment of the base types. While values are provided at execution time by the API calls, the values are unknown at compilation. The Figure 3.1 presents an example of such buffer.

```
layout(std430, binding = 0) buffer_0 {
    int ext_0;
    int ext_1[2];
    float ext_2;
}
```

Figure 3.1: Declaration of a buffer object named buffer_0, of layout format std430 bound to location 0. The buffer contains an integer ext_0, an array of two integers ext_1, and a float ext_3.

3.1.2 Basic types and operations

ESSL supports signed and unsigned integers, floats and booleans. Values of all types can be grouped in built-in vector types (from 2 to 4 elements). The relevant types are `ivecX`, `uvecX`, `vecX` and `bvecX` respectively where X is the number values. floating-point values can be grouped in two-dimensional matrices (the two dimensions are between 2 and 4). The relevant types are `matX` for square matrices of dimension X and `matXxY` for all matrices of dimensions X $\times$ Y (including square matrices).

The four traditional arithmetic operations (+, -, * and /) are supported for all arithmetic types. Operations between a vector and a scalar are defined component-wise and result in a vector of the same dimension. In the case of matrices, the multiplication operator is defined as the matrix-wise multiplicative operator and is defined for all mathematically well-defined cases, including vector / matrices multiplication and matrices / vector multiplications shown in Figure 3.2.

Integer-based types all support modulo operations (%), left and right shifts (<<, >>) as well as bitwise and, xor and or (&, ^ and |). Operations between a vector and a scalar are defined component-wise.

```
// Variable declaration
uint a = 0u;
ivec3 x = ivec3(1,2,3);
mat2x3 p = mat2x3(1.0f,2.0f,3.0f,
                  4.0f,5.0f,6.0f);
bool f = true;

// Examples of operations
uint b = a % 3u; // scalar / scalar operation
ivec3 y = x - 2; // vector / scalar operation
mat3 q = mat3x2(1.0f, 1.0f,
                1.0f, 1.0f,
                1.0f, 1.0f) * p; // matrix / matrix operation
vec2 r = p * vec3(1.0f); // matrix / vector operation
```

Figure 3.2: Examples of variables declaration and operations. The matrix q presents an example of matrix multiplication between a 3 $\times$ 2 matrix and a 2 $\times$ 3 matrix, resulting in a square 3 $\times$ 3 matrix. The multiplication of the matrix p by a vec3 results in the new vec2 variable r.

While integer addition, subtraction and multiplication are free from undefined behaviours (values are required to wrap if they overflow or underflow), divisions introduce two undefined behaviours in the case of the division by 0 or in the case of the division of MIN_INT by (-1). Integer-based operations also introduce undefined behaviours for the modulo operations (both operand need to be positive) and shifts (the right operand needs to be between 0 and 32 included). Floating-point operations all include undefined behaviours which are detailed in subsection 3.1.6.

Matrices and vector components can be accessed through the same notation as array indexes using []. Using a single index on a matrix returns the column of the given index (as a vector). Vectors elements can also be accessed through the **??** notation as demonstrated in Figure 3.3 which all component is represented by a letter from a set (r, g, b and a), (x, y, z, w) and (s, t, p, q). While components can be duplicated in a swizzle for lecture, such swizzle cannot be written.

Conversions between all types are permitted as long as enough values are provided for the resulting types. It is possible to provide a larger argument than necessary, in which case, only the first values are kept as shown in Figure 3.3. Vectors and square matrices can also be initialized from a single value. In that case, all the components of the vectors are initialized on the value, while in the case of matrices, only the diagonal is affected.

The order of evaluation of operations is largely undefined. The left or right operand can be evaluated in any order for most operations except assignments and boolean operations (and &&, or ||). A more precise description of undefined behaviours related to the evaluation order is given in subsection 4.3.6 along with their suppression by post-processing.

```
// Examples of swizzles and array accesses
int c = ivec3(1, ivec2(2, 3)).yyzx.rgb.s; //c = 2
vec2 d = mat2(1.0f)[1]; // d = vec2(0.0f, 1.0f)
float e = mat4x4(mat3(2.0f))[2][2]; // e = 2.0f

// Examples of type conversions
int f = int(false); // f = 0
vec2 g = vec2(1.0, uvec4(1u, 2u, 3u, 4u)); // g = (1.0, 1.0)
```

Figure 3.3: Examples of swizzles and type conversions. c is built from multiple swizzles which is equivalent to yyzx.s (rgb is the identity order) and then as y (s designates the first element). d is built from the square identity 2×2 matrix, by taking its second column. g is built from a floating-point and an unsigned vector for which all but the first value are useless.

3.1.3 Qualifiers

For any type described before, it is possible to precise extra constraints or expectations that the variable implementation should meet. Such constraints are implemented as *qualifiers* such as `const` which precise a compile-time constant to the compiler. Variables usually have multiple qualifiers which are omitted when they are kept to their default values.

For example, all arithmetic variables admit a precision qualifier that precise the size of their binary representation. In the specific context of a compute shader, all arithmetic variables are declared by default as `highp` for high-precision variables. In high-precision, floating-point values are expected to match the 32 bits implementation of the IEEE-754 format and signed and unsigned integers are expected to be stored on 32 bits. The other available qualifiers `mediump` and `lowp` only describe minimal sizes for the variables but implementations are free to choose a size for their operand. While GLSL 4.5 offers support for precision qualifiers, they are considered to be of no-effect in most cases and floating point values are guaranteed to be represented on 32 bits.

Qualifiers can also be applied to buffer and buffer values. For example, the `layout` qualifier has been used in Figure 3.1 to describe the location and the format of the buffer. Buffer qualifiers include `coherent` which indicates to the compiler that read and write operations need to be coherent across all shader invocations, `writeonly` and `readonly` to precise a write / read-only buffer. Such qualifiers can be either declared on the full buffer or added only to a specific buffer member.

3.1.4 Functions

GLSL functions include both *built-in* and *user-defined* functions. As for variables, parameters are qualified to indicate when they need to be copied to and from the function. `in` parameters are only copied to the function at beginning of their execution, `out` are copied from the function to the calling context at the end of the execution and `inout` are copied to the function and copied back as described before. Functions all have a return type which can either be `void` or one of the types described in subsection 3.1.2. The type of the parameters is ensured at function declaration time.

Contrary to C, expressions used as parameters are guaranteed to be evaluated from left to right at the call time of the function. The order in which values will be copied back is, however, undefined. It is therefore undefined to call a function with the same variable for multiple `out` or `inout` parameters as represented in Figure 3.4.

Each shader contains a `main` function which is called at the execution start. There is no nested function declaration and no forward declaration in GLSL. It is however possible to declare multiple time the same function as long as only one declaration matches a function prototype. Overloaded functions are supported.

Multiple built-in functions introduce undefined behaviours if parameters are out of bound. It is for example the case of `clamp` for which the minimal bound needs to be lower than the maximal bound.

```
// prototype without definition
void g(in int x, out int y, out int z);

// f uses the prototype of g without the definition
ivec3 f(in int x, inout int y, out int z) {
    g(x, y, z);
    return ivec3(x, y, z);
}

// g is then given a definition
void g(in int x, out int y, out int z){
    y = x;
    z = x+1;
}

void main(){
    int a = 2;
    g(a, a, a); // undefined behaviour
    // a = 2 or a = 3 are both valid
}
```

Figure 3.4: Function declarations and undefined behaviours examples. The undefined behaviour in the call of g is caused by the use of the variable `a` for both formal parameters `y` and `z`, resulting in an undefined order of copy back. `a` can therefore contain the value of `y` (the initial value of `a`) or the value of `z` (resulting in `a + 1`).

3.1.5 Compute groups

While all shaders are inherently executed in parallel, it is possible to explicitly configure compute shaders to be executed in *workgroups*, containing one or more local threads (the actual number is defined by its *local size*). The local size of a compute shader is given directly in the shader code and corresponds to the number of threads necessary to execute the shader on a given set of values input. While the actual order of execution of threads is undefined, it is possible to synchronize the thread executions thanks to built-in functions. On the other hand, the number of workgroups is defined at execution time through the API to execute one shader across multiple sets of values. The order of execution between all workgroups is undefined, and they should be all independent.

Compute shaders define specific variables to query either the workgroup number or the local number. Thanks to those variables, it is possible to differentiate between the *shader invocations* both within a workgroup or in between workgroups to query the appropriate data from the buffers. While data access is incoherent by default, it is possible to qualify a buffer as `shared` to ensured that its values are coherently handled with the `barrier()` function.

3.1.6 Floating point precision

As described in subsection 3.1.3, ESSL supports multiple floating-point representations based on the IEEE-754 2008 specification [42] (More details about the IEEE representation are given in subsection 4.2.4). While the representation should logically follow the IEEE specification, the ESSL specification relaxes multiple constraints associated with floating-points, leading to a number of undefined and compiler-dependent behaviours.

Floating-points are required to support the full-range of finite values from the IEEE single precision $]-2^{128}, 2^{128}[$ as well as both infinities. However, there is no obligation to support `NaN` values (which can arise from operations such as $0/0$). Moreover, as the IEEE representation supports two representations of 0 (one positive and one negative), compilers are permitted to exchange one representation to the other.

While the floating-point representation in itself leads to compiler-dependent behaviours, the specification also relaxes the precision of operations and function returning undefined behaviours. It is possible to separate three categories:

- **Rounded values**: Arithmetic operations such as $+$, $-$ and $\times$ are exact except for rounding errors. Functions which are described as a combination of such operations are also correct except for rounding errors. Types conversions are also included in this category. However, the rounding mode is undefined and cannot be set by the application.
- **Defined precision**: The / operation as well as some arithmetic functions (`log` and `exp`) have a fixed authorized precision. This precision is expressed in terms of *units in the last place* (ULP) giving the number of bits that are allowed to differ between implementations. This category of operations is specific to the GLSL specification and significantly lowers the expectation of the IEEE standard regarding floating-point precision, leading to discrepancies between implementations results.
- **Undefined precision**: All built-in functions which do not fall in the two previous categories are considered to have an undefined precision. In particular, this applies to trigonometric functions as well as matrices functions such as determinant calculations.

Compilers are also allowed to perform optimizations such that the resulting values are "slightly different" [41] from non-optimized code. Especially, they are allowed to recompute values and to replace some operations which are equivalent in *real-life arithmetic* to benefit from hardware support. Examples of such optimizations include the replacement of multiplications by successive additions or the use of a single binary instruction to perform $a \times b + c$. Even if those operations are arithmetically well-defined, different results can be obtained due to the floating-point approximations and are therefore forbidden by the IEEE-754 norm. While traditional C compilers such as Clang and gcc do not perform such optimizations by default, both offer a `fast-math` mode to authorise such optimizations [43; 44].

3.1.7 Summary of undefined behaviours

Category	Undefined and implementation-defined behaviours
Representation of values	Binary size of values Floating-point special values
Memory access	Out-of-bound array access Unitialized variables Layout and unitialized buffers
Arithmetic	Integer-based division Negative modulo operands Out-of-bound shift operands Floating point operations
Functions	Built-in functions Floating-point based functions
Evaluation order	Function call copy-backs Initializers parameters Side-effecting expressions
Control-flow	Non-terminating loops
Race conditions	Interaction between workgroups and threads

Table 3.1: Summary of the undefined behaviours divised into categories

Most undefined behaviours of GLSL have been described in the previous subsections and are summarized in Table 3.1. Two categories of undefined behaviours are not described above: *memory accesses based* undefined behaviours and *control-flow based* undefined behaviours.

