# Compilation of a functional shading language to a SPIR-V intermediate representation

ANDRZEJ SWATOWSKI, University of Warsaw, Poland

CCS Concepts: • **Computing methodologies Graphics processors**; • **Software and its engineering Compilers**; • **Software and its engineering Parallel programming languages**;

Additional Key Words and Phrases: functional programming, shading languages, static single assignment, SPIR-V

## 1 INTRODUCTION

Since the debut of the GeForce 3 in 2001, programmers can control parts of the graphics pipeline when rendering 3D scenes in either video games or 3D modelling software [13]. They do so by writing programs, usually called *shaders*, that run on GPU in specific parts of the graphics pipeline. Programmers can write shaders using specialized shading languages — historically, the most popular ones have been OpenGL's GLSL and Microsoft's HLSL [2]. Both languages model shaders as effectful imperative programs. On the other hand, one can think of a shader as a pure mathematical function that maps an input stream of homogeneous records to an output stream [11]. Such a point of view — interpreting computations as functions — is traditionally associated with functional programming. Even though in the presence of such correspondence combining functional and shader programming might seem obvious, to the best of our knowledge there has been little significant attempt to apply functional paradigm in shader programming. With the introduction of Vulkan graphic API and SPIR-V intermediate representation of shaders, we were able to build Vinci — a functional shading language and its dedicated compiler targetting SPIR-V.

Our main contribution is the design of a proof-of-concept compiler for a toy ML shading language. We will first discuss the limitations of GPU programming (Section 2) and then exploit correspondence between CPS and SSA intermediate forms (Section 3). To make our language usage easier, we will propose few abstraction mechanisms unusual in shader programming (Section 4). We will discuss related work in Section 5.

## 2 LIMITATIONS OF GPU SHADER PROGRAMMING

One of the most critical limitations of GPU shader programming is the inability to call recursive functions on a GPU. Vulkan and OpenCL specifications are very restrictive — Vulkan disallows cycles in the static function-call graph of a shader [4], and OpenCL does not support recursion [3]. Such limitation severely limits the expressive power of a functional language, as most of the control flow in functional programs is expressed using recursion. Conveniently, Vulkan allows the usage of imperative loops. As we know, a tail-recursive call may be translated into a while-loop [16]. In Vinci, we require shader programmers to use only the tail-recursive calls, as it allows us to translate such calls into an imperative loop.

Another notable limitation is the lack of function pointers in Vulkan. Although it once again limits the expressive power of a functional language, as it prevents a programmer from using higher-order functions, it is not the subject of this paper. One of the possible solutions would be to *defunctionalize* the code [15], but such transformation falls outside the scope of our research.

Furthermore, GPU does not allow for dynamic memory allocation. Apart from thread-local registers, all of the memory required by GPU computation must be preallocated before starting

Author's address: Andrzej Swatowski, as386085@students.mimuw.edu.pl, University of Warsaw, Institute of Informatics, Warsaw, Poland.

the computation using library functions that communicate with the device drivers [4]. Functional programs allocate a significant amount of memory while running, most notably to pass closures or partially evaluated functions. To eliminate the need for introducing closures, we use the transformation called lambda lifting in our compiler, transforming local functions into global ones [1].

## 3 CPS-SSA CORRESPONDENCE AND THE COMPILATION

Both syntax and semantics of Vinci are based around those of functional programming languages from the ML family. In contrast to those languages, Vinci does not offer algebraic data types or pattern matching. On the other hand, it uses C-like structures (similar to records from functional programming) and tuple destructuring syntax. Vulkan expects shaders to work on C structures [4], hence the idea to include them in the language. As shaders mostly perform linear mappings and use vectors and matrices, including tuple destructuring syntax seems beneficial to the programmer.

