CS 583 Project Report
Functional Domain Specific Shader Language (FDSSL)

Ben Friedman & Cody Holliday

June 2021

1 Overview

Our project is FDSSL, or the Functional Domain Specific Shader Language. We implemented this language in Haskell, and we designed it to compile directly to OpenGL Shader Language (GLSL). This DSL enables writing a single program that compiles to two separate GLSL programs, the vertex and fragment shaders. These are normally written as separate GLSL programs, but exist as part of the same pipeline from vertex to fragment. This DSL abstracts this fragmentation away, so we can reason about the shader pipeline as one program. This makes is easier to verify the integrity of programs from a type level, as well as to allow building shaders compositionally.

The intended users of our project fall into two categories, people who are interested in starting with graphics, and people who would prefer working with a simpler functional language than GLSL. People can use FDSSL to write programs that will represent standard vertex and fragment shaders, allowing them to perform vertex and color transformations from a higher level of reasoning (like surface deformations and lighting calculations), abstracting away the implementation details needed to produce the equivalent GLSL programs. Most importantly, people can reason about the an entire GLSL program together in one spot, to make it simpler to understand.

To provide some brief background, GLSL is a C-like language that compiles to an instruction set that runs on the GPUs of most contemporary hardware [4]. GLSL programs can be described as single program multiple data (SPMD), as they are run on parallel with little to no connection between separate instances [6]. We believe GLSL shaders, lacking a notion of state between individual executions, are well suited for reasoning about in a higher-level, and in a functional paradigm. We believe this could be helpful, as GLSL is hard to reason about due to its imperative style, is slow to debug, often leads to code duplication across programs, and performs undefined actions for runtime errors.

There exist several other instances of functional shader langauges. These include: Renaissance, Vertigo, Parasol, and Indigo, among others [1, 2, 3, 5].

The first two are quite old at this time, being developed in the early 2000's, the third is a smaller project that is undergoing a large rework, and the last is only available as part of a professional editing suite. Our project differs from all of these by reasoning about the GLSL pipeline and compiling to GLSL directly, which is something none of these languages do together.

1.1 Example

An basic FDSSL program is as follows:

```
uniform Float uOffset

// vertex shader, updates the gl_Position for the rasterizer
// & outputs a 2-tuple vXY
vert v : (Vec3 aVertPos) -> (Vec2 vXY) = {
  set gl_Position vec4 aVertPos[0] aVertPos[1] aVertPos[2] 1.0
  out vXY (vec2 (aVertPos[0] + uOffset) (aVertPos[1] + uOffset))
}

// fragment shader
// accepts a 2-tuple vXY to color R & G based on
frag f : (Vec2 vXY) -> () = {
  set gl_FragColor (vec4 vXY[0] vXY[1] 0.0 1.0)
}

e0 : Prog = mkProg v f
```

This program describes a pair of GLSL shaders, a Vertex shader `v` and a Fragment shader `f`. The syntax `() -> ()` denotes the signature of a shader, being the opaque values provided *and* produced for usage in a later stage. This example shows a shader which accepts a 3-tuple `aVertPos` (vertex position) and produces a 2-tuple `vXY` for usage in a fragment shader. This program also describes a single uniform (or global) float called `uOffset`, which is provided from outside the program, and happens to be used to offset the values in `vXY`. When we say outside the program, we mean whatever is whatever context is running the GLSL shader program, which can C++, Haskell, Python, Javascript, etc. In this example the vertex shader simply sets the output vertex position to an `x,y,z,w` coordinate (w is an additional parameter to denote a relative positional scale in 3D space, such as for points with the same relative position but farther in the distance or closer than expected). The fragment shader here simply sets a `red, green, blue` color value with an `alpha` transparency value. Finally, `e0` denotes a final program that incorporates all uniforms present and the shaders `v` and `f`. The final meaning of program `e0` is a GLSL shader that shows all vertices in their original positions with no transformations, but sets a varying color from red to green in the fragments that are produced; the result of which can be seen below in 1. Multiple programs can be defined in the same FDSSL file.

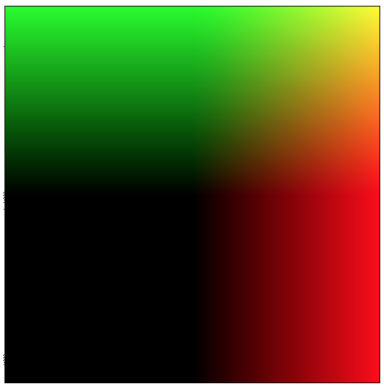

Figure 1: The result of evaluating the Vertex and Fragment shaders produced from the FDSSL program `e0`; a simple green, red and black sheet.

2 Import Aspects

- `Prog` in **Syntax.hs** describes an FDSSL program that represents a complete GLSL program with vertex and fragment shaders included, as well as external variables such as `uniforms` and `attributes`. `Prog` captures the transformations that describe the GLSL pipeline from Vertex to Fragent shader. This representation is the core of our language, and it is what we typecheck and pass to the pretty printer.