Contrary to C, GLSL does not support pointers and undefined behaviours related to memory accesses are limited to array accesses and non-initialized memory (see subsection 3.1.1). While negative array accesses are treated as errors, array accesses which are *out-of-bound* but positive are treated as undefined behaviours.

As GLSL permits loops without reachable exit conditions, compilers are authorized to deal with such programs in any way they find appropriate and results that could be collected from the execution of such shaders are completely undefined. While long-running / never ending shaders are normally killed by the OS watchdog depending on the driver, shader interruptions can have unpredictable results and side-effects on the OS.

3.2 Tested Reducers

Reducers are usually compared according to three criteria:

- the speed with which they produce reduction output.
- the size of the reduction output (either in terms of code words or lines).
- the engineering effort required to transfer them to another language.

3.2.1 Language-independent reducers: C-Reduce

C-Reduce [18; 2] is a reducer initially developed for the C language. It uses a set of transformations in multiple subsequent passes to reduce the code. While some transformations are directly related to the C language, most of them only rely on general assumptions such as having "{" as a control flow delimiter. While it is not the case for all languages (python, for example), it is general enough to be used with a wide range of languages such as GLSL without adaptation of the tool. As C-Reduce integrates different transformation types, one transformation might unlock extra reduction opportunities in the subsequent passes. Therefore, as with most reducers, C-Reduce repeats its transformation process as long as a *fix-point* has not been found.

To use C-Reduce, it is necessary to produce an interestingness test, in the form of a hard-coded shell script that performs the compilation of the original code, as well as reduced versions (given that for all reduction attempts, the name of the code file will be identical). C-Reduce uses a temp directory to build every reduction attempt. The shell script needs to perform all sanity checks which are suitable and compile the code. As the reducer is independent of the chosen language, it does not deal by itself with undefined behaviour. Moreover, as it does not assume much on the structure of the underlying language, C-Reduce performs transformations that might produce invalid code, such that the reduction attempt is not recognized as being part of the programming language.

3.2.2 Grammar-aware reducers: PERSES

PERSES [3] is a grammar-aware reducer. While C-Reduce can perform transformations that produce invalid code, PERSES uses the user-provided *context-free grammar* to ensure that all reduction attempts are well recognized as program codes. It does not ensure that the code will compile, as many errors are related to the context in which instructions are written. However, by avoiding generating improper code, PERSES diminishes the number of reduction attempts, which increases reduction speed. While the grammar is effectively dependent on the considered language, it is usually easy to find a grammar recognizing a given language, either from an open-source project, the specification files or one developed for a random program generator.

Once PERSES has been compiled with the necessary grammar, it can use the same files as for C-Reduce to perform reduction (hard-coded shell script and original code). While C-Reduce uses a `/tmp` directory, PERSES build a temporary directory in the current working environment and then isolates every reduction attempt in an inner directory. As PERSES has no further knowledge about the programming language than its grammar, the reduction can introduce undefined behaviours.

3.2.3 Language-aware reducers: Glsl-reduce

Glsl-reduce [23] is the reducer developed by the GraphicsFuzz project, it targets undefined behaviours and possesses two modes of action. In the first mode, which is directly related to the way the GraphicsFuzz project generates new shaders, the reducer tries to remove transformations applied to the original shader to come up with the closest shader which triggers the bug as described in subsection 2.2.3. In the second mode of reduction, the reducer applies transformations directly on the parsed *abstract syntax tree* (AST) to remove or rearrange nodes. It takes care of some undefined behaviours which can be introduced, especially *array in-bounding* and *loops limiting*. Glsl-reduce can perform such transformations only because it is aware of the GLSL language. Porting it to other languages would require considerable engineering efforts.

While the first reduction mode is of no interest for GLSLsmith, the second gives a good example of a language-aware reducer. To use the second more, it is necessary to produce the shader code, an interestingness test (which admits the shader name as an argument, contrary to C-Reduce and PERSES) and a JSON file of the name of the shader code.

3.3 Compilers under test

To simplify the buffer collection for differential testing, all tests have been executed on the same platform, a personal laptop running Ubuntu 21.04 with a discrete graphics card NVIDIA GeForce GTX 1050 Mobile TI. Thanks to ANGLE and the MESA project which develop open-source drivers, it has been possible to test five different compilers.

While most differential test settings assume independent compilers, it is not the case in that test configuration, therefore identical bugs can be triggered across multiple implementations through a common dependency. Known dependencies are summarized in Table 3.2 (as NVIDIA compilers are not open-source, the dependencies are assumed).

GLSLsmith has been used in conjunctions with 5 different compilers: 2 OpenGL API compilers and 3 Vulkan API compilers (through ANGLE):

- **OpenGL NVIDIA driver** (*linux display driver v460.80 - v470.57*): A proprietary

NVIDIA driver used with an NVIDIA GeForce GTX 1050Ti GPU for the OpenGL API [45].

- **LLVMpipe driver** (*Mesa3d 21.1.1- 21.1.7*): A software (CPU) driver developed as part of the Mesa3D project for the OpenGL API[46].
- **ANGLE / Swiftshader driver** (SWANGLE) (*latest github available version*): A software implementation developed by Google for the Vulkan API [47].
- **ANGLE / Vulkan-Intel MESA driver** (*Mesa3d 21.1.1- 21.1.7*): A driver developed as part of the Mesa3D project for the OpenGL API on combined Intel Graphics Card [48].
- **ANGLE / Vulkan-NVIDIA driver** (*linux display driver v460.80 - v470.57*): A proprietary NVIDIA driver used with an NVIDIA GeForce GTX 1050Ti GPU for the Vulkan API [45].

Compiler	NVIDIA	Mesa code base	ANGLE / GLSLang
OpenGL NVIDIA	✓		
LLVMpipe		✓	
Swiftshader			✓
Vulkan-Intel		✓	✓
Vulkan-NVIDIA	✓		✓

Table 3.2: Summary of the known compiler dependencies

Chapter 4

GLSLsmith Framework

4.1 General organisation

The workflow of GLSLsmith can be decomposed into three steps. First, the generator creates a shader and dumps it into a ShaderTrap test file. Then, the ShaderTrap is post-processed to remove undefined behaviours and results are compared and categorized. Finally, if the shader is of interest, the code is reduced. In this case, an interestingness test script is created, and the shader code is split from the ShaderTrap harness. Reducer attempts reduction on the extracted shader. Then, the interestingness test merges back the shader code with the ShaderTrap harness and the resulting ShaderTrap code is sent back to execution.

From a technical point of view, the code is divided into three different parts: the *generator*, the *post-processing unit* and the *execution / reduction supervision*. The generator and the post-processing unit are written in Java, while the supervision is ensured by a set of python scripts. This division does not match the steps of the workflow, so that the post-processing unit can be used independently from the rest of the framework.

The **generator**, described in section 4.2, is in charge of the shader generation and is usually used in batch mode. It generates self-contained ShaderTrap test files. Shader generated at this stage can contain undefined behaviours. The ShaderTrap code contains the final values which will be passed to the rest of the process. After generation, the shader is sent to post-processing.

The **post-processing unit**, introduced in section 4.3, can extract information from the ShaderTrap code and replace the shader after the suppression of undefined behaviours. No other information than the GLSL and ShaderTrap code is provided to the post-processing unit. As such, it does not assume anything on the provenance of the code and can be used with any shader meeting the criteria for post-processing (no name clashes with functions and variables and special considerations for floating points as describe in subsection 4.2.4). The resulting ShaderTrap test produced by the post-processing unit is free of errors and undefined behaviours. It can be executed through ShaderTrap.

The **supervision scripts**, presented in section 4.4, control the bug-finding process and ensure the automatic generation, execution and reduction of test cases. They provide a set of tools that can be used independently or as part of the complete automatic process. Especially, it is possible to use the execution supervision scripts to perform efficient manual reduction with or without post-processing. To perform automatic reduction, the supervision script can extract the shader from the ShaderTrap testing code and build a relevant interestingness test for the reducer. When a potentially reduced file is produced, the execution framework reassembles the shader with the *test harness*, enforces post-processing and categorizes the results.

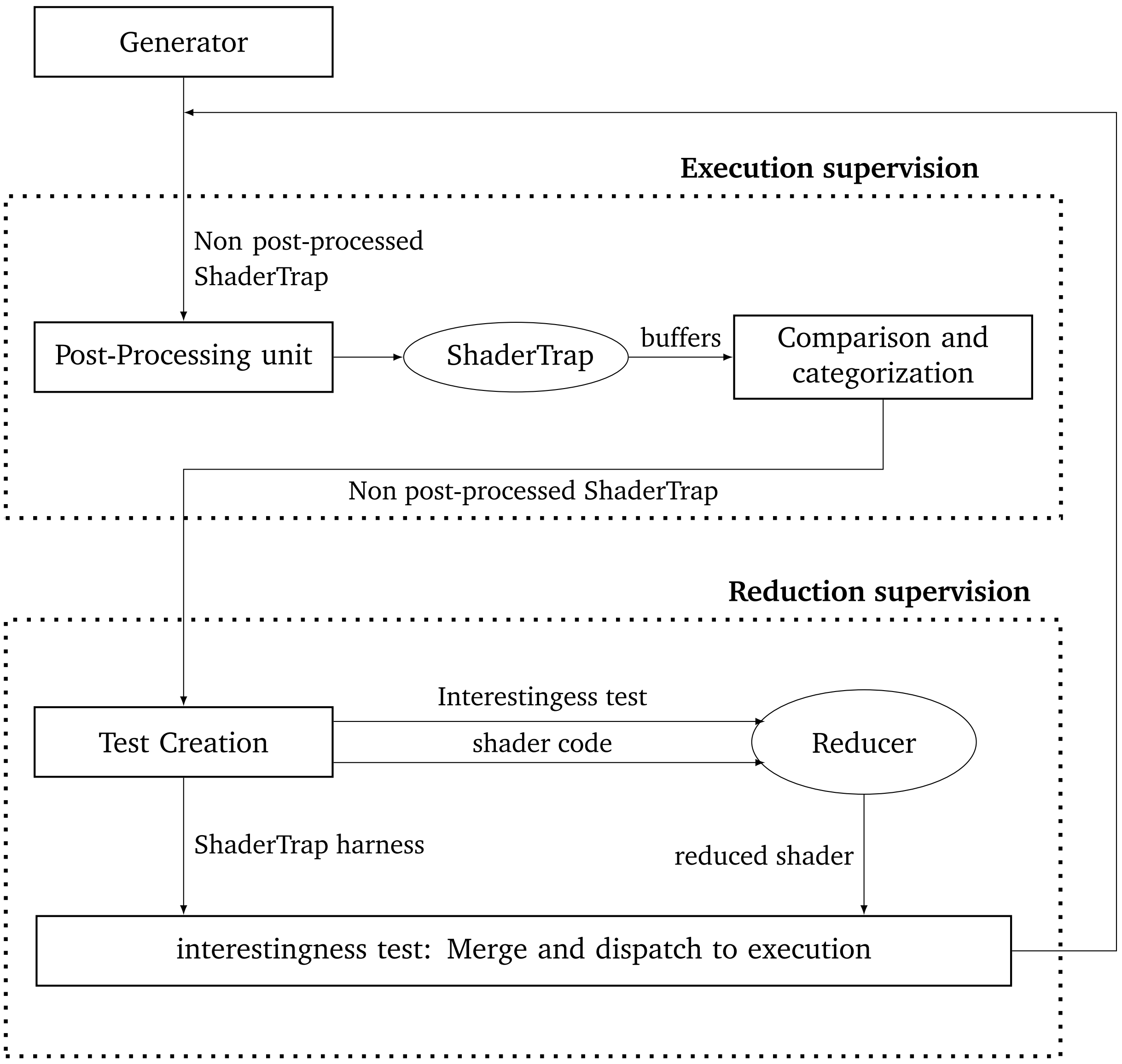

Figure 4.1: General overview of the GLSLsmith framework

4.2 Generator

4.2.1 Generation process

The generator first generates the shader in an *abstract syntax tree* (AST) representation in which every element of the shader code is represented by a *token* linked to its outer element. The shader is then printed to a text format and the ShaderTrap harness code is printed. As the shader generation is independent of the scripting, it would be possible to add support for other scripting languages besides ShaderTrap such as VkRunner or AmberScript as described in subsection 2.3.3.