SPIR-V is a binary intermediate language for graphical shaders parsed by the Vulkan library and sent to the GPU. SPIR-V is using static single assignment (SSA) form to represent functions [10]. While the SSA form is usually used as an intermediate representation of imperative programs, Richard Kelsey proved it is formally equivalent to Continuation-Passing Style (CPS) intermediate representation used in functional language compilers by defining syntactic functions converting CPS to SSA and SSA to CPS [8]. As SPIR-V expects the code to be in SSA form, we can borrow the idea from Kelsey and transform the functional code represented by the CPS to the SSA form expected for the SPIR-V.

To convert an expression written in our shading language to the CPS form, we use a function based on the one presented by Andrew Kennedy for his toy ML language [9] but adjusted to meet the needs of our language. The most critical adjustment is the conversion of the constructor of a C-like structure and the dot operator used for structure member selection (introduced in Vinci for a programmer's convenience). However, such conversions can be easily modelled after those for tuple constructor and tuple projection, as the C-like structure can be seen as isomorphic to a tuple.

SPIR-V specification requires the code to label so-called *merge blocks* before any branching instruction [10]. *Merge block* is the one that the control flow will eventually approach after branching — Vulkan uses those labels to validate the correctness of the SSA form's control-flow graph. SPIR-V instruction `OpSelectionMerge` precedes branches generated by if-else, while `OpLoopMerge` precedes those representing loops. After the CPS-to-SSA conversion and control-flow graph cleaning [17], the compiler runs a graph traversal algorithm searching for branching instructions and labelling them accordingly. It can either find a loop (and annotate it with the `OpLoopMerge` instruction) or search for the lowest common ancestor of two branching blocks (for the `OpSelectionMerge`).

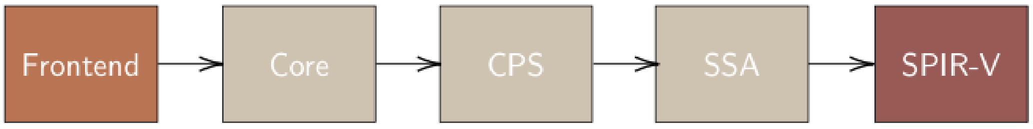

Fig. 1. Scheme of the Vinci compiler

The compilation process can be logically divided into five stages (presented in Figure 1): parsing Vinci code, desugaring Vinci into a small internal language called Core, transforming Core into CPS form, transforming CPS into SSA form, and outputting the SPIR-V code. Core's name is an apparent reference to Haskell's internal language, as it serves a similar purpose. We also distinguish the "internal" SSA form from SPIR-V, as the latter is very verbose, and it is easier to run optimizations and transformations on a more straightforward internal SSA form.

## 4 POLYMORPHISM AND TYPE INFERENCE

To make programming with Vinci more effective, we propose abstraction mechanisms linked with type theory, unusual in shader programming — polymorphism and type inference.

Polymorphism is known to speed up the programmer's work and make the code less error-prone. Without polymorphism, the programmer must write the same function multiple times for every combination of types it will be used with. Both GLSL and HLSL, being inspired by C, do not support polymorphism. On the other hand, as Apple's Metal Shading Language (MSL) design is based on C++14, it offers parametric polymorphism in the form of C++ templates [6]. In Vinci, we decided to do a similar thing and introduce parametric polymorphism typical for functional languages from the ML family. As types of input and output variables of the shader must be explicitly annotated, we can run a graph traversal algorithm monomorphising polymorphic functions, similar in vein to algorithms monomorphising C++ templates. The algorithm starts its run in the shader `main` function (shader's entry point) and recursively monomorphises functions for given types if such a combination of types is not monomorphised already.