- `Shader` in **Syntax.hs** encapsulates the environment of the main function in a GLSL shader. We denote the "inputs" and "outputs" of the shader as parameters of the constructor, so that they can be easily referenced within the shader as well as easily composed between shaders. Composition is a feature we would like to perform further work on, but the current form is designed to directly combine two shaders together, so that the first shader's expressions are evaluated before the second shader's expressions. The result sequences the effects that are generated through evaluation, and allows later shaders to take advantage of elements in the environment that would otherwise not be available.

- `Func` in **Syntax.hs** holds the definition of a given function. We tag the function with a shader type during the type checking process, because it may be that the user references a variable that is specific to a certain shader type. This is not represented within the type itself, but is handled in the `Funcs` type.

- `Opaque` in **Syntax.hs** is an important type because we cannot access its value. We need to keep track of what type of opaque value it is, uniform, attribute, varying, so that we can check if it's available in a shader, if the variable can be modified, etc. We keep track of the type through the `Type` data type.

- `typeCheckProg` **TypeChecker.hs** is where type checking begins. `runTypeChecker` initializes the state, but it is only during this function that it does things such as checking for overlapping names and inserting shaders in the correct places.
- `parseFDSSL` in **Parser.hs** uses all of the different root expression functions. In this context, we mean uniforms, functions, shaders, and composition. Within this function you can see the specific parser functions that make up the language.
- `runPrettyPrinter` in **Pretty.hs** is the start of the printing tree. It uses the State monad to keep track of indentation in a fashion similar to Peano numbers.

3 Design Decisions

3.1 Abstract Syntax

In our project we choose to use a standard Abstract Syntax in Haskell to capture the essential structure of FDSSL. We considered using HOAS, PHOAS, or encoding our abstract syntax through a GADT. However, we decided early on that we wanted to have type errors that were unique to FDSSL and not Haskell. These other approaches of implementing our abstract syntax would have carried numerous benefits by embedding FDSSL types through Haskell's type system, and would have saved us the effort of writing our own type system; but we considered the control over the type messages was more important for our use case.

3.2 Parser Combinators for Parsing

We decided to utilize paser combinators through Parsec to build the FDSSL parser. This was done partially due to the experience that our group has had had with parser combinators before taking this course, but also due to the effectiveness of parser combinators. We were able to design minimal parsers for sections of FDSSL, such as expressions, functions, or shaders, and we were able to combine them effectively. This saved us development and debugging time during our work on FDSSL.

3.3 Monad Transformer Stack for TypeChecking

We used a monad transformer stack to allow us to build a simple type checker for FDSSL. We believed that this stack would be sufficient as FDSSL is statically typed, with no type inference needed as all terms are explicitly annotated. Our type checking stack was implemented as follows via the `State` and `Error` monads:

```
type TypeChecked m = (
  MonadState AllEnvs m,
  MonadError Error m)
```

This monad transformer allowed us to accumulate a type checking environment as new bindings become known, comprised of mutable and immutable bindings. `AllEnvs` contained a mixture of local bindings, functions, and uniform declarations. This let us differentiate between constant bindings across shaders, and unique bindings for shaders or functions.

3.4 State Monad for Pretty Printing

We utilized the State monad to implement the pretty printer for FDSSL. We initially used a series of `show` instances to build a prototype pretty printer, but this had several issues with tracking indentation. By switching to the `State` monad we were able to track both the indentation level at any point while pretty printing, and we were also able to maintain a stack of pretty printed syntax. This was particularly helpful as it allowed retrieval and manipulation of previously printed elements in the state to make changes to those elements. This was done to convert from FDSSL to GLSL, which needed notions of terminating statements with semi-colons, and prefacing of expressions that produced final values with **return** keywords. Admittedly, this would have been better to have a separate abstract syntax for GLSL, but we would like to consider this a long term goal past this project deadline.

4 Future Work

We had high ambitions for features when we started this project, and we accomplished part of what we intended to finish. The current version of our TypeChecker does not ensure FDSSL programs are type safe, as it is unfinished. However, it does do a very good job at covering most cases to prevent issues, but we would like to further improve it. Recursion is not prevented in our type checker currently as well, which we need to add (recursion is disallowed in GLSL). Finally, we would like to add `Compiler` and `GLSLSyntax` modules to decouple FDSSL programs from GLSL programs. This would introduce a compiler step from FDSSL abstract syntax into GLSL abstract syntax, followed by pretty printing our GLSL programs. There are other changes we would like to make as well to improve upon our early working model of this language.

References

[1] Chad Anthony Austin. Renaissance : a functional shading language. https://lib.dr.iastate.edu/rtd/18895, 2005. Accessed on April 19th, 2021.

[2] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on haskell*, Haskell '04, pages 45–56. ACM, 2004.

[3] Aubrey Jones. parasol: A highly-composable functional shader language for opengl and vulkan. https://github.com/aubreyrjones/parasol, 2015. Accessed on April 19th, 2021.

[4] John Kessenich, Google, Dave Baldwin, and Randi Rost. The opengl shading language, version 4.60.7. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf, 2019. Accessed on April 19th, 2021.

[5] Glare Technologies Limited. Indigo shader language. https://www.indigorenderer.com/documentation/manual/indigo-scenes/materials/indigo-shader-language, 2021. Accessed on April 19th, 2021.

[6] Bo Joel Svensson, Mary Sheeran, and Ryan Newton. Design exploration through code-generating dsls: High-level dsls for low-level programming. *Queue*, 12(4):40–52, April 2014.