To focus the engineering effort on the generation of diverse shaders, the generator and the post-processing are built as submodules of the open-source Graphicsfuzz framework [4]. Both reuse the AST format, as well as the shader parsing capabilities of the framework.

To generate a shader, GLSLsmith uses a set of recursive functions which generates the communication interfaces as well the body of the main function. The *program state* is updated through the complete generation such that recursive functions only build valid code without the need for back-tracking.

While the generation of the shader code is ensured by a single `ShaderGenerator` class, all choices of random types and operations are delegated to a secondary class `RandomTypeGenerator` to enable unit-testing.

As the suppression of undefined behaviours is delayed to the post-processing unit, shaders emitted by the generator are free to contain undefined behaviours and errors. Especially, array indexes are not required to be in-bound and can even be negative. Attempts to execute the resulting code without post-processing may therefore lead to compile-time errors when the array accesses are folded.

4.2.2 Integers and Boolean support

The generator supports the full range of operations (including ternary operations) for signed, unsigned integers and boolean values. Especially, it supports vector / scalar operations as well as comparison operations. Through generation, operand types are chosen at random from the available types on a given operation.

There is full support for vector swizzles (as well as with indexes) so that vector components can be used in expressions. As permitted per the standard, multiple swizzles can be written in a single variable access, either to obtain a smaller vector or a scalar. In the case where the original vector value is only read, resulting swizzles are permitted to contain duplicate: the resulting vector can be of a bigger size than the original vector as shown in Figure 4.2. When the swizzle is a *l-value*, the size of the swizzles is however constrained by the original size of the vector: all can be at most the same size as the former one.

```
int x = ((var_0).ts).gr.s;
bool y = (var_4).yx.rg.st[1]);
uvec4 z = (((var_3[5].rg).xxyx).ttts);
(w[ext_1].rgb).xz = bvec2(false);
```

Figure 4.2: Examples of swizzles generated by the generator. The first two swizzles resulting in x and y are example of generation for scalar values. The y swizzle uses an array index as described in subsection 3.1.2 to access the first element of the last resulting swizzle. The two following swizzles or z and w demonstrate the difference between read and written swizzles: z is built as an expanded vector from 2 to 4, while the swizzle from w is reduced to 2.

More generally, if across generation, the left operand is an l-value for which writing is permitted, side-effecting operations can be generated. L-values include variables, array values as well as vectors, and swizzles as described above.

Function calls that return integer or boolean values are directly generated into expressions to add more diversity and trigger optimizations. This includes both type conversions and more generally all functions that return the correct type can be generated.

4.2.3 Control-flow support

The generator supports the complete set of control-flow instructions, including if / else, switch, while, for and do-while loops. It also generate early exit instructions (`break`, and next execution `continue`).

ESSL and GLSL, in a similar fashion to C, are permissive in the construction of loops and conditional instructions. Especially, switch cases are valid even if they don't contain any cases or if they contain fall-through statements. Similarly, for-loops are not guaranteed to contain initialization, condition and loop expressions such that `for(;;)` defines a valid loop. While those structures are probably against best practices in all languages, GLSLsmith does not make such assumptions and generates the full extent of authorized structures.

While the generation of *atypical control-flow* structures can be an interesting way to find bugs, more traditional structures likely trigger specific optimizations. For that reason, GLSLsmith can also generate *guided* `switch` and `for` structures, which matches with real-world usages.

Guided switch structures include a random number of consecutive value cases as well as a `default` case which is added at random. To ensure that one of the `case` will be executed, the offset of the first value is added to the switch expression and the value of the expression is evaluated modulo the number of cases. In rare cases and due to the definition of the `abs` function, the actual value is undefined be-

```
// Example of guided switch generation
 switch(1u + (var_3 % 2u))
  {
   case 1u:
   {
    const uvec4 var_0 = uvec4(15u, 1u, 3u, 1u);
   }
   break;
   case 2u:
   {
    ext_8[5] %= ((! (true)) ? ~ 5 : 2]));
   }
   case 3u:
   {
    var_1.z.r += (var_0[5] ? (var_2[ext_7]) -- : float((ext_5)));
   }
   break;
  }
```

Figure 4.3: Example of guided control-flow generation.
Inner expressions of the cases and the loop have been cut for simplicity.

fore post-processing (modulo operations are only defined for two positive numbers and `abs(MIN_INT = MIN_INT)`). While the case statements are likely to be found in a real-world application, the format of the switch expression is largely constrained, as the modulo operation will be changed to a wrapper by the post-processing step (see subsection 4.3.1. An example of switch operation obtained with the guided switch generation is given in Figure 4.3.

```
// Example of guided for-loop generation
for(int for_var_2 = 1; (for_var_2 >= - 17) ; for_var_2 -= 2)
  {
   ext_3[+ (+ ++ ext_6)] = 1u;
  }
```

Figure 4.4: Example of guided control-flow generation.
Inner expressions of the cases and the loop have been cut for simplicity.

Guided for structures include the definition of an induction variable declaration, a classic comparison to a fixed value and an increment expression. For each guided for-loop generation, a random increment expression is chosen between addition, subtraction, multiplication and division, a number of steps is chosen, and the upper or lower bound of the comparison expression is computed at the level of the generator. The resulting expressions, therefore, only contain the variable declaration and numerical constants. While it does not cover perfectly all the cases found in real-world applications (where the upper or lower bound can be another variable), such for loops have a usual appearance, one of the generated loop is given in Figure 4.4.

4.2.4 Floating-point support

As described in subsection 3.1.6, floating-point testing is a difficult subject due to the high number of undefined behaviours and the relative imprecision of operations. In order to offer exact comparisons between floating-point values, GLSLsmith does not generate divisions (excluding undefined behaviours related to `NaN`). It limits its support to operations and functions for which the only error can come from round-off by selecting specific literals to generate. As the chosen set of literals is directly determined by the IEEE-754 representation (32 bits, single-precision floating-point) [42], a brief description of the format is necessary.

All values are represented on 32 bits divided as follow:

- **1 sign bit**: 0 for positive values, 1 for negative values
- **8 exponent bits**: an 8-bit unsigned number from 0 to 255, its value is interpreted as the exponent of 2 with which the mantissa will be multiplied (after subtraction of a bias of 127). 0 and 255 are reserved for special numbers, giving powers of 2 between -126 and 127.
- **23 mantissa bits**: the 23 first bits of the result of the division by the exponent of the number (excluding the leading 1).

While only 23 bits of the mantissa are stored, it effectively brings 24 bits of precision on any value which admits a floating-point representation as:

- For a value X, the exponent E is chosen as: $2^E < X < 2^{E+1}$.
- The result of the division of X by 2^E will always be $X/2^E = 1.b_1...b_{23}\ b_{24}....$
- The 23 first bits $b_1...b_{23}$ are stored within the mantissa.

As the generator controls which floating-point literals appear in a program, it is possible to only generate literals that admit an exact floating-point representation. One way to do so is by arbitrarily fixing the power of two that the last bit of the mantissa represents, written r in the following. Once that power of two is fixed, all numbers which are non-zero multiplicative of r and within range of $]-2^{24} \times r, 2^{24} \times r[$ admit a unique and exact floating-point representation.

Moreover, the operations performed on such values benefit from some stability (named *pseudo-stability* in the following): the product, the addition and subtraction of two numbers are guaranteed to be themselves multiples of r in real arithmetic and are likely to admit a unique and exact floating-pointing representation: it is sufficient to check that the result of the operation is within the range of $]-2^{24} \times r, 2^{24} \times r[$ and different from 0 (the value 0 can arise as a result of addition and subtraction). While there is no guarantee that the floating-point arithmetic will perform the same operation as in real arithmetic, the operation should not result in a rounding error.

The case of the 0 value is special: it does not admit a representation with the same format as the other powers of r but instead admits two representations: $+0$ defined as all 32 bits set to 0 and -0 defined as all bits to 0 except the sign bit set to 1. The most conservative approach is to exclude is to exclude 0 from the *pseudo-stable* set of values generated. The second is to admit both -0 and $+0$ in the set, given that the new extended set is still *pseudo-stable* by addition, subtraction and multiplication. The GLSL and ESSL specifications permit to exchange -0 and $+0$ for optimization reasons. While it does not impact subtraction, addition and multiplication as all other values are guaranteed to be finite, comparing the two forms of 0 might lead to discrepancies between comparisons results and they should be considered as identical in buffer outputs.

Pseudo-stability is well-defined for any chosen r such that $r \times 2^{24} < 2^{127}$ which leads when adding the minimum constraint to $2^{-126} < r < 2^{103}$. Interestingly, if $r = 2^0 = 1$ is chosen, it gives all integers with absolute value of less than 2^{24}, using values in the same range as the one used by [1] to avoid rounding errors (see subsection 2.1.2) excluding however the case of 0 which is relative to the specific undefined behaviours of GLSL. The choice of $r = 1$ presents two main advantages for GLSLsmith: random floating-points can be generated directly from the random integer generator, and as unary increments and decrements are defined as “adding or subtracting 1.0”, those operators can be safely generated by the shader generator. As generated operations can lead the values to escape the *pseudo-stable* range, wrappers are necessary to constraint further operations (as discussed in subsection 4.3.1).

4.2.5 Skewed random generator

While a uniform random generation can be appropriate to generate test cases, it can be expected that some values are more likely to trigger code optimizations either by their arithmetic nature, such as the integers 0 and 1 or because they represent extremum values for the fixed-size representations of the variables, either if they are directly present in the shader code or if they are computed after operations.

To test the impact of the values in the shader, GLSLsmith does not generate values uniformly for arithmetic values. Instead, it chooses uniformly between 4 sub-types of value generators for integers and unsigned integers. It is possible to enable only some of them on a specific run of the generator. Once a generator has been chosen, values are picked from it uniformly at random.

The *full-range* generator generates values in the whole range of permitted values. The range can be restrained to deal with generation constraints such as in a shift operation, in which the right operand needs to be generated between 0 and 32. In that case, the full random generator is always picked, so that values are guaranteed to be in-bound.