One of the most widely known benefits of functional programming is the ease of introducing type inference into the program compilation phase. We decide to base our type inference algorithm around *Algorithm W* proposed by Milner for the Hindley-Milner (HM) type system [12]. One thing we need to adjust in the algorithm is the handling of operator overloading. As we know, the HM type system allows for parametric polymorphism but not for ad hoc one, and operator overloading is considered an instance of the latter [14]. OCaml solves the problem by forcing the programmer to use different operators for integer and floating-point number operations, e.g. `+` and `+.` for addition. In our opinion, it may make shaders unnecessarily cluttered. Inspired by Haskell and the work of Mark P. Jones [7], we include an additional step of constraint checking in the unification step, basically introducing a small subset of type classes into the algorithm. Every polymorphic function that uses arithmetic operators is annotated with a specific constraint (a type class) passed into all the call sites and checked against the concrete type if necessary. This step ensures that no operator would be used on values of incompatible types (e.g. addition operator `+` on the Boolean data types). Considering how powerful type classes are, we refuse to make them available to users of our language and decide to maintain them as a specific part of the type inference algorithm instead. That said, however, having introduced constraints for arithmetic operators, we decide to extend their support to Vulkan shader's library functions such as `sin`, `abs` or `clamp`.

## 5 RELATED WORK AND CONCLUSION

In our opinion, there is still little research on the compilation of languages different from GLSL to SPIR-V. Translation of a non-imperative language APL to SPIR-V representation was recently studied [5], but the translation was done by hand by the author instead of relying on an automatic compiler. A library for outputting SPIR-V for the D language was also proposed [18], but it focuses on supporting the compute kernels instead of graphic shaders.

We have presented the design of a proof-of-concept functional shading language compiler. When creating the compiler, our primary goal was to discover the limitations of GPU programming and explore possibilities of compiling functional language. We have compiled multiple functional programs written in our toy ML language to SPIR-V and saw the output shaders modify the rendered image in real time. We do hope that this paper sparks the interest of the functional programming community in GPU programming.

## REFERENCES

[1] Olivier Danvy and Ulrik Pagh Schultz. 2002. Lambda-Lifting in Quadratic Time. In *Proceedings of the 6th International Symposium on Functional and Logic Programming (FLOPS '02)*. Springer-Verlag, Berlin, Heidelberg, 134–151.

[2] Kayvon Fatahalian and Mike Houston. 2008. GPUs: A Closer Look: As the Line between GPUs and CPUs Begins to Blur, It's Important to Understand What Makes GPUs Tick. *Queue* 6, 2 (March 2008), 18–28. https://doi.org/10.1145/1365490.1365498

[3] The Khronos OpenCL Working Group. 2011. The OpenCL Specification Version 1.2. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[4] The Khronos Vulkan Working Group. 2021. Vulkan 1.2.182 - A Specification. https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf

[5] Juuso Haavisto. 2020. *Leveraging APL and SPIR-V languages to write network functions to be deployed on Vulkan compatible GPUs.* Master's thesis. Université de Lorraine. https://hal.inria.fr/hal-03155647

[6] Apple Inc. 2021. Metal Shading Language Specification Version 2.4. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf

[7] Mark P. Jones. 1999. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Erik Meijer (Ed.). Department of Information and Computing Sciences, Utrecht University, Paris, France, 7–22.

[8] Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, USA) *(IR '95)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/202529.202532

[9] Andrew Kennedy. 2007. Compiling with Continuations, Continued. *SIGPLAN Not.* 42, 9 (Oct. 2007), 177–190. https://doi.org/10.1145/1291220.1291179

[10] John Kessenich, Boaz Ouriel, and Raun Krisch. 2021. SPIR-V Specification Version 1.5. https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf

[11] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. 2004. Shader Algebra. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 787–795. https://doi.org/10.1145/1015706.1015801

[12] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[13] John Nickolls and David Kirk. 2013. Graphics and Computing GPUs. In *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface* (5th ed.), David A. Patterson and John L. Hennessy (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Appendix C, C3–C83.

[14] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[15] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[16] Guy Lewis Steele. 1977. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 Annual Conference* (Seattle, Washington) *(ACM '77)*. Association for Computing Machinery, New York, NY, USA, 153–162. https://doi.org/10.1145/800179.810196

[17] Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[18] Nicholas Wilson. 2018. DCompute: Compiling D to SPIR-V for Seamless Integration with OpenCL. In *Proceedings of the International Workshop on OpenCL* (Oxford, United Kingdom) *(IWOCL '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 3 pages. https://doi.org/10.1145/3204919.3204922