Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Elegance in Haskell GPGPU Programming

Jack Joseph Gilbride

April 19, 2022

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MCS (Computer Science)

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

General Purpose Graphics Processing Unit (GPGPU) programming enables substantial performance improvements in massively parallel problems, but requires a firm understanding of low level hardware concepts that are traditionally abstracted from an application layer programmer. Haskell, a purely functional programming language, is commonly referred to in the literature as "elegant", which has connotations with beautiful, modular and intuitive, but has no formal definition. Haskell's elegance potentially offers a solution to enable quicker and easier GPGPU programming. GPGPU programming in Haskell is a worthwhile exercise if this elegance can be preserved alongside performance benefits.

In the literature there is no formal definition for elegance, but the same concepts permeate discussions. Lazy evaluation, purity, type variables, higher-order functions, and elegant syntax (referred to as declaration-style programming) are found to be properties of elegance in Haskell. They form the working definition to assess the elegance of Haskell GPGPU programming. Accelerate and fractals are chosen as the suitable GPGPU library and problem area respectively under which to examine the question, before abstracting the results to Haskell GPGPU programming in general.

Working examples of generating the Mandelbrot Set in Haskell exist in Accelerate, demonstrating how to generate a static image, and a dynamic interactive GUI. The elegance of these is assessed, before iterating on the dynamic version to maximize the elegance under the working definition. The existing implementation is found to be high in purity, middling in declaration style & type variables, and low in laziness & higher order functions. The iteration introduces some improvements in type variables and declaration style, which remain limited by Accelerate. Laziness is unable to be introduced due to the execution of the program as a deep embedding in CUDA. Higher-order functions cannot be introduced to this specific implementation, but are found to be possible in general, and restricted in this implementation due to their link to laziness.

An evaluation of the results when applied to general GPGPU programming in Haskell show that Haskell GPGPU programming is elegant in terms of purity, syntax and higher-order functions, but not in terms of laziness and type polymorphism. An evaluation of research methods shows that type polymorphism was an invalid property under which to judge elegance. Under the iterated working definition, Haskell GPGPU programming is mostly elegant, aside from lack of lazy evaluation, which is seen to practically affect the elegance and modularity of code.

Acknowledgements

I would like to thank my supervisor, Glenn Strong, for the valuable guidance and support that he has given me throughout the year. I would also like to thank my family and friends for their unwavering support of my studies

Contents

List of Tables

Glossary of Terms

Haskell is a purely functional programming language.

A **Graphics Processing Unit (GPU)** is a specialised processor with a highly parallel architecture, designed to efficiently compute massively parallel problems.

General Purpose GPU (GPGPU) programming is the use of GPUs for general purpose programming, more generally than computation for computer graphics.

CUDA is a parallel computing platform and application programming interface developed by NVIDIA, that allows software to use NVIDIA GPUs.

A **Domain Specific Language (DSL)** is computer language specialised to a particular application domain.

An **Embedded Domain Specific Language (EDSL)** is a DSL written as an extension to an existing host language.

Accelerate is an EDSL for GPGPU programming in Haskell.

Nikola is an EDSL for GPGPU programming in Haskell.

Obsidian is an EDSL for GPGPU programming in Haskell.

Lazy evaluation (laziness) is an evaluation strategy which delays the evaluation of an expression until the value is needed.

A **pure** function is a function whose application has no side-effects.

Polymorphism is the use of a single symbol to represent multiple different types.

A **higher-order function** is a function that either takes one or more functions as arguments, or returns a function as a result.

1 Introduction

Haskell is a functional programming language commonly considered colloquially and in the literature to be "elegant". Concepts such as purity, lazy evaluation, higher-order functions, type classes and elegant syntax enable a compositional approach to programming where functions are first-class citizens. This enables a concise, modular approach to programming where each function cleanly deals with a particular aspect of program behaviour at a suitable level of abstraction.

Graphics processing units (GPUs) are specialised processors with a highly parallel structure. This makes them highly efficient at computing highly parallel problems; problems that can be solved across independent, concurrent threads. A GPU's reduced instruction set architecture maximizes individual instruction speed, enabling each thread to execute very quickly. GPU programming was traditionally based around graphics applications, but the development of General Purpose GPU (GPGPU) programming tools allowed GPUs to be leveraged for performance increases in different domains such as machine learning and image processing.

GPGPU programming utilizes GPU hardware concepts, and generally, a GPGPU programmer requires a firm understanding of these concepts to leverage performance benefits. GPGPU tools such as CUDA and OpenCL operate at this level of abstraction and provide their platform primarily through the C and C++ programming languages. A consequence of this is that GPGPU programs are typically of an imperative, procedural nature and deal with low-level concepts such as memory allocation and address spaces.

Haskell offers a potential solution to the knowledge gap required for GPGPU programming. Haskell's "elegance" is far removed from the imperative, procedural, low-level nature traditionally required to leverage these concepts. GPGPU programming libraries exist in Haskell, but there is little discussion in the literature about whether their use preserves its elegance. A substantial move away from elegance would diminish the benefit of Haskell GPGPU programming, as would a substantial move away from the performance boost of traditional GPGPU tools. So while Haskell's elegance potentially provides the solution to more accessible GPGPU programming, a lack of it would invalidate this solution in favour of other higher-level languages.

With these motivations, the research question addressed in this project is: *To achieve an acceptable performance improvement in GPGPU Haskell programming, do any elegant properties of a traditional functional approach need to be sacrificed? If so, what particular ones need to be sacrificed, and why?*

This question should be analysed under a particular problem area, one which can produce generalizable results. The problem area chosen for this project is fractal generation. Fractal generation is particularly suited to Haskell, as its functional and compositional nature allow a program to closely resemble its mathematical counterpart, and its use of lazy evaluation enables the processing of infinite lists which are a fundamental element of fractals. Fractal generation is also particularly suited to GPGPU programming, as it is a massively parallel problem where each pixel can be generated independently by individual GPU threads. The elegance of Haskell and the performance of GPUs can both be utilized effectively in fractal generation, which makes it a sensible problem under which to examine their intersection.

To see if "elegance" can be maintained in Haskell GPGPU programming, the properties that make a Haskell program elegant first need to be clearly defined. There is an element of subjectivity to this, but it can be well informed and grounded by background research. An appropriate Haskell GPGPU library must also be chosen to carry out the implementation.