The *special value* generator generates values from a limited list of values which are the most probable values for wrong behaviours or different optimizations. For unsigned integer, the list is composed only of 0, 1, 2 and the maximum permitted unsigned integer ($2^{32} - 1$). For signed integers, the list is composed of 0, 1, -1, 2, -2 and the two extremum values $2^{31} - 1$ and -2^{31}. As the 2-complement representation is used, it is interesting to note that -2^{31} does not have a positive equivalent representation.

The *small-value* generator generates values in the range $[-128, 128[$ ($[0, 128[$ for unsigned integers). The bounds have been arbitrary chosen but the idea is to generate values close to 0, 1 and 2 for eventual subsequent optimization triggered after folding. In the case of unsigned integers, values close to 0 can also lead to the use of large values by the program. This is a direct consequence from the specifications of GLSL and ESSL: operations which should produce negative values are required to wrap from the highest values instead.

The *large-value* generator generates values in the range $[-2^{31}, -2^{27}[\cup[2^{27} - 1, 2^{31} - 1[$ ($[2^{28} - 1, 2^{32} - 1[$ for unsigned integers). Again, the bounds are arbitrarily chosen but values are expected to wrap from positive to negative after a few operations.

For floating-points literals and as discussed in subsection 4.2.4, GLSLsmith generates integers. While the full range of $] - 2^{24}, 2^{24}[$ could be used, the literals are generated in the range $] - 2^{12}, 2^{12}[$ in order to ensure that at least one operation will be performed before a potential overflow.

4.2.6 Function support

As many built-in functions needed to be registered for random generation, a *function registry* holds the list of available / supported functions in the given context. Supported functions represent a large subset of the built-in functions but exclude functions for which the resulting types are not well defined (`mediump` and `lowp` values) as well as floating points values for which the precision is unknown.

More precisely, all float-based functions whose definition contain an exact math operation using only addition, multiplication and subtraction are supported as their precision is inherited from those operations.

4.3 Post-processing unit

The post-processing unit uses many transformations to conservatively remove undefined behaviours. First, the ShaderTrap code is parsed into a program state format containing both the shader code and reconstructed values from the buffers. Then the shader code goes across a series of post-processing steps meeting a common interface. Each post-processing step consists of a *Visitor* which goes through the AST, identifies undefined behaviours and modify the programs to correct them. As the post-processing steps are applied multiple times across reduction, an effort is made to reduce the overhead introduced. Only the *Wrappers step* (subsection 4.3.1) and the *Parameter Order Control step* (subsection 4.3.6) type the AST.

Undefined behaviours can be decomposed into two categories depending on the situation in which they are generated. If an undefined behaviour is generated by the generator, it needs to be corrected for the shader to be executed so that every generated shader is effectively a test case. However, if an undefined behaviour is introduced at reduction time, it can either be transformed into valid code, or into statically incorrect code (see subsection 4.3.4) to avoid to type the complete AST. The invalid code is then either caught back by a subsequent step using typing or by all compilers (as a compile-time or linking-time error) and therefore is discarded.

4.3.1 Wrappers

Wrappers constitute an important part of the avoidance of undefined behaviours in GLSLsmith. They are used in three different places: *integer arithmetic-based* undefined behaviours avoidance, *built-in function parameters* undefined values avoidance and *float-based precision loss* avoidance. Wrappers can be either built as preprocessor macros or functions. GLSLsmith chooses to include wrappers as functions so that possible side-effecting operations are only effected once at function call as in Figure 4.5. All wrappers can be identified by their naming convention of "SAFE_" + OP_OR_FUNC_NAME in a similar fashion to Csmith [8]. Visitors which build wrappers work by first visiting the full program changing necessary operations to function calls and registering wrappers for later generation. Then at the end of each pass, all newly defined wrappers are added to the program.

```
// Original code
void main() {
    int x = 1;
    int y = (x++) / 2; // y = SAFE_DIV(x++, 2);
}

// Code using wrappers
int SAFE_DIV(int A, int B)
{
 return B == 0 || A == −2147483648 && B == −1 ? A / 2 : A / B;
}

void main() {
    int x = 1;
    y = SAFE_DIV(x++, 2);
}
```

Figure 4.5: Example of integer-based arithmetic wrapper
The increment operation on x is effected only once at function call. If the wrapper was to be written as a pre-processor directive, it would be effected twice, (one for each A in the SAFE_DIV wrapper.

While wrappers are generally independent of one another, some of them require modulo operations to ensure that values are taken in bounds. As modulo operations are well defined only in the case of two positive values, an extra `SAFE_ABS` wrapper is designed to deal with the MIN_INT case and enforce that values returned by `abs` are effectively positive or null. As the correct `SAFE_ABS` needs to be defined before the actual wrapper, every registered wrapper is declared first as a prototype (which enables forward declaration) and is then given an implementation.

The second special wrapper is "SAFE_FLOAT_RESULT" which wraps any non-side-effecting operation or function returning a float-based value. Contrary to the other wrappers, it does not test the value of the operand or parameters but instead, test if the value is within the range of float without a precision loss (following the *pseudo-stability* criteria as described in subsection 4.2.4) and as demonstrated in Figure 4.6.

While there are many ways to write equivalent wrappers, two general principles have been applied in the writing of all wrappers:

- Tests on values are made in ternary operators when possible such that they are likely to get inlined by the compiler
- Default code paths which are executed when the operand effectively hits forbidden values perform an operation of the same type as the one which should have been performed without undefined behaviours.

```
// Original code
void main()
{
 float x = dot(1.0, 1.0 + 1.0);
}

// Code using wrappers
float SAFE_FLOAT_RESULT(float A)
{
 return abs(A) >= 16777216.0f || abs(A) < 0.5f ? 10.0f : A;
}
void main()
{
 float x = SAFE_FLOAT_RESULT(dot(1.0, SAFE_FLOAT_RESULT(1.0+1.0)));
}
```

Figure 4.6: Example of floating-point based arithmetic wrapper
Contrary to the integer-based wrappers, the wrapper is added on the result of the + and on the result of the function call. Here the "SAFE_FLOAT_RESULT" defaults to value 10.0f.

In the case of floating-point wrappers, it is impossible to perform a default operation as this one cannot be guaranteed to be in range, each wrapper instead returns a unique value, such that the program never defaults to a succession of operations on identical constants.

While the application of each step is independent and can be made in any order, it is preferable to add wrappers last as they add a high number of new functions and functions calls which imply an overhead in subsequent typing of the AST nodes.

4.3.2 Index-bounding

Array indexes are not guaranteed to be in bounds by the generator (some might even be trivially negative). Multiple options are possible to enforce that indexes are taken in the size of an array. The post-processing unit implements two of them: *index modulo array length* and *clamping* shown in Figure 4.7.

In *index modulo array length* mode, the index-bounding step uses a modulo operation to inbound the values. As modulo operations are well-defined only in the case of two positive values, a `SAFE_ABS` wrapper is generated on the left operand. The right operand of the modulo expression is the `length` operator, defined for all arrays in GLSL (and guaranteed to be positive). It permits indexes to reach any possible value of the array with the same probability. It is the default mode used in production by the post-processing unit.

In *clamping* mode, the index values are clamped between 0 and the length of the array in a similar fashion to Glsl-reduce [23]. This strategy does not require extra function definition and only uses the built-in `clamp` function that is susceptible to be optimized out, especially as the length of the array is known in most cases at compilation time. However, as values are clamped, the first and last values of the array are extremely more likely to be used, hiding possible problems which could have arisen for inner values assignments.

```
// Original code
int x[5];
y = x[i];

// Index modulo array
int x[5];
y = x[SAFE_ABS(x) % x.length()];

// Clamping
int x[5];
y = x[clamp(i, 0, x.length()+1)];
```

Figure 4.7: In-bounding of values

4.3.3 Initialization enforcement

While generation enforces that all variables are initialized, reduction steps are likely to remove the initialization part of variable declarations. While it might not be a problem in the case of not-compiling shaders, discrepancies between uninitialized values might shift a bug to undefined behaviour. This post-processing step, therefore, initializes all uninitialized values to the same value of 1. It does not however prevent the complete removal of a useless variable. As all variables are typed at declaration, the complete necessary type for the initialization (array size if any, type of element, constructor) is evident.

This step does not deal with buffers which do not have an initialized value given by the shader but instead rely on the one provided by the testing script, those are dealt with by the communication control step (subsection 4.3.4).

4.3.4 Communication control

This step is the equivalent of the initialization enforcement step for buffers. It ensures that buffers are either removed completely by reduction or have identical values across all compilers. This process might seem identical to the initialization enforcement, it does not benefit from the same information as variables.

While the data parsed back from the ShaderTrap harness can give the type of the inner members of the buffer in most cases, arrays of size 1 have the same outer interface as the type of a single element. In that case, it is impossible to differentiate between the two options. However, as the buffers interfaces are guaranteed to be correct at generation-time, they can only be ill-defined due to a reduction attempt. Therefore, the communication control step takes the most-likely option and declares the value as an element and not an array. If the hypothesis is finally incorrect, the shader will simply not compile.

4.3.5 Loop limiters

Loops produced by the generator are unlikely to terminate by evaluating their conditional expression to false (except for guided for loops). There might terminate due to the inner `break` statements generated by the generator or they run forever. While never-ending shaders can be caught back by the supervision framework, their outputs are undefined (as described in subsection 3.1.7) and the test case benefit from the execution of as many parts of the shader code as possible. Therefore, loop limiters [23] are introduced by the post-processing unit. The loop limiters can be introduced in two different ways by post-processing, either by including a *global limiter* that allows a global budget to the shader or by including a *local limiter* per loop with a local budget. While loop limiters are simply extra counters added to loops to ensure that they exit (either by adding an extra condition to condition expression or by adding an inner conditional break), there are multiple parameters to take into consideration before writing them and which can impact the whole generation process.

With a global limiter strategy, the shader code is given a budget for the execution of all its loops. This guarantees the shader to terminate as the same budget is consumed by all loops. However, the budget will be consumed by the first non-terminating loop and all the other loops will not be executed (or only once). This problem is however nullified if all loops are guaranteed to terminate, either by design or thanks to a local limiter.

Local limiters define an identical fixed number of executions for all loops. As they are independent of one loop to another, they enable all loops to effectively have a chance to be executed. The main problem with local limiters is that multiple inner loops can increase the complexity of outer loops: while the outer loop will be executed 10 times, an inner loop will be executed 10×10 times, etc. To prevent this case from happening, GLSLsmith declares local limiter variables as global variables, therefore the inner-most loop will always have the same budget as the outer-most loop. The inner-most loop will probably consume all its budget on the first execution of its outer loop but is effectively executed.

After some tests with solely the global limiter, the global limiter and local limiters or with local limiters only, they were no difference in the number of shaders reported with either a global limiter and local limiters and with only local limiters, while they were less with only the global limiter. All but one compiler never exhibits time outs even with only local limiters and the last compiler still times out on some shaders even with global limiters. Therefore, by default, the post-processing unit only includes the local limiters.

In both strategies, there are two possible positions to increment the value of a limiter, either the first instruction of the loop or the last one. Increasing the loop limiter / breaking at the end of a loop execution presents the advantage of effectively executing the loop body at least once (when the original condition of the loop is met). However, if the loop contains a reached `continue` instruction, the loop will never be exited. Moving the increment instruction as the first one of the loop has the interest of removing this possibility but a loop might exit without being executed at all. In the case of GLSLsmith, as `continue` statements can effectively be generated, the increment is realized at the beginning of the loop execution, followed by a conditional break. The conditional break could have been removed and the conditional expression rewritten. However, it would enforce all loops to have a conditional expression, which is not one of the obligations of GLSL as described subsection 5.1.4.

4.3.6 Parameter order control

While GLSL enforces the evaluation order of functions parameters when they are copied to a function, it does not specify the order in which values should be copied back (for `inout` and `out` parameters). Similarly, operands are not guaranteed to be executed from left to right in an operation and the wording for array constructors, which are written in a similar format to function calls, is ambiguous. As the work of this step is tightly dependent on the precise rules of the evaluation order in ESSL, a summary is provided below and examples are given in Figure 4.8:

- **Function calls parameters** are all evaluated from left, to right, at call time, an undefined behaviour can only be found in `out` / `inout` parameters, as those values are required to be L-values to be copied back into, there are no possible nested function calls whose results would be themselves `out` parameters. Undefined behaviours therefore only appear if the same expression is used in different parameters.
- **Array constructors** parameters are assumed to have an undefined order of evaluation, as multidimensional arrays are permitted, an inner parameter can effectively be itself an array constructor. Undefined behaviours can happen as long as the expression is used in another parameter (even if the expression is an inner array constructor) and until the outer-most array constructor is exited.
- **Operations and operand** evaluation order is globally undefined. The left or right operand can be evaluated first except in the following cases:
  - The operation is an **assignment**, then the left operand is guaranteed to be evaluated before the right operand.
  - The operation is a **logical OR or AND**, in both cases the left operand is guaranteed to be evaluated before the right operand (it does not apply to the XOR operations).
  - The operation is a ternary operation, the condition is evaluated first and only one of the two conditional expressions is then evaluated.
  - Two operations apply to the variable identifier, then the order of precedence of the operators is used, this situation occurs for unary operations (increment, decrements and array access).

  Expressions can contain function calls, array accesses and assignments operations can be enclosed in operations for which the evaluation order is undefined.

An obvious solution to deal with those undefined behaviours is to rewrite all side-effecting operations to use temporary variables which would be affected before the execution. However, it would lead to a significant rewriting of all operations and the effective removal of all side-effecting operations. The difficulty is therefore to minimize the number of temporary variables to remove all undefined behaviours.

The post-processing unit differentiates three cases for which rewriting is necessary, evaluating members from left to right. In the case of arrays, the conservative option has been retained and the whole array is marked as read / written. While some closer bounding might be possible in many cases, it would require computing the

```
int f(out int p0, out int p1, int p2);

f(x, x, x); // undefined behaviour on the first two parameters
f(x, y, x); // no undefined behaviour
f(x[f(z, y, z)], y, a);
// no undefined behaviour, f(z, y, z) is evaluated at call time

int[3](int[2](x, y),int[2](x -= 3, z), int[2](w += 2, w));
// undefined behaviour on x and on w

p[i] = i++; // no undefined behaviour
(p[i] += 2) == 3 || (p[i] -= 2) == 4; // no undefined behaviour
++p[p[1]]; // no undefined behaviour

x ++ + ++ x; // undefined behaviour
(x += 2) + (x *= 2) // undefined behaviour
f(x, y, z) + (x *= 2) // undefined behaviour on x
(x += 2 == 3 || y *=2 == 3 ? y += 3, x +=2) * (y += 2);
// undefined behaviour on y
```

Figure 4.8: Examples of undefined-behaviours related situations

value of the array indexes. In all rewritings that the post-processing step performs, the temporary variable is set to the value of the original variable before the offending line of code:

- **Write after write**: A variable is written twice, the second variable needs to be rewritten to avoid undefined behaviour.
- **Write after read**: A variable is written after being read, the writing variable will be rewritten to avoid undefined behaviour.
- **Read after write**: A variable is read after being written, the read will be rewritten to avoid undefined behaviour. As read after read are not a concern, all reads share a default temporary variable.

Examples of the rewriting performed by the *Parameter Order Control* are given in Figure 4.9.

While all undefined behaviours from expressions, array constructors and functions are identified by keeping track of read and written variables, the conditions in which tracked variables are either added to the rewriting set or flushed from it differs:

- **Function call parameters** are added to the rewriting list at the end of the execution of the parameter containing it, they can be flushed as soon as the function call is exited. They are therefore tracked in a stack of sets.
- **Array constructors** are added to the rewriting list at the end of the execution of the parameter containing it, however, they cannot be flushed until the last array constructor is exited. They are therefore tracked in a single common set.

```
int f(out int p0, out int p1, int p2);

int x_1 = x;
f(x, x_1, x);

int x_1 = x;
int w_r = w;
int[3](int[2](x, y),int[2](x_1 -= 3, z), int[2](w += 2, w_r));

int x_1 = x;
x ++ + ++ x_1;

int x_1 = x;
(x += 2) + (x_1 *= 2);

int x_1 = x;
f(x, y, z) + (x_1 *= 2);

int y_1 = y;
(x += 2 == 3 || y *=2 == 3 ? y += 3, x +=2) * (y_1 += 2);
```

Figure 4.9: Rewriting as performed by the post-processing unit

- **Operations and operand** are the most difficult to handle. They are added to the tracking list as soon as they are met. However, when a partial evaluation order is defined (assignment for example), the set of forbidden values is guaranteed to be the one used before evaluating that sub-expression. Whenever the sub-expression is exited, all met variables are added to the forbidden set so that outer expressions have the correct set, including variable contained in the assignment.

As a variable identifier is likely to be found in multiple lines of code, all variables are ensured to be unique with an extra identifier in their name referencing the line of code (or array access number). This was however not added in the Figure 4.9 for simplicity.

4.4 Supervision framework

4.4.1 Framework overview

To produce interesting results in a meaningful way, it is necessary to coordinate the activities of all programs from the generation to the reduction of the shader code. To completely automate the process, it requires to:

1. Produce batches of random shaders (Generator).
2. Enforce the post-processing step (Post-processing unit).

3. Compile and execute shaders while dispatching them to correct back-end (ShaderTrap).
4. Assemble buffers results in a meaningful way and classify outputs.
5. Discard useless shaders, save the interesting ones.
6. Build interestingness tests for a chosen reducer.
7. Split the shader code from the ShaderTrap file.
8. Dispatch the shader code for reduction (Reducer).
9. Assemble back the reduction result and store the resulting ShaderTrap code.

While the automatic pipeline is interesting for many cases, the framework provides multiple ways to perform only certain steps, especially for development settings. For example, when new functionalities are added to the generator or the post-processing unit, it is interesting to ensure that the resulting shaders will correctly compile on a large set. Other interesting functionalities include a manual reduction helper which provides a way to execute only a subset of the tasks described above:

1. (Optionally) Enforce the post-processing step.
2. Compile and execute shaders while dispatching them to correct back-end (ShaderTrap).
3. Assemble buffers results in a meaningful way and classify outputs.

Lastly, a reporting tool permits obtaining some stats on the current execution campaign (for example, the number of shader crashing or giving a different result for a given compiler) or the size of the files. It also enables to find shaders that are likely to suffer from remaining undefined behaviours in the generator, by detecting if more than two buffer values exist or if the two groups of compilers give different results. Lastly, it can be used to report the affected compiler for a given shader code file. This is particularly useful when looking through the resulting reduced file to look for new bugs.

4.4.2 Bug categorization

As reduction is based on the reproduction of a bug with a smaller code base, the way a bug is categorized directly impacts its chance to get correctly reduced. As possible undefined behaviours are suppressed by post-processing before reduction (as described in section 4.3), the principal risk during reduction is bug slippage.

All shaders that present a potential defect exit execution with an error code of 1, as well as a generated internal 4-digit code, returned as a string. In a given installation, the first tested compiler is given the id 1, the second 2 etc. As it is impossible to know the compilers which will be stressed on a given installation, error codes are only guaranteed to designate a specific compiler (or compiler set) in a fixed installation context. A summary of the error code categories is given below:

- **1000**: All compilers failed to compile or execute the given program.
- **1000 + 1<<id**: Compiler(s) of id failed to compile or execute the given program.
- **2000**: All compilers failed to compile or execute the given program.
- **2000 + 1<<id**: compiler(s) of id timed out on the given program.
- **3000 + 1<<id**: Compiler of id gave a different result compared to all other compilers.
- **3099**: All ANGLE / Vulkan compiler gave a different result compared to all non-ANGLE compilers.
- **4001**: Two or more non-trivial groups of compilers give different results.
- **5000**: Result is different from the provided reference file.

From experiments, independent compilers are unlikely to crash or time out on the same shader. In such rare case, the resulting error code permits to identify all compilers in the set (powers of two can be seen as boolean flags). However, different output results between compilers can have multiple meanings: undefined behaviour, miscompilation of a given shader or miscompilation in a common dependency. As described in section 3.3, compilers are not independent, and in particular, all Vulkan implementations rely on ANGLE [32]. The error code **3099** therefore permits to differentiate where the possible miscompilation happens and to report potential bugs to the correct project.

It is also interesting to note that error codes **1000** and **2000** do not refer to possible compiler defects but rather to defects in the generator or the post-processing unit. They are particularly convenient in development to isolate and reproduce errors in the framework.

In a similar fashion, the last error code **5000** does not indicate that a compiler presents a different behaviour, but it can be used with the reduction framework to eliminate dead code in a given shader. The idea was that for some bugs, it might be interesting to first delete all dead code before attempting to reduce the live part of the code. It has however not been used in production (as reduced shaders have been of sufficient quality without this pass (see subsection 5.3.2).

4.4.3 Reduction supervision

While one of the difficulties of the automation was to safely automate the execution (and especially the automatic dispatching to the correct back-end), performing reduction in an autonomous mode for shaders represent multiple obstacles. As the reducer uses the interestingness test, the test needs to be sufficient to replicate a bug. Contrary to the case of the C language, where a test program is typically self-contained, the shader by itself is insufficient to be compiled. Therefore the created interestingness test needs to perform the following:

1. Check that the shader code effectively contains a main
2. Merge the given code back to the ShaderTrap harness (whose location is independent from current execution)
3. Call the execution script with the resulting shadertrap code (this is done at the position of the harness which is controlled by the script)
4. Check if the error code matches with the one given at test generation

As the interestingness test is able to call back the execution script, it does not have to deal with post-processing by itself but instead relies on the code developed to initially execute the code. While GlslangValidator (the reference front-end compiler for GLSL introduced in subsection 2.3.1) is not directly called by the interestingness test, possible invalid code is caught back through Shadertrap, which validates all shader code with GlslangValidator before compiling it. This situation is however unlikely as the wrong shader would have to go through the GlslangValidator pre-processor as well as the post-processing unit of GLSLsmith without raising an error.

Reduction supervision, therefore, requires to be able to build a shell script to call Python code from the framework itself. A second problem associated with the task is that the different reducers are not built to use the same shell script format. While they all expect the shell code to exit with an error when the bug is not reproduced, some use the hard-coded initial name of the shader as the argument of all calls, while the other will pass an extra argument with the name of the new file to compile. It was however possible to combine both approaches to build a single interestingness test which only depends on the error code passed at creation. An example of generated interestingness test is given in Figure 4.10.