Once elegance has been defined and a library chosen, an existing implementation will be chosen to assess the elegance of it. Once the elegance has been assessed, it will be iterated on, to increase the elegance as far as possible. If it is not possible to achieve the full elegance of the traditional functional approach, then some sacrifice to elegance has to be made. The background research and work on the implementation will inform a discussion on whether this sacrifice has to be made, to what extent, and for what reasons.

2 Background Research

2.1 "Elegance" in Haskell

Haskell is a general purpose, purely functional programming language. Haskell incorporates many innovations in language design and as a result it is commonly considered in the literature and colloquially to be "elegant". This paper assumes a familiarity with Haskell of the reader. Those unfamiliar with Haskell are guided to Hudak and Fasel's "A Gentle Introduction to Haskell" [1] for an overview of the language, and the Haskell 2010 Language Report [2] for a more in depth understanding. Nonetheless, this section introduces the key properties of Haskell that will be referred to throughout the paper, and should provide a basic understanding of these concepts to those unfamiliar with Haskell.

While Haskell is commonly considered elegant, there is no agreed formal definition of what "elegance" actually is. One thing that is made clear by members of the original Haskell committee in "A History of Haskell" is that the ability to elegantly code was intended by design [3]. The paper never explicitly defines elegance, but touches on the concept throughout, while discussing design choices and properties of Haskell which contribute to it.

Haskell's core property as per the authors is that "Haskell is lazy". Lazy evaluation is demand driven; evaluation of functions and variables is deferred to when their values are required in a computation, if at all. One of the authors, Hughes, had previously discussed this in his 1989 paper "Why Functional Programming Matters", citing laziness as one of the two key elegant properties of functional languages [4]. Hughes particularly attributed laziness to the ability to modularise a problem into a generator which generates possible answers, and a selector which lazily selects the appropriate one. Laziness allows the generator to generate infinite output, and termination conditions to be separated from loop bodies. Any lazily evaluated part of a program can be modularised in this way. As Hughes notes, with a program made up of functions, lazy evaluation can be used "as glue to stick their parts together".

As an immediate consequence of laziness comes Haskell's second core property; "Haskell is pure" [3]. When code execution is demand-driven, it is difficult to reliably perform

side-effects, including I/O. So Haskell is also pure; each line of code has no side effects. Without side-effects, every call to a function f returns the same value, meaning "f really is a function in the mathematical sense". The authors note that pure purity was an "elegant attack on the challenge of programming, and it was that combination of power and beauty that motivated the designers".

While laziness and purity are Haskell's core properties, the authors note "Haskell's most distinctive characteristic" as type classes. Type classes are groups of types which have the same operations defined over them. For example, every instance of the Num type class must have the function (+) defined over it. This is more elegant than a type system where addition for each type must be defined with a unique function name. Functions may make use of type classes by leveraging ad-hoc polymorphism, where its arguments are not bound to a specific type, but a specific type class. This makes the function more general, but ensures that its arguments will still have the necessary type class properties to work in a computation. Haskell also allows parametric polymorphism, where a type is completely unconstrained. Again this allows functions to be polymorphic in their type, reducing function duplication and increasing generalisability.

Aside from language constructs like laziness, purity and type classes, the syntax and style of Haskell are commonly credited with contributing to its elegance. The authors from the Haskell Committee draw a distinction between two different "styles" of functional programs that can be written. The first is "declaration style" programming, which defines functions as multiple equations. Declaration style makes use of the where clause, places function arguments on the left hand side of the equals, and uses pattern matching or guards to identify which case each equation covers. The second type is "expression style", in a function is a larger body composed of smaller expressions. Expression style programming makes use of let expressions, lambda abstractions, case expressions and if expressions to build larger function bodies. The distinction between expression style and declaration style puts a formal definition on the idea of style and syntax which later informed me when defining elegance.

A fourth key feature noted in the History of Haskell as enabling elegant design is the use of higher-order functions. A higher-order function is a function that takes one or more functions as arguments, or returns a function as its result, which allow operations to be combined in ways that otherwise would not be possible. As per the authors, they "allow encoding nonstandard behaviours and also provide the glue to combine operations". Hughes' "Why Functional Programming Matters" states the ability to use higher-order functions to be one of the two key elegant properties of functional languages [4]. Functional languages allow functions to be expressed as a combination of a higher order function and particular specialising functions. This modularisation leads to "smaller and simpler and more general modules", which can be used in other functions to encode other behaviours with little

effort.

2.2 NVIDIA CUDA

NVIDIA CUDA is a computing platform and programming model for GPGPU programming, which specifically targets NVIDIA GPUs [5]. While CPUs gain speedup and hide memory access latencies through data caching and flow control, GPUs achieve this through their parallel architecture, designed to be utilized for highly parallel problems [6].

They gain their speedup and hide memory access latencies through these parallel computations, unlike a CPU which achieves this with data caching and flow control [6]. A program made up of parallel and sequential parts could leverage a mix of GPUs and CPUs to utilize these distinct specialisations. Before CUDA was released, GPU tools were written specifically for graphics applications. GPGPU problems needed to be translated into a graphics context and written using graphics tools [7]. CUDA was released by NVIDIA in 2006 and allowed programmers to solve GPGPU problems without needing to translate them into a graphics context.

CUDA's programming model consists of a set of language extensions to the C programming language [6]. The language extensions expose three key concepts - a thread group hierarchy, shared memories and barrier synchronization.

A key concept in CUDA is kernels. Each thread is identified by the thread ID, a one-dimensional, two-dimensional, or three-dimensional index which represents the thread's position in its thread block. Thread blocks can contain up to 1024 threads. Blocks can be grouped into one-dimensional, two-dimensional or three-dimensional grids. A kernel can be executed by every thread in a grid of thread blocks. A grid of M blocks, each containing N threads, would execute the kernel $M*N$ times.

There are five memory spaces for a CUDA thread; the thread's local memory, the thread block's shared memory, global memory, constant memory and texture memory. Threads within a block can coordinate their execution through shared memory and synchronization. CUDA assumes that the CPU and GPU, the *host* and *device* respectively, have separate memory spaces, so provides managed memory to bring them into a common address space. The CUDA runtime provides functionality for device memory allocation, device memory deallocation, and host-device data transfer.