As one of the objectives of GLSLsmith is to be able to compare the qualities of multiple reducers, the supervision generator can add an extra instruction to collect the number of calls made to the test within reduction (see subsection 5.3.2).

4.4.4 Testing GLSLsmith

While deterministic features of the generator can be tested in isolation, tests are difficult to write for functions which provide random results. When introducing new features, new programs generated by the generator, could be wrong GLSL code. However, the execution framework permits to perform end-to-end testing on multiple batches of shader, validating the new generation features.

In a similar way, it is impossible to test all possible combinations of expressions for post-processing. While contrary to the generator, it is possible to test each step in

```
#!/usr/bin/env bash
set -o pipefail
set -o nounset
set -o errexit
ROOT="[...]/glslsmith/prod"
ERROR_CODE="3008"
if [ $# -eq 0 ]
then
    SHADER="test.comp.glsl"
else
    SHADER_ROOT=(${1//./ })
    SHADER="${SHADER_ROOT}.comp"
fi

cat "$SHADER" | grep "main"

python3 ${ROOT}/scripts/splitter_merger.py
  --merge ${ROOT}/test.ShaderTrap "$SHADER"

ERROR_CODE_IN_FILE=$( (python3 ${ROOT}/scripts/reduction_helper.py
   --config-file ${ROOT}/scripts/config.xml
   --shader-name ${ROOT}/test.ShaderTrap 2>&1 > /dev/null) || true)

echo $ERROR_CODE_IN_FILE
if [ "$ERROR_CODE_IN_FILE" == "$ERROR_CODE" ]
then
    exit 0
else
    exit 1
fi
```

Figure 4.10: Example of interestigness test showing a possible miscompilation of Vulkan-Intel with the installation described in chapter 3. The ROOT variable is set to a valid location, truncated in the listing.

isolation on “trivial” examples, more complicated errors are caught up by the execution framework, either as shaders which cannot be post-processed or in the case of remaining undefined behaviours by the increase of errors with multiple sets of value (reported with the error code **4000**).

Chapter 5

Results

The use of the GLSLsmith framework has led to the discovery and reporting of multiple bugs in the tested compilers. After a brief description of the reporting process in subsection 5.1.1, a more detailed summary of the identified bugs is given in subsection 5.1.2 and a possible categorization of the origin of bugs is given (subsection 5.1.3 and subsection 5.1.4) along with some shader examples.

The detailed examination of the bug codes tends to prove the various design choices of GLSLsmith, including the support for guided structures (subsection 5.2.2) and the way to perform floating-point testing (subsection 5.2.3). Such results permit to discuss qualitatively the broader impact of the post-processing unit on the bug finding process subsection 5.2.4.

Finally, the interactions between the post-processing unit and the studied reducers (C-Reduce, PERSES and glsl-reduce) are studied in more details in section 5.3 and the qualities of the three reducers are compared according to multiple criteria.

5.1 Identified bugs

5.1.1 Bug reporting processes

In order for a bug to be fixed, it is necessary to report it correctly to the project owners. Once a clean reduced shader is produced and once the correct source of the discrepancy has been determined in the case of Vulkan (see subsection 4.4.3), it needs to be encapsulated in the expected scripting language used by a project: `shader_runner` for GLSL and `Vkrunner` for Vulkan to report to Mesa for example.

Then a bug report containing the shader, the testing configuration as well as an explanation of the expected results has to be filled either in a bug-tracking system (Gitlab issues for Mesa, GitHub issues for Khronos-related projects, Google Issue Tracker for ANGLE and SwiftShader) or on a forum (NVIDIA).

Reporting a bug revealed itself to be time-consuming and possibly frustrating, when no answer is given by the project owners. On all bugs reported, open-source projects are the most likely to answer. Moreover, as the code is open-source, it is possible to follow the handling of the bug and to obtain some insights of the inner working of a compiler. As an example, a bug reported to SwiftShader turned out to be caused by spirv-opt, a shader optimizer developed by Khronos for Vulkan. Thanks to that knowledge, it has been possible to differentiate between bugs triggered by SwiftShader and spirv-opt in subsequent reports (by calling shaders before and after optimization on any Vulkan back-end).

In the case of the Mesa project, which is fully open-source, bug reporting has also led to the discovery of multiple related bugs and the addition of equivalent shaders to the conformance and regression test suite to ensure that all future releases of the compilers will not exhibit such bugs.

5.1.2 Summary of identified bugs

While the duration of the bug reporting campaign has been limited by the time constraint of the project, GLSLsmith found 23 bugs believed to be unique. Among those 23 bugs, 15 have been successfully reported so far. As GLSLsmith is able to find multiple bugs in a single run, bugs reports are deliberately reported in a slower, gradual manner to ensure that the development team has the time to deal with previously reported bugs (as advised for *responsible bug reporting* [49]). A summary of the found and reported bugs is given by Table 5.1 and Table 5.2.

Compiler	Crash	Miscompilation
NVIDIA	1	8
LLVMpipe	4	2
Swiftshader	1	2
Vulkan-Intel	0	3
Vulkan-NVIDIA	0	2

Table 5.1: Summary of the bugs found by compiler

Compiler	Crash	Miscompilation
NVIDIA	0	5
LLVMpipe	4	2
Swiftshader	0	2
Vulkan-Intel	0	2
Vulkan-NVIDIA	0	0

Table 5.2: Summary of the bugs reported by compiler

During the campaign, one of the reported bugs turned out to be an undefined behaviour for which the post-processing was not efficient (this bug is not reported in Table 5.1 and Table 5.2). It lead to the expression evaluation control of the *Parameter order control* step (see subsection 4.3.6). After reporting, some of the bugs have also have been added to the regression test suite of the Mesa project ensuring that all future implementations will not exhibit such bugs. On all the fixed bugs, 8 have been fixed within 24h of being reported, suggesting that bugs found through random generation can be of importance for real-life applications. Those results are summarized in Table 5.3.

While not all bugs can be categorized, some similar bugs can be found across multiple implementations and multiple shaders have been found to trigger the same wrong code error across independent compilers.

Status of bug	Number
Added to regression suite	4
Fixed	9
Confirmed	12
Successfully Reported	15
Not Reported	8
Undefined behaviour	1

Table 5.3: Status of found bugs (Tests added to regression have been fixed and therefore confirmed after report).

5.1.3 Wrong integer optimizations

Bugs that fall into that categories are related to signed and unsigned integer arithmetic optimizations (5 have been identified so far). Especially, many of the reported bugs were triggered by the folding of division operations combined with negative values. Discrepancies between *mathematical integer arithmetic* and *machine integer arithmetic* (unsigned integers are required to wrap when they overflow / underflow on the minus operator, `-MIN_INT=MIN_INT`) are responsible for 4 of the bugs. The last one, presented in Figure 5.2 was affecting a large number of division operations on a platform.

```
void main()
{
 ext_2 /=  -(1 + 1 / ext_1); // ext_2 = 4 and ext_1 = 2
}
```

Figure 5.1: Shader 9 affecting the Vulkan-Intel compiler
While the code should return `-ext_2` , Vulkan-Intel returned `ext_2`. The bug was due to an incorrect optimization on both division and modulo operations. It has been quickly confirmed with the comment “wow, just wow”, fixed and added to the regression tests.

5.1.4 Atypical control-flow statements

As GLSL is permissive with control-flow statements, the generator found 5 bugs in atypical control-flow instructions. While some of them are unlikely to be found into production code, they might appear after conversion from one framework to another or when a developer tries to find a bug in its own code and comments out unnecessary instructions. The 2 found miscompilations could also be executed by real-world application after optimization (or to compress the size of the shader code). The 3 other bugs, all crashes have been found with LLVMpipe and are all-related to variable shadowing into for and do-while loops.

```
void main()
{
  ext_0 += ext_1; // ext_0 = 5
  switch(ext_1 ^= ext_0) // ext_0 = 5 ^ 5 = 0
  {
    case 0u:
      ext_2 = 1u;
      break;
    case 5u:
      ext_2 = 2u;
      break;
  }
}
```

Figure 5.2: Shader 4 affecting the LLVMpipe compiler
While the case 0u should be executed, LLVMpipe instead executed the 5u case. The side-effect in the switch operation was in fact performed twice. The bug was confirmed with the comment:"This bug (double evaluation of switch expression) affects all mesa drivers. It's amazing that it survived for so long (looks like it existed at least since 2011)", quickly fixed and an adapted shader has been added to the regression test suite.

5.2 Justification of the design choices

5.2.1 Skewed random values

As described in subsection 4.2.5, the generator does not generate random literals uniformly but instead combines multiple random generators. While the impact of the small and large values generator has not been demonstrated yet, multiple special values identified in the conception such as 0, 1 and the MIN_INT have been found in the reported shaders. Even if those values have possibly been generated by the uniform distribution, operations using the MIN_INT are responsible for 4 bugs and 1 and -1 are both partly responsible for 2 bugs. This strongly supports the idea that the special value generator is of interest of bug finding.

After reporting, it has been found that MIN_INT issues were partly related to the incorrect assumptions made on integer arithmetic such as `abs` returning a positive value (as shown in Figure 5.3) or -MIN_INT being different from MIN_INT.

```
int SAFE_ABS(int A)
{
 return A == -2147483648 ? 2147483647 : abs(A);
}

void main()
{
 ext_8 = SAFE_ABS(abs(ext_7));
 // ext_7 = -2147483648, ext_8 = 2147483647
}
```

Figure 5.3: Shader 21 affecting the Vulkan-Intel compiler
Optimizations wrongly compile the test in the SAFE_ABS function due to call in the parameter to the built-in `abs`. The issue has been confirmed but not yet fixed.

5.2.2 Guided structures

As described in subsection 4.2.3, the generator generates both guided `switch` and `for` control-flow statements. While guided `switch` statements have not yet permitted to find unique new bugs (some duplicate have been found), one bug presented in Figure 5.4 have been found to be only generated thanks to guided `for` loops. It proves the interest of the generation of such features. Moreover, as such bug could be found in applications, they are likely to be of importance.

```
void main()
{
 // ext_3 = 0;
 for(uint i = 12u; i > 3u; i -= 3u)
  {
   ext_3 += 1u;
  }
 // ext_3 = 3;
}
```

Figure 5.4: Bug 24 affecting the NVIDIA compiler
The initial value of ext_3 is set to 0 by a buffer (not represented here). As the value of i is decremented by 3 at each iteration, the inner loop should be executed 3 times (for i=12u, 9u and 6u). However, after execution, NVIDIA reports a value of 2. The bug has been reported but not yet confirmed.

5.2.3 Floating-point results

While floating-point support has been exploited only at the end of the project and did not benefit from the same testing time, early results are encouraging: 3 bugs have been found relying on floating-points. Among the 3 bugs found, one is possible to reproduce without relying on floating-point values, leaving 2 bugs directly related to floating-points. The two bugs have not been reported yet, however, as both cases are independent from the precision of floating-points operations, they are likely to be correct (in the first case, a correct shader does not compile on a platform, in the second case, different buffer values are affected by an operation).

Another shader found by differential testing has turned to be a discrepancy between -0 and +0 in the buffer results. In that specific case, the two implementations could have been considered equal with a more elaborate comparison function on the buffer results. It tends however to confirm that inclusion of the 0 in a relaxed *pseudo-stable* range (see subsection 4.2.4) may lead to additional undefined behaviours which could be difficult to catch such as direct comparisons within the shader between -0 and +0.

5.2.4 Post-processing insights

While the first purpose of post-processing is to remove undefined behaviours, it effectively introduce patterns in the generated code. If such patterns can be detrimental for the full explorations of optimizations (as described in subsection 2.2.1 with Csmith), some of the bugs reported has been found to be generated only thanks to post-processing (as the example given in Figure 5.3). Especially, the post-processing unit can generate structures that are not supported at a given time by the generator, guiding the addition of features. For example, the support for arrays has been extended in the generator after the discovery of a bug triggered by the rewriting of an undefined behaviour in an expression by the Parameter Order Control (subsection 4.3.6).

Another example of introduced behaviour is the generation of user-defined function calls, even if the generator does not build function definitions. Through the different manual reductions performed, many bugs have been found to disappear when the user-defined function call was removed. One explanation for such behaviour is that the compiler is able to call built-in functions though compilation (as required for the support of constant values) which lead to possible optimizations that the compiler cannot perform with user-defined function calls.

These results, as well as the bugs found thanks to the guided control-flow structures support the idea that the introduction of repetitive pieces of code into generated code may expand the code coverage of a given generator. Those contrast with the findings of `CsmithEdge` [25] and might indicate that generators which would randomly select between systematic wrapper generation and relaxed wrapper generation (either by static or dynamic analysis) might achieve a better overall language coverage.

5.3 Reducers comparisons

5.3.1 Impact of the post-processing on reduction

As the reducer is unaware of the post-processing step, it can drastically modify the impact of the considered reductions: a reduction attempt from the reducer may lead to changes in the non post-processed code which is then canceled by the undefined behaviour removal. On the other hand, the suppression of certain tokens by the reducer can lead to a significant decrease in the post-processed version, for example because a wrapper is not generated or because it eliminates a complete buffer block (as buffers which are not completely eliminated are regenerated by the Communication Control step described in subsection 4.3.4).

As expected, it is necessary to add a final step of reduction at the end to eliminate unnecessary wrappers, compress buffers blocks when possible and eliminate unnecessary ShaderTrap instructions (such as declaring unused buffer or dumping buffers which do not carry information for the given test). While this final step is not automated, it is largely simplified as the resulting shaders are only a few lines long. Interestingly, it is sometimes easier to perform part of this reduction before post-processing to have a better reading of the program (for example converting the test expression of a while loop to a a conditional if expression removes loop limiters as introduced in subsection 4.3.5).

A second unexpected and potentially more detrimental side-effect of post-processing is that reduction steps which do not lead to the suppression of any token in the post-processed version can mislead the reducer. As the reducer thinks that a better code has been found, it might restart a full series of reduction attempts, in a process of *over-reduction* while the program is already 1-minimal with regards to the undefined behaviours suppression, possibly delaying the results.

5.3.2 Benchmark presentation

The quality of the three reducers (C-Reduce, PERSES and glsl-reduce as described in section 3.2) has been evaluated on a set of 22 shaders exhibiting the different bugs found by GLSLsmith. The initial sizes of the programs are between 28 and 408 lines (mean 179 lines). The shaders reproduce 13 of the 15 reported bugs. The other shaders are either possible duplicates which have been reduced during the campaign or bugs which have not yet been reported. The distribution of the reduced shaders classed by compilers and between crash and miscompilation is given in Table 5.4. The two other reported bugs are not reproducible with current compiler versions as a fix has been deployed.

The benchmark has been used to evaluate two different characteristics of the reducers: their efficiency and the quality of the outputs [3]. To evaluate the efficiency, two metrics have been used: the duration of the reduction, which gives a practical measurement but is hardware-dependent and the number of calls to the interestig-

Compiler	Crash	Miscompilation
NVIDIA	1	6
LLVMpipe	3	2
Swiftshader	0	1
Vulkan-Intel	0	9
Vulkan-NVIDIA	0	1

Table 5.4: Distribution of the shaders per compiler

ness test. The number of calls depends only on the reducer characteristics and is therefore hardware independent. Multiple metrics have been proposed to evaluate the quality of a reduced shader such the number of lines [20], the number of tokens [20; 3], the number of bytes / characters [2]. For this benchmark, the number of lines and the number of characters have been reported. As the reduction output is not the final state of the evaluated shader, values have been computed on the post-processed version of the shaders. As the final step of reduction likely will have to remove unnecessary wrappers, the number of wrapper calls has been also reported. C-Reduce and PERSES support multi-threading, however, as the test is not self-contained, race conditions might occur in the common execution directory and multi-threading has not been enabled.

5.3.3 Benchmark results

After the execution of the benchmark, all reducers manage to provide smaller shaders which are free of undefined behaviours and which are easy to analyze for the final manual reduction step. As shown in Table 5.5, glsl-reduce returns the longest shaders of the three reducers. Looking more closely at the reduced shaders, this can partially be explained by the fact that glsl-reduce does not attempt any reduction on the buffers. On the contrary, both PERSES and C-Reduce aggressively reduce the buffers. While necessary buffers are regenerated by the communication control step of post-processing (subsection 3.1.1), useless ones can be safely eliminated.

Even if the number of arithmetic operations within reduced shaders is low, some still require multiple wrappers to be fully defined. In particular, shaders which contain multiple array indexes generates calls to the SAFE_ABS wrapper.

If all reducers can be considered of good quality, C-Reduce happens to be inefficient when compared with glsl-reduce and PERSES and is unpractical to use in production settings. On the contrary, both Glsl-reduce and PERSES seem to be strong candidates to perform reduction within a continuous setting of generation / reduction. Glsl-reduce is the fastest of the three reducers, which was expected as it is a glsl-specific reducer. It generates two times less test calls as PERSES. The number of test calls could be critical to reduce shaders which time out. As no sanitizer exists

		Minimum	Maximum	Mean	Median
C-Reduce	Test calls	2703	21368	7807	6700
	Reduction time (h:m:s)	2:10:50	20:15:20	6:42:59	5:30:04
	Line count	11	51	24.3	23
	Character count	202	1484	616.2	516
	Wrapper calls	1	16	3.6	3
PERSES	Test calls	94	462	209.5	174
	Reduction time (h:m:s)	0:04:00	0:29:33	0:12:07	0:09:44
	Line count	12	59	26.6	24
	Character count	259	1431	620.1	497
	Wrapper calls	0	8	2.7	2
glsl-reduce	Test calls	31	274	96	82
	Reduction time (h:m:s)	0:02:00	0:21:03	0:07:23	0:05:57
	Line count	28	70	37.1	36.5
	Character count	464	1993	1049	946
	Wrapper calls	0	13	4	2.5

Table 5.5: Summary of the results per tested reducer

for GLSL, it is impossible to compare directly the efficiency of a reducer with and without post-processing. However, a previous evaluation of PERSES with C bugs [3] reports a mean reduction time of around 30 minutes for programs which are twice as long as the examined shaders. It is therefore reasonable to consider that the post-processing slows down the reduction process.

While timing evaluations are provided for C-Reduce [2], they only apply to C programs for which C-Reduce has more transformations. By examining the proposed shaders across reduction as well as the logs, conjectures can be made to explain the poor efficiency of C-Reduce. As it uses a fix-point research to decide if a program is completely reduced, it can be anticipated that even small suppression in a pass trigger a complete new pass of attempted transformations, even when the suppression was meaningless. Two origins can be suggested for those deletions. A first source can be GLSL in itself, as it supports shortened expressions when the values of some parameters are obvious. Such shortened expressions include the omission of the size of an array upon declaration when the size is obvious (initializer size, affectation from another array) or the local size of the compute shader which can be omitted on two directions. A second source could be post-processing: by removing or renaming buffer elements, the reducer can think that it produced a smaller shader. Such transformations introduce undefined behaviours, are therefore eliminated, and the final shader exhibits the same properties as before. As the reducer is blind to the post-processing, it will select the new shader if it exhibits the wanted property and will launch a new cycle of reduction.

Chapter 6

Conclusion

6.1 Achievements

GLSLsmith has permitted the identification and the report of 15 bugs so far, and may lead to the possible discovery of new bugs in the upcoming weeks. The design of its automation framework also enables to target other local compilers without adding extra code. While GLSL and Vulkan have been tested extensively thanks to metamorphic testing, GLSLsmith is the first framework enabling the generation of random shaders from scratch and the differential-testing of graphic compilers working, in a context where compilers are not fully independent.

The generator of GLSLsmith benefits from the wide coverage of GLSL features it supports, including vector types, buffers and the complete set of control-flow instructions. Other features, such as the support for floating-point testing, the generation of guided structures or the use of skewed random values generators may be of interest for the testing of other languages. Especially, GLSLsmith builds its support for floating-point testing in a challenging environment where the constraints of the IEEE specifications are partly relaxed by controlling the choice of floating-point literals and ensuring that operations never escape the *pseudo-stable* range where no round-off errors occur.

While the mechanisms by which GLSLsmith ensures the absence of undefined behaviours in programs have been used since Csmith (for example the addition of wrappers to perform safe arithmetic operations), GLSLsmith applies them in a new way, delaying the suppression of undefined behaviour to a "just-in-time" post-processing step. This method enables to reduce the engineering effort required to use a reducer without undefined behaviours. It confirmed the interest of the PERSES reducer to port reduction to multiple languages and while the use of C-Reduce is slow compared to PERSES and glsl-reduce, it can still be used as a last-resort reducer.

6.2 Limitations

GLSLsmith has been successful in its initial objective of finding and reporting bugs. However, the generator still lacks multiple GLSL features of interest which would enable a greater diversity of programs. Especially, the generator is unable to generate function definitions and function calls are therefore restricted to the built-in functions. Matrices which are key features of the floating-point of GLSL are also not included in the test. As the testing framework has been only used in the context of ESSL shader generation, certain features of GLSL are not supported such as implicit type conversions. Moreover, as GLSLsmith uses compute shaders to get value outputs instead of pictures, the testing setting is different from the classic graphics pipeline. The framework cannot be used to find bugs happening through the conversion to pictures.

While floating-point testing has given some interesting early results, it is impossible to fully confirm if this will lead to the same number of bug findings as for integer programs or if the numerous constraints on literal generations and wrappers will prevent the shader to exhibit other bugs. If the generation of integer-based values is completely free of undefined behaviours, the two representations of 0 in floating-points creates the risk of false-positives and the suppression of 0 from the supported range of values is a blocking point to implement matrix support in the generator (matrices constructors generate diagonal matrices when used with a single value).

Lastly, while post-processing have been used to successfully suppress undefined behaviours, GLSL does not support pointers, largely limiting memory-based undefined behaviours. It is expected that porting the same type of implementation to other languages such as C would bring additional difficulties (or might even be impossible). Moreover, the efficiency of C-Reduce seems to be greatly diminished compared to the other reducers and performances announced for the C language. As GLSL does not benefit from sanitizers, it has been impossible to compare if the problem comes from the post-processing step or of C-Reduce used with GLSl but it might indicate that post-processing might be detrimental in the case of fully generic reducers.