Any C source file containing CUDA extensions must be compiled with the NVIDIA CUDA Compiler (nvcc). These files can contain a mix of host code and device code. nvcc allows for offline compilation, where all code is precompiled and linked, or just-in-time compilation, where device code is compiled to binary code at runtime and specialised to the device.

Thread synchronization is an important concept in CUDA. Two threads which read from or write to the same memory location without synchronization cause undefined behaviour. The memory fence call, `__threadfence()`, enforces ordering on memory access. The `__threadfence()` call can occur at three levels of abstraction; just the thread block, all threads on the device, or all threads across the host and all devices. Another function used to synchronize a thread block is `__syncthreads()`, which waits until all threads in the block have reached it. If this is used in conditional code, where the condition is evaluated differently within the same block, then the code may hang or produce unintended side effects. CUDA also allows atomic functions on memory, e.g. `atomicAdd()`.

When working with data structures as input and output in CUDA, the programmer must be careful to index into them correctly. CUDA's low-level C implementation gives the programmer full control over memory indexing, but also means that the programmer has the same concerns as a C programmer, such as ensuring memory has been allocated, that the memory space is large enough, and that different functions do not overwrite each others results. With multi-threaded programming, there is the extra concern that multiple threads will write to the same location. Thus the programmer must be careful to index correctly into the data structure for each thread. This can be accomplished by calculating the index into the data structure based on the block ID, block dimension and thread ID.

To summarize, GPGPU programming enables high performance benefits for highly parallelisable problems. However it requires a firm understanding of low level concepts that are traditionally not exposed to an application level programmer, including memory spaces, address spaces, thread synchronization, indexing calculations, and data transfer. So while GPGPU programming is useful for performance, it requires a removal of abstraction between the hardware and application layers.

2.3 GPGPU EDSLs

2.3.1 Obsidian

Obsidian is a Haskell EDSL for GPGPU programming [8]. Its aim is to simplify the development of GPU kernels - in particular, to reduce the dependence on thread IDs present in CUDA [7][9]. This is for two reasons. Firstly, to remove the need to define indices based on block IDs, block dimensions and thread IDs, which is a non-trivial problem. Secondly, to reduce the close link between the number of threads and the size of the data structure the kernel works on, caused by indexing by thread IDs. This leads to little flexibility once the kernel is designed in terms of the size of the data structure or number of threads used.

Obsidian provides the functionality of CUDA kernel programming at a higher level; indexing based on thread IDs is abstracted away from the user, as is the dependency between number

of threads and the data structure.

An Obsidian array does not specify an area of memory like its CUDA counterpart. Instead it is a Haskell abstract data type: `data Arr a = Arr (IndexE -> a) Int`, defined by an indexing function and a static length. Arrays can contain Integer, Float, Boolean, array or tuple values. Obsidian provides a number of common array functions in its array library including `rev`, `fmap`, `foldr`, `pair` and `zipp`.

To be executed by CUDA, an Obsidian array language program must be compiled to a kernel function. The function `pure` converts an Obsidian function to the correct kernel type in Haskell. These kernel types can be executed on the GPU from within GHCI using `execute`. We can write a simple program using a GPU to increment an array's contents like such in GHCI [9]:

```
execute (pure \$ fmap (+1)) [(0..9) :: IntE]
[1,2,3,4,5,6,7,8,9,10]
```

The above code generates a CUDA kernel which takes an input array and output array, increments the contents of the input array and stores the result in the output array. This can all be done in one line and the programmer does not need to allocate addresses or memory as they would in native CUDA, as Obsidian handles this automatically. The programmer also does not specify the number of threads either within the function or as an argument to Obsidian. This is because Obsidian handles this automatically, using one thread per array element. Obsidian then, has generated and run a CUDA kernel which uses ten threads to increment a ten element array, abstracting the address space, memory allocation and thread ID indexing from the programmer.

Two or more kernels can be sequentially composed using the `->-` operator to create a new kernel. The `->-` operation between two `pure` functions combines them via Haskell function composition, before generating the newly composed function in CUDA. Kernels can also be combined by explicitly storing the result from the first kernel, and passing it to the second kernel, using Obsidian's `sync` function. Using `sync` results in a call to CUDA's `__syncthreads()` if the number of threads is greater than the number of threads that can be run in one iteration, called a warp.

The knowledge that the size of a kernel's output determines the number of threads used, as well as the ability to use `sync`, gives the Obsidian programmer a lot of control over the CUDA code generated. For example, the programmer has the ability to make the call to `pair` towards the end of their kernel, so that an output array `[a, b, c, d, e, f]` becomes `[(a,b), (c,d) (e,f)]`, halving the number of array elements, therefore halving the number of threads. The close relationship between Obsidian and CUDA allows programmers to continue to think about concepts like blocks, grids, shared memory and

synchronization while abstracting some more C-centric concepts like the address space and index pointers.

One key limitations of Obsidian noted by Svensson is that the kinds of algorithms that can be expressed are those where the size of the outputs can be determined by the size of the inputs; it does not have the complexity for more general and data-dependent output sizes. Another limitation noted is that Obsidian allows for the design and implementation of individual kernels, but not for the coordination of kernels. GPGPU algorithms usually involve the coordination of multiple kernels, much as a sequential program is made up of multiple functions.

2.3.2 Accelerate

Accelerate is a Haskell EDSL for GPGPU programming, which, like Obsidian, aims to simplify GPGPU programming by reducing the workload and expert knowledge required to write CUDA programs [10][11].

Accelerate provides the developer with a higher level abstraction to GPGPU programming than either native CUDA or Obsidian. Instead of directly creating GPU kernels, the developer leverages Accelerate's paramaterized array operations. The compilation of array operations to kernels is hidden from the user, enabling them to utilize GPUs for performance benefits at a much higher level of abstraction.

Accelerate operations take place over arrays of type `Array sh e`, where `sh` specifies the shape and `e` specifies the element type. `e` may be a signed integer, unsigned integer, floating point number, double precision floating point number, character, boolean, tuple or array index. The array's shape is constructed with the constructor `Z` and infix operator `(:.)`. For example, a three-dimensional array of floats is of type:
`Array (Z:.Int:.Int:.Int) Float`, and element ijk is indexed at `(Z:.i:.j:.k)`.

Like CUDA and Obsidian, Accelerate distinguishes between CPU host memory and GPU device memory. In Accelerate, this distinction is made by the type of the array. A CPU array is of type `Array sh e`, while a tuple of one or more GPU arrays is identified by the type constructor `Acc`. Host-to-device memory transfer, i.e. the conversion of a CPU array to a GPU array, is achieved via the following function:

```
use :: (Shape sh, Elt e) => Array sh e -> Acc (Array sh e)
```

In addition to embedded array computations, Accelerate also enables embedded scalar computations with the type constructor `Exp`. Like those on `Acc` terms, computations on `Exp` terms are executed on the GPU device. `Exp` computations do not support recursion or iteration, as computations of this form cannot be achieved efficiently on GPUs. In addition, collective `Acc` operations may contain scalar `Exp` operations, but the inverse is not

true.

Scalar operations support Haskell's standard bitwise and arithmetic operations via overloading. They also support equality and conditional operators. Since Bool cannot be overloaded, this is achieved by appending `*` to the standard notation, e.g `==*`, `<=*`, `<*` etc. Conditional expressions are possible in the form `c ? (t, e)`, which evaluates to `t` if `c` is true and otherwise evaluates to `e`. Accelerate's collective array operations comprise of common Haskell list functions including `map`, `zipWith`, `fold` and `scan`.

Key to the mapping of Accelerate's collective array operations to CUDA kernels is the concept of algorithmic skeletons. Algorithmic skeletons are pre-defined CUDA kernels implementing each of the collective array operations. These skeletons were each hand-tuned to ensure efficient use of global memory and shared memory for communication between threads. The skeletons are paramaterized by types and scalar embedded expressions. For example, one of the parameters to `map` is the expression to map over the array. The general behaviour of `map` is predefined in CUDA to ensure efficient use of the GPU's hardware, but the function to map over the array can be defined by the programmer.

Accelerate code is compiled and run as CUDA code utilizing the CUDA API binding library [12] developed by the creators of Accelerate. This provides a single point of entry which compiles and evaluates the embedded GPU program:

```
CUDA.run :: Arrays a => Acc a -> a
```

The execution of *CUDA.run* is a two-pass process. The first pass generates the code for each collective array operation using nvcc. In this phase, memoisation is used to ensure that any skeletons used more than once are not compiled more than once. Additionally, this phase performs host-to-device memory transfer. The second pass performs any host-side computations and also invokes the compiled GPU kernels. The GPU kernels execute asynchronously, but the evaluator will block a kernel which requires the result of another kernel until the latter has finished execution.

The authors of both Accelerate and Obsidian note that Accelerate is a higher level abstraction from CUDA [9][11]. Obsidian exists at a lower level with the aim of providing the programmer with finer control of the use of GPU resources, shared memory and thread synchronization. Many of Accelerate's built in array computations could be built using Obsidian primitives. Accelerate abstracts away the concepts of shared memory and synchronization and controls this under-the-hood. The key concept that Accelerate maintains is the difference between device and host memory, although this does not need to be fully understood by the user, just that a "CPU Array" and "GPU Array" are of different types, and the type system ensures that this works correctly. While Obsidian programming defines single individual kernels, which need to be synchronized separately, Accelerate array

computations can be made up of multiple kernels under-the-hood, and synchronization is handled by the library. This enables purely-Accelerate GPGPU programs to be much larger than purely-Obsidian ones, as they can be multi-kernel. The difference in level of abstraction from CUDA is something intended by the creators of both EDSLs, as Obsidian aims to provide finer-grained control over design decisions while Accelerate aims to remove them as complications. While Obsidian gives an experienced GPGPU programmer more ability to work with concepts like thread blocks, shared memory and synchronization for performance benefits, Accelerate aims towards performance benefits in elements that it has abstracted, such as hand-tuned code skeletons and kernel memoisation.

A limitation noted by the author is that Accelerate currently targets just one GPU on the system, it does not target multiple GPUs on the machine where applicable.

2.3.3 Nikola

Nikola [13] is an EDSL for GPGPU programming in Haskell, which aims to simplify GPGPU programming by allowing Haskell code to be run on GPUs with minimal syntactic changes [14]. Like Obsidian and Accelerate, Nikola is compiled to CUDA code before it is run on the GPU.

Nikola allows general function compilation onto the GPU. This means that the type of a Nikola function is not fixed. They may be of any arity, take arguments of any size, and may be called any number of times with arguments of different sizes. A Haskell function requires minimal syntactic overhead to be converted to a Nikola function which will be executed on the GPU.

Nikola takes a similar approach to overloading standard Haskell operators as Accelerate does. Standard arithmetic operations are overloaded. Since `Bool` cannot be overloaded, conditional syntax is altered slightly by appending a `.` before and after the standard notation, e.g. `.<.` etc. Conditional expressions are possible in the form `c ? (t, e)`, which evaluates to `t` if `c` is true and otherwise evaluates to `e`.

Function application in Nikola follows the same syntax as standard Haskell. Behind the scenes, Nikola uses a concept called "let-sharing", so that an expression which appears multiple times in a function only needs to be evaluated once, after which all other occurrences of it point to the evaluated result. This is possible because Nikola translates pure functional code from Haskell, where let-sharing occurs by default. This does not occur automatically in CUDA code however, so Nikola carries "reification" to enable let-sharing for all sub-expressions in the function. Nikola also utilizes another sharing called "lambda-sharing", where sub terms that occur as a result of a lambda share that lambda. For example, if there are multiple occurrences of the function square, this will only be expanded once into `/x -> x * x`, and all occurrences of square will point to this one

expansion. Unlike let-sharing, this does not result in a performance increase but does reduce the final CUDA code size. To make use of lambda-sharing the developers provide a function called `vapply`, which ensures that the function it is passed undergoes lambda-sharing instead of being inlined. Let-sharing is achieved via reification which is done in Nikola's top-level function `reifyAndCompile`.

The translation of a Haskell function into a Nikola function is relatively simple; applying `vapply` to utilize lambda-sharing, rewriting conditionals to the correct Nikola syntax, and updating the type signature of the function to the `Exp` expression data type for Nikola to parse. For example, a function of type `Float -> Float` would be changed to type `Exp Float -> Exp Float`. These are the only changes needed due to the overloading of standard Haskell notation and preservation of Haskell function application as Nikola function application.