6.3 Future works

Expanding the support by the generator of the specification features, it would be interesting to ensure that the generator works for GLSL. More generally, GLSLsmith would benefit from a larger support of platforms including Android devices and the support for WebGL technologies. In particular, it could lead to the discovery of security vulnerabilities which have a broader interest for compiler development teams.

To expand floating-point testing, multiple approaches would be of interest. First, it would be interesting to compare if the number of shaders exhibiting different values evolves by changing the value of r for the *pseudo-stable* range. To resolve the problem of the two representations of 0 being inter-exchangeable, wrappers could be added to problematic operations (such as comparisons). While directly testing for 0 values in the wrappers is expected to be useless, checking if the absolute value of the variable is close to 0 (`abs(x) < 0.5f`) should work.

As wrappers have been found to prevent optimizations and hide bugs in the case of Csmith, the implementation of static analysis into the post-processing unit to identify trivially useless wrappers would benefit both program diversity and the final manual reduction step. Dynamic analysis as proposed in CsmithEdge is also expected to be of interest especially in the case of floating-point where numerous wrappers are needed.

Lastly, while no mitigation measures have been taken during the project to reduce the side-effects of post-processing with reducers and with C-Reduce in particular, two possible solutions would be to either early-stop reduction when the program size decreases to a certain point (from the different experiments, 95% of the initial size seems to be a correct threshold) or after a set running time. A second approach would be to cache the answer of the interestigness test, either signaling the identical produced shader as non-interesting or to reduce the overhead due to the compilation and execution.

6.4 Ethical considerations

Previous fuzzing projects applied to graphics compilers have identified security vulnerabilities, especially in the case of WebGL. GLSLsmith has nevertheless not identified any security-related bug and does not test WebGL. By being available as an open-source project, GLSLsmith can be used by anyone, including potential malicious users. However, it also has the potential to help development teams to identify such pre-existing bugs, ensuring the reliability of the compilers, in an equal measure.

Except potential cybersecurity risks, random program generators can uncover an important number of bugs in a short period of time. One might want to report as many as possible either to prove the interest of generator or to increase the reliability of the compilers. However, a considerate and more ethical approach is to file bug reports at a slower pace to ensure that programmers do not feel pressured or criticized for their work. The priority should always to be useful to the compiler development, especially in the case of open-source projects. As described in subsection 5.1.2, this second approach has been followed throughout the project as it is believed to be the best course of action in the long term.

Bibliography

[1] Nagai E, Hashimoto A, Ishiura N. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions;7:91–100. pages 2, 4, 6, 8, 31

[2] Regehr J, Chen Y, Cuoq P, Eide E, Ellison C, Yang X. Test-Case Reduction for C Compiler Bugs:11. pages 2, 6, 8, 22, 52, 53

[3] Sun C, Li Y, Zhang Q, Gu T, Su Z. Perses: syntax-guided program reduction. In: Proceedings of the 40th International Conference on Software Engineering. Gothenburg Sweden: ACM; 2018. p. 361–371. Available from: `https://dl.acm.org/doi/10.1145/3180155.3180236`. pages 2, 7, 22, 51, 52, 53

[4] Donaldson AF, Evrard H, Lascu A, Thomson P. Automated testing of graphics shader compilers. Proceedings of the ACM on Programming Languages. 2017 Oct;1(OOPSLA):93:1–93:29. Available from: `https://doi.org/10.1145/3133917`. pages 2, 9, 27

[5] Chen J, Patra J, Pradel M, Xiong Y, Zhang H, Hao D, et al. A Survey of Compiler Testing. ACM Computing Surveys. 2020 Feb;53(1):4:1–4:36. Available from: `https://doi.org/10.1145/3363562`. pages 3, 4, 5

[6] Tang Y, Ren Z, Kong W, Jiang H. Compiler testing: a systematic literature analysis. Frontiers of Computer Science. 2020 Feb;14(1):1–20. Available from: `https://doi.org/10.1007/s11704-019-8231-0`. pages 3, 5

[7] Csmith;. Available from: `https://embed.cs.utah.edu/csmith/`. pages 3

[8] Yang X, Chen Y, Eide E, Regehr J. Finding and Understanding Bugs in C Compilers:12. pages 3, 5, 6, 33

[9] Livinskii V, Babokin D, Regehr J. Random testing for C and C++ compilers with YARPGen. Proceedings of the ACM on Programming Languages. 2020 Nov;4(OOPSLA):196:1–196:25. Available from: `https://doi.org/10.1145/3428264`. pages 3, 4, 5, 6, 8

[10] Groce A, Zhang C, Eide E, Chen Y, Regehr J. Swarm testing. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ISSTA 2012. Association for Computing Machinery;. p. 78–88. Available from: `https://doi.org/10.1145/2338965.2336763`. pages 4

[11] Donaldson AF, Lascu A. Metamorphic testing for (graphics) compilers. In: Proceedings of the 1st International Workshop on Metamorphic Testing. MET '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 44–47. Available from: `https://doi.org/10.1145/2896971.2896978`. pages 4, 5, 6, 9

[12] Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. ACM SIGPLAN Notices. 2014 Jun;49(6):216–226. Available from: `https://doi.org/10.1145/2666356.2594334`. pages 4, 5

[13] Sun C, Le V, Su Z. Finding compiler bugs via live code mutation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery; 2016. p. 849–863. Available from: `https://doi.org/10.1145/2983990.2984038`. pages 4

[14] Xu DN, Peyton Jones S, Claessen K. Static contract checking for Haskell. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '09. New York, NY, USA: Association for Computing Machinery; 2009. p. 41–52. Available from: `https://doi.org/10.1145/1480881.1480889`. pages 4

[15] McKeeman WM. Differential Testing for Software. 1998;10(1):8. pages 4

[16] Le V, Sun C, Su Z. Finding deep compiler bugs via guided stochastic program mutation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. Pittsburgh PA USA: ACM; 2015. p. 386–399. Available from: `https://dl.acm.org/doi/10.1145/2814270.2814319`. pages 5

[17] Hashimoto A, Ishiura N. Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs;9(0):21–29. Available from: `https://www.jstage.jst.go.jp/article/ipsjtsldm/9/0/9_21/_article`. pages 5

[18] C-Reduce;. Available from: `https://embed.cs.utah.edu/creduce/`. pages 6, 7, 8, 22

[19] Zeller A. Yesterday. my program worked. Today, it does not. Why?:15. pages 7

[20] Misherghi G, Su Z. HDD: hierarchical delta debugging. In: Proceeding of the 28th international conference on Software engineering - ICSE '06. Shanghai, China: ACM Press; 2006. p. 142. Available from: `http://portal.acm.org/citation.cfm?doid=1134285.1134307`. pages 7, 52

[21] UndefinedBehaviorSanitizer — Clang 13 documentation;. Available from: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`. pages 7

[22] Frama-C - Framework for Modular Analysis of C programs;. Available from: `https://frama-c.com/`. pages 7

[23] Donaldson AF, Evrard H, Thomson P. Putting Randomized Compiler Testing into Production. 2020:30. pages 7, 23, 35, 36

[24] Holmes J, Groce A, Alipour MA. Mitigating (and exploiting) test reduction slippage. In: Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation. ACM;. p. 66–69. Available from: `https://dl.acm.org/doi/10.1145/2994291.2994301`. pages 7

[25] Even-Mendoza K, Cadar C, Donaldson AF. Closer to the edge: testing compilers more thoroughly by being less conservative about undefined behaviour. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE '20. New York, NY, USA: Association for Computing Machinery; 2020. p. 1219–1223. Available from: `https://doi.org/10.1145/3324884.3418933`. pages 8, 50

[26] Lidbury C, Lascu A, Chong N, Donaldson AF. Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '15. Association for Computing Machinery;. p. 65–76. Available from: `https://doi.org/10.1145/2737924.2737986`. pages 8

[27] Herklotz Y, Wickerson J. Finding and Understanding Bugs in FPGA Synthesis Tools. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. FPGA '20. Association for Computing Machinery;. p. 277–287. Available from: `https://doi.org/10.1145/3373087.3375310`. pages 8

[28] SPIR-V Tools. The Khronos Group;. Original-date: 2015-11-11T12:56:25Z. Available from: `https://github.com/KhronosGroup/SPIRV-Tools`. pages 10

[29] Donaldson AF, Thomson P, Teliman V, Milizia S, Maselco AP, Karpiński A. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021. Association for Computing Machinery;. p. 1017–1032. Available from: `https://doi.org/10.1145/3453483.3454092`. pages 10

[30] The Khronos Group; 2021. Section: General. Available from: `https://www.khronos.org`. pages 11

[31] Reference Compiler;. Available from: `https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/`. pages 11

[32] ANGLE - Almost Native Graphics Layer Engine. Google;. Original-date: 2014-08-26T14:59:07Z. Available from: `https://github.com/google/angle/blob/47279c726e5ac0f5574c4796d6251b6d1614a566/README.md`. pages 12, 42

[33] Zink — The Mesa 3D Graphics Library latest documentation;. Available from: `https://docs.mesa3d.org/drivers/zink.html`. pages 12

[34] OpenGL - Drawing polygons;. Available from: `https://open.gl/drawing`. pages 12

[35] Base code - Vulkan Tutorial;. Available from: `https://vulkan-tutorial.com/Drawing_a_triangle`. pages 12

[36] Piglit;. Available from: `https://piglit.freedesktop.org/`. pages 12

[37] VkRunner. Igalia;. Original-date: 2018-03-10T09:29:27Z. Available from: `https://github.com/Igalia/vkrunner/blob/1b4cc6b129e857d88ba9487bc8a4983d6a11df02/README.md`. pages 13

[38] Amber. Google;. Original-date: 2018-11-13T22:00:36Z. Available from: `https://github.com/google/amber/blob/d5572879717d6053a739143b7d616f162431d86a/README.md`. pages 13

[39] ShaderTrap. Google;. Original-date: 2021-01-21T13:22:49Z. Available from: `https://github.com/google/shadertrap`. pages 13

[40] Kessenich J. The OpenGL Shading Language 4.5:213. pages 14

[41] Simpson R. The OpenGL ES Shading Language:203. pages 14, 20

[42] IEEE Standard for Floating-Point Arithmetic:1–70. Conference Name: IEEE Std 754-2008. pages 20, 30

[43] FloatingPointMath - GCC Wiki;. Available from: `https://gcc.gnu.org/wiki/FloatingPointMath`. pages 20

[44] Clang Compiler User's Manual — Clang 13 documentation;. Available from: `https://clang.llvm.org/docs/UsersManual.html#controlling-floating-point-behavior`. pages 20

[45] Unix Drivers | NVIDIA;. Available from: `https://www.nvidia.com/en-us/drivers/unix/`. pages 24

[46] LLVMpipe — The Mesa 3D Graphics Library latest documentation;. Available from: `https://docs.mesa3d.org/drivers/llvmpipe.html`. pages 24

[47] google/swiftshader. Google; 2021. Original-date: 2016-06-30T09:25:24Z. Available from: `https://github.com/google/swiftshader/blob/90c0551ca547914f96b32d50bba9c2126b45be41/docs/ANGLE.md`. pages 24

[48] Intel® Graphics for Linux*;. Available from: `https://01.org/linuxgraphics`. pages 24

[49] Responsible and Effective Bugfinding – Embedded in Academia;. Available from: `https://blog.regehr.org/archives/2037`. pages 46