CUDA code is subject to a number of constraints that Nikola must take into account. Recursion is not allowed as it cannot be achieved efficiently on the GPU. Function pointers are also disallowed as their use does not support the device memory model. Additionally, all memory used by a kernel must be allocated before its invocation. Nikola has language features which enforce these properties. Functions are only allowed where all of its arguments are bound, i.e. closures and partial applications are disallowed. Vector operations are restricted to ones for which a static upper-bound memory requirement can be calculated. The size inference is restricted to allow functions to be compiled to a single kernel.

Like that of Obsidian and Accelerate, Nikola code is translated to CUDA and compiled by invoking nvcc. This is achieved in the `reifyAndCompile` function. The arguments are copied into GPU memory, the kernel is built, additional memory needed is allocated, the kernel is invoked, and finally the results are copied back to CPU memory. Nikola gives the user the option to compile at either Haskell run-time or Haskell compile time. The benefit of the latter is that a Nikola function does not need to be recompiled every time it is called, leading to performance improvements.

2.4 Fractals

2.4.1 Fractals

Benoit B. Mandelbrot, who coined the term "fractal", described fractals as "shapes whose roughness and fragmentation neither tend to vanish, nor fluctuate up and down, but remain essentially unchanged as one zooms in continually". [15] Mandelbrot contrasted this to a standard geometric shape which becomes "smoother" as we zoom in to a point on that shape (e.g. a curve which seems to converge closer to a straight line, the tangent, as we zoom in), or a chaotic shape whose roughness varies up and down under the same process.

This key property of fractals is called self-similarity. A more concise definition for self-similarity is that "each part is a linear geometric reduction of the whole."

Fractals are often made up of "structures of great richness", but the algorithms to generate them are often quite short [15]. Typically, the algorithm is made up of loops of simple basic instructions, which are followed repeatedly. Each iteration makes the shape increasingly complex.

Mandelbrot coined the theory of fractals including the name and their properties in 1975, although mathematical objects with these properties had been discovered independently in the preceding century, including the Cantor set, Fatou sets and Julia sets. Mandelbrot's advancement of the theory was aided by advances in computing, particularly in computer graphics. These allowed him to generate and visualize large numbers of iterations not previously possible with hand-drawn illustrations. He demonstrated the potential complexity of fractals with highly detailed computer-generated images of Julia sets.

Fractal geometry occurs and has implications in a multitude of areas including physics, probability theory, economics, hydrology, engineering and mathematics. The reason that I am concerned with it is its suitability to both Haskell programming and GPU programming. In section 2.4.2 I will examine why Haskell is a particularly good suitor to fractal generation, due in particular to its compositional nature and ability to process infinite lists. In section 2.4.3 I will examine how GPUs can be utilized for massive performance increases in fractal generation. The applicability of both of these programming areas make fractals a suitable lens under which to examine their intersection.

2.4.2 Elegance in Composing Fractals

Mark P. Jones' functional pearl "Composing Fractals" describes programs for composing fractal art, in particular the Mandelbrot Set and the Julia Set [16]. The main goal of the paper is to showcase the "elegance" of composing such a program with key properties of Haskell. The author makes numerous references to elegance in the paper but it is never explicitly defined. While "A History of Haskell" was an appropriate source from which to define elegance generally, "Composing Fractals" demonstrates how this definition can be applied in practice and how fractal generation is a suitable domain under which to test elegance.

Leveraging lazy evaluation is a core feature of Jones' fractal generating functions. He utilises the iterate function in combination with his next function on each point p, to generate an infinite sequence corresponding to each point. The key test as to whether p is in the mandelbrot set is whether this sequence diverges. However, this calculation is a non computable function, as it will never terminate if the list does not diverge. Jones makes use of the prelude function take to handle this, to take the first n elements of the list, and test if

they diverge. This is an approximation which gets increasingly accurate as n increases, but it is a computable function. Jones has used laziness to full effect in this program. The use of iterate allowed him to construct an infinite data structure with minimal effort, and the use of take allowed him to take the first n elements from it, making the function computable. Lazy evaluation enables this approach. A strict approach would first try to evaluate the list, then take the first n elements from it, while a lazy approach only evaluates each element as it is needed. Laziness has enabled Jones to construct fractals concisely and closely to their mathematical definition. There is no explicit mention of purity in this functional pearl, but as a result of laziness all of the functions are pure with no side effects. Again, this helps to ensure that the functions representing these mathematical concepts are truly functions "in a mathematical sense".

The use of type variables is another core feature of Jones' program. In his chooseColor function, which select a color from a palette for a point p, the color returned and those in the palettes' type is a type variable. This enables flexibility and reuse of this function for different implementations. For example this program can be used by an implementation with palettes of ASCII characters, or palettes of RGB color values. This increases the flexibility of the code without increasing the amount of code. Jones continues to use type variables to the same effect throughout the program. An Image maps each point to a type variable color. A grid is made up of points represented by the type variable a. The draw function itself is polymorphic both in its color and image types. Jones takes advantage of this flexibility by demonstrating how the same code can be leveraged by an ASCII terminal-based or an RGB graphics window-based implementation.

Jones additionally attributes the use of Haskell's "lightweight syntax" to his elegant solution. Upon inspection of the code throughout the paper, it is clear that his use of syntax is of the "declaration style" conceived in A History of Haskell. Function arguments are on the left side of the equals. There are no if statements or case statements. Where expressions are used instead of let expressions throughout. The use of declaration style results in short, understandable functions. The only move from declaration style programming is in the outmost function, rgbRender, which takes place inside the IO monad. This sequential, monadic style is unavoidable to safely perform IO, and in general should be avoided in elegant programming aside from the outmost function where IO should be performed.

Additionally, Jones attributes the elegance of the program to its compositional nature enabled by higher-order functions. He increasingly defines and makes use of higher-order functions as the abstraction level of functions increases. fracImage takes a fractal function as an argument, alongside a palette, to generate a fractal image. An image itself is a function that maps from a Point to a color. An image function is one of the arguments to sample, which samples the image at each position to generate a grid of colors to display. The draw function takes both a fractal function and a render function and composes them

together with other arguments to generate the final result. Use of higher-order functions enables elegant Haskell programming in these examples by allowing nonstandard behaviour in composing functions and arguments together concisely. A function taking and utilising another function increases the flexibility available to the user and its use alongside declaration style programming enables concise, elegant function definitions throughout.

Though Jones never explicitly defines the concept of elegance, by analysing the properties leveraged in his approach we can see that they are the same as the key properties expressed by the authors of a History of Haskell. By having a concrete definition of elegance, it is possible to assess when a piece of code is "elegant", i.e. when it follows these properties. Fractal generation is clearly a domain particularly suited to Haskell, and this suitability can be credited to the ability to elegantly program. Laziness, purity, type variables, higher-order functions and declaration style programming all enabled a concise, understandable and flexible implementation. This makes fractal generation a suitable domain in which to examine the elegance of Haskell implementations.

2.4.3 Fractal Generation using GPUs

GPGPU programming, discussed in Section 2.2, can be used to generate fractals. The use of GPUs in fractal generation can enable a substantial performance increase, which can be seen in the generation of the Mandelbrot Set and corresponding Julia Sets. Wyatt et al. explored this by implementing and comparing sequential and parallel-GPU versions [17]. Both implementations of both sets traversed the complex plane and generated each point. The sequential version looped across the points, while the GPU version virtually assigned processors to each. The implementation was written in the C programming language, utilizing NVIDIA CUDA, discussed in Section 2.2, to leverage the GPUs. The comparison ranged across image dimensions from 10-by-10 to 10,000-by-10,000, and ran on an Intel Core i7-4770K CPU and an NVIDIA GeForce GTX TITAN graphics card.

Computation time was similar between the CPU and GPU implementations for small image dimensions, but as the dimensions increased, computation time increased much more slowly for the GPU implementation. Generation of the largest image, not including the constant image display time, took longer than 10 seconds on the CPU implementation, and less than 0.0001 seconds on the GPU implementation. This enabled the creators to dynamically update the fractals in real-time as the user updated the parameters, which would not have been possible without the GPUs. Due to the strong link between the Mandelbrot Set and the Julia Set, the authors demonstrated this by allowing the user to select a point on the Mandelbrot Set, which instantly generates the corresponding Julia Set. The authors note "smooth transitions" in the Julia Set as this point is changed on the GUI.

The authors noted that this has a number of implications and contributions. The knowledge

that can be gained about the link between the Mandelbrot and Julia Set via real-time interaction can be valuable to fractal artists or graphic designers in need of specific fractal patterns. Examinations of the sets via deep zooms, possible with extreme precision due to the performance improvement, have similar implications for exploring fractals. The authors also note the educational implications, as freely varying parameters with visual feedback gives much more accessible results than standard algorithmic workings. Whether for aesthetic or mathematical reasons, users can "interact with the mathematics of chaos without the need for rigorous background work".

3 Design

3.1 Design Choices from Research

3.1.1 Elegance

The first design choice from research that I needed to make was to form a rigid definition of elegance. The design of my final implementation must maximize elegance by this definition. My research showed that four key concepts were commonly attributed to elegant programming; laziness/purity, type polymorphism, lightweight syntax and higher-order functions.

An elegant program in Haskell should be lazy and pure throughout. In contrast, a non-elegant program uses strict language features, and features which perform side effects. These language features are in Haskell because they are essential in some use cases, but their use moves away from some of the original design goals of Haskell, and reduces the elegance of the program.

An elegant program should also make use of type variables to achieve ad-hoc and polymetric polymorphism. Where possible, these types should be unconstrained, otherwise these types should belong to the least restrictive type class possible. An elegant program should make good use of style and syntax. To put more rigidity on this aspect of the definition, we can look at the concepts of expression style and declaration style programming. Both are present in Haskell, and are traditionally utilised in the same program, and sometimes in the same function definition. However it can be argued that mixing these two programming styles leads to redundant, less elegant code. In particular, expression style programming results in lengthier functions, with concepts more commonly found in imperative programs than mathematical functions. In keeping with the idea of an elegant program being made up of pure functions "in a mathematical sense", an elegant program should use declaration style programming. It should not use expression style programming, or a Monadic, sequential style of programming where possible.

In addition to laziness, purity, type polymorphism and lightweight syntax, the final property

of elegance that my research informed me about was use of higher-order functions where possible.

Guided by my research in Section 2.1, I state the following definition of elegance:

A Haskell program is elegant if it:

1. Is lazy and pure in 100% of its evaluations.
2. Makes use of type variables wherever possible to achieve type polymorphism.
3. Uses declaration style programming.
4. Uses higher-order functions wherever possible.

These key elegant properties follow directly from the research in Section 2.1 and Section 2.4.2. Each of these four properties were seen to be core concepts of Haskell in "A History of Haskell", with practical applications of them to conceive an elegant program in composing fractals [3][16]. Additionally, laziness and higher-order functions were stated as the two key properties that enabled the elegance of functional programming in "Why Functional Programming Matters" [4].

3.1.2 Performance

My final implementation must achieve a "reasonable performance threshold" to be accepted. Guided by my research in Section 2.4.3, I define the following threshold: - Generation of a 10,000 x 10,000 fractal image should take less than 0.001s.

I feel that this is a reasonable threshold as the research in Section 2.4.3 found that this is achievable in less than 0.0001s. I have made the threshold less strict than this because I expect Haskell to introduce additional overheads in comparison to a direct C implementation, such as garbage collection, conversion of Haskell representations to CUDA representations, and less control of optimisations to machine code. This threshold is still acceptable as it is much faster than the 10s figure for a CPU implementation.

3.1.3 Choice of GPGPU Library

My research in Section 2.3 investigated three GPGPU libraries in Haskell; Obsidian, Accelerate and Nikola. Out of the three libraries, Obsidian is the least appropriate for my approach. Its low-level implementation and direct mapping to CUDA kernels does not map well to elegance. While it does make use of function application, and the type system in some cases, its nature does not allow for the pure, lazy functional programming desired. Choosing Obsidian would force me into a low-level, strict, sequential implementation with thread synchronization, which eliminates it as a suitable library for my implementation.

Accelerate and Nikola both seem much more capable of the elegance that I am aiming for.

Nikola's functionality is quite a large subset of that of general Haskell, as the authors aimed to allow Haskell to be converted to Nikola with minimal syntactic overhead. Let-sharing, lamda-sharing and overloading of operations mean that Nikola's syntax is quite close to Haskell's, so similar levels of "elegance" should be achievable. However there are some restrictions with regards to recursion, bounding of arguments and memory size inference.

Accelerate's approach is much different from Obsidian. Accelerate deals with array computations which exist at a similar level of abstraction to higher-order functions in Nikola. Accelerate and Nikola were initially developed and published around the same time, but the Accelerate repositories continue to be maintained while Nikola was last updated in 2013 [18][13]. Accelerate has its own website & documentation and is available as a package in Cabal, which is not the case for Nikola [10][19]. Due to Accelerate's continued upkeep it has much more engagement than Nikola on both GitHub and the Haskell Reddit discussion board [20][21]. Accelerate's continued upkeep and use over the last decade indicates that it is of a higher standard than Nikola today and is more suited to modern hardware and software. In addition, Accelerate has two features that Nikola does not; it supports generative functions, such as replicate, and it can span multiple CUDA kernels, rather than just one. Due to the additional functionality and higher use and upkeep, I have decided to use Accelerate in my implementation. The abstraction level should offer the same potential for elegance as Nikola does, and using array computations may offer additional insights as to whether they can play a role in program elegance.

3.1.4 Choice of Fractal

Implementations of the two most commonly generated fractals; the Mandelbrot Set and the Julia Set, already exist as examples for the Accelerate library. The Julia Set has an implementation in the official Accelerate-Examples library, while the Mandelbrot Set has one both in this library and as a tutorial on the official Accelerate website [22][23]. The programs in Accelerate-Examples allow GUI interaction with the fractals, much like that discussed in Section 2.4.3. The Mandelbrot tutorial on Accelerate's website takes a different approach and statically generates an image of the fractal. Nikola's central repository also has an implementation of the Mandelbrot Set, which is adapted from the Accelerate version [13].

I have chosen to implement the Mandelbrot set, as it is the most common fractal that I encountered in my background research. In my design and implementation, I can utilize learnings from Section 2.4.1, Section 2.4.2 and Section 2.4.3. Additionally I can utilize the online resources mentioned in this Section, which gives me the strongest opportunity to answer my question with regards to performance and elegance.

The static Mandelbrot example offered on the Accelerate website compiles and runs as expected. I feel that this is a good base upon which to start from as it is relatively short (150 lines) and easy to understand. I will assess the performance and elegance of this, and update it to make it as elegant as I can.

The dynamic Mandelbrot example on cabal fails due to a dependency on the gloss-accelerate package, which fails to build. However, the code for this package is publicly accessible on GitHub. I will examine this and integrate elements to improve the functionality towards a dynamic implementation. Again I will assess the elegance and performance of this integration.

Finally, I will iterate on the integrated example to maximise the elegance according to my definition. This final iteration, and the final judgements on performance and elegance from it, will enable me to answer my research question.

4 Implementation

4.1 Machine Specifications

The implementation was designed, written and tested on the same machine over the course of the project to maintain consistency for performance. The machine was a Dell XPS 8920 Desktop, with an Intel Core i7 7700k CPU and an NVIDIA GeForce GTX 1080 GPU. The CPU is slightly faster while the GPU is slightly slower than those used by Wyatt et al. [17]. I do not expect me to have an impact on my performance with regards to the performance threshold, which I have set quite high to account for overheads also introduced through Haskell.

4.2 Implementation Challenges

A key challenge faced in the implementation stage of the project was initial setup to get the code running. The project required the installation of NVIDIA CUDA onto the machine to leverage the NVIDIA GPUs. The starting project was not a stack project, as a result the dependencies were installed using cabal. This was manageable for the static example, which required three key libraries; `accelerate` to leverage Accelerate functions, `accelerate-io` to generate the image and `accelerate-llvm-native` to leverage the GPUs.

However, the second project required many more dependencies due to the presence of boilerplate code required to generate the GUI. The Mandelbrot code and its boilerplate code was from the Accelerate-Examples project on GitHub [22], but this failed to compile due to dependency issues. As a result, the code for generating the mandelbrot set had to be brought to a new project, along with the necessary boilerplate. It was difficult to determine the minimum amount of boilerplate needed from the project, as redundant boilerplate would lead to redundant dependencies on packages. Once this code and boilerplate was brought over, the dependencies needed to be installed with cabal. This led to incompatibility issues with packages, which needed to be solved with fresh re-installs and trial and error. The setup of the project, including installation and configuration of CUDA, setup of the static project, attempted compilation of Accelerate-Examples, transfer of code with boilerplate and

installation of dependencies cost three weeks of development time. An assessment of this method and reflection on how this could have been avoided can be found in Section 5. A list of cabal dependencies required to run the dynamic project and their installation order can be found in Appendix A1.1.

Once the dynamic implementation was set up and running, there were no further issues with dependencies or boilerplate code when iterating on the elegance of the program.

4.3 Assessing Version 1: Static Image Example

The first version of the code that I assessed is the Mandelbrot Set example from the official Accelerate website. This generates a .bmp image of the Mandelbrot set in the project directory.

4.3.1 Elegance

mandelbrot

```
mandelbrot :: Int -> Int -> Int -> Float -> Complex Float -> Float
        -> Acc (Array DIM2 (Complex Float, Int))
mandelbrot screenX screenY limit radius (x0 :+ y0) width =
  A.generate (A.constant (Z :. screenY :. screenX))
             (\ix -> let z0 = complexOfPixel ix
                         zn = while (\zi -> snd zi       < constant limit
                                         && dot (fst zi) < constant radius)
                                    (\zi -> step z0 zi)
                                    (lift (z0, constant 0))
                     in
                     zn)
  where
```

The function `mandelbrot` exists at the top of the implementation. It contains multiple functions in its `where` statement; `complexOfPixel`, `dot`, `step` and `next`. These functions will be analysed under different headings for the purposes of assessing the elegance of each function individually.

This function is pure as it is at the top level a call to `A.generate`, an Accelerate operation, which are pure and do not mutate arrays [24]. However, the code is not lazy as Accelerate's implementation as a deep embedding means that the function gets processed as an abstract syntax tree which gets converted to CUDA code, which is then compiled and run to return the result to Haskell. This compilation and running is not lazy, and executes the GPU

kernels when `A.generate` is called.

There is no use of type variables in this function. All types are concrete types, meaning is no parametric or ad-hoc polymorphism present.

The function is not written in declaration style, instead in a mix of declaration style and expression style. Function arguments are on the left hand side and a `where` clause is used, but lambda abstractions and a `let` expression are also present.

The function makes use of the higher-order Accelerate function `while`, which takes functions as its first two arguments. But `mandelbrot` itself is not a higher-order function, as it takes no functions as arguments.

complexOfPixel

```
-- Convert the given array index, representing a pixel in the final
-- image, into the corresponding point on the complex plane.
complexOfPixel :: Exp DIM2 -> Exp (Complex Float)
complexOfPixel (unlift -> Z :. y :. x) =
  let
      height = P.fromIntegral screenY / P.fromIntegral screenX * width
      xmin   = x0 - width  / 2
      ymin   = y0 - height / 2
      re     = constant xmin + (fromIntegral x * constant width)
                             / constant (P.fromIntegral screenX)
      im     = constant ymin + (fromIntegral y * constant height)
                             / constant (P.fromIntegral screenY)
  in
  lift (re :+ im)
```

`complexOfPixel` is called from `mandelbrot`, so forms part of the abstract syntax tree formed from the implementation due to the deep embedding. As a result it has the same relationship to laziness and purity. The code is executed purely and strictly on the GPU. The use of `Exp` ensures that this function gets evaluated as a deep embedding, meaning that it cannot be leveraged by lazy code.

Again there is no use of type variables, so there is zero ad-hoc or parametric polymorphism.

The function is not written in declaration style, instead in a mix of declaration and expression style. Function arguments are on the left hand side, but the function is structured as a `let` expression.

The function is not, and does not make use of, a higher-order function.

dot

```
-- Divergence condition
dot :: Exp (Complex Float) -> Exp Float
dot (unlift -> x :+ y) = x*x + y*y
```

Again `dot` is called utilizing the deep embedding to form an AST, so is pure, but strict. `Exp` in the function signature enforces this.

There is no use of type variables, so there is zero ad-hoc or parametric polymorphism.

The function is written in declaration style. Function arguments are on the left hand side and no expression style constructs are used.

The function is not, and does not make use of, a higher-order function.

step

```
-- Take a single step of the recurrence relation
step :: Exp (Complex Float) -> Exp (Complex Float, Int)
            -> Exp (Complex Float, Int)
step c (unlift -> (z, i)) = lift (next c z, i + constant 1)
```

Like `dot`, `step` is pure, but strict. It has no ad-hoc or parametric polymorphism. It is written in declaration style, with function arguments on the left hand side and no expression style constructs. It is not, and does not make use of, a higher order function.

next

```
next :: Exp (Complex Float) -> Exp (Complex Float) -> Exp (Complex Float)
next c z = c + z * z
```

Like `dot` and `step`, `next` is pure, but strict. It has no ad-hoc or parametric polymorphism. It is written in declaration style, with function arguments on the left hand side and no expression style constructs. It is not, and does not make use of, a higher order function.

escapeToColour

```
-- Convert the iteration count on escape to a colour.
escapeToColour :: Int -> Exp (Complex Float, Int) -> Exp Colour
escapeToColour limit (unlift -> (z, n)) =
  if n == constant limit
    then black
    else ultra (toFloating ix / toFloating points)
```

```
      where
        mag     = magnitude z
        smooth  = logBase 2 (logBase 2 mag)
        ix      = truncate (sqrt (toFloating n + 1 - smooth)
                    * scale + shift) `mod` points
        scale   = 256
        shift   = 1664
        points  = 2048 :: Exp Int
```

`complexOfPixel` forms part of the abstract syntax tree rooted at `mandelbrot` due to Accelerate's deep embedding, so forms part of the pure, but strict, code. There is no ad-hoc or parametric polymorphism. It is written in a merge of declaration style and expression style programming, with function arguments on the left hand side and a `where` clause used, but also an `if` statement. It is not, and does not make use of, a higher order function.

ultra

```
-- Pick a nice colour, given a number in the range [0,1].
ultra :: Exp Float -> Exp Colour
ultra p =
  if p <= p1 then interp (p0,p1) (c0,c1) (m0,m1) p else
  if p <= p2 then interp (p1,p2) (c1,c2) (m1,m2) p else
  if p <= p3 then interp (p2,p3) (c2,c3) (m2,m3) p else
  if p <= p4 then interp (p3,p4) (c3,c4) (m3,m4) p else
                  interp (p4,p5) (c4,c5) (m4,m5) p
  where
    p0 = 0.0;    c0 = rgb8 0   7   100; m0 = (0.7843138, 2.4509804,  2.52451)
    p1 = 0.16;   c1 = rgb8 32  107 203; m1 = (1.93816,   2.341629,   1.6544118)
    p2 = 0.42;   c2 = rgb8 237 255 255; m2 = (1.7046283, 0.0,        0.0)
    p3 = 0.6425; c3 = rgb8 255 170 0;   m3 = (0.0,       -2.2812111, 0.0)
    p4 = 0.8575; c4 = rgb8 0   2   0;   m4 = (0.0,       0.0,        0.0)
    p5 = 1.0;    c5 = c0;               m5 = m0
    -- interpolate each of the RGB components
    interp (x0,x1) (y0,y1) ((mr0,mg0,mb0),(mr1,mg1,mb1)) x =
      let
          RGB r0 g0 b0 = unlift y0 :: RGB (Exp Float)
          RGB r1 g1 b1 = unlift y1 :: RGB (Exp Float)
      in
      rgb (cubic (x0,x1) (r0,r1) (mr0,mr1) x)
          (cubic (x0,x1) (g0,g1) (mg0,mg1) x)
```

```
          (cubic (x0,x1) (b0,b1) (mb0,mb1) x)
```

`ultra` forms part of the deep embedding's AST due to its call from `complexFromPixel` and its `Exp` filled type signature, so it is pure but strict. There is no ad-hoc or parametric polymorphism. It is written in a merge of declaration style and expression style programming, with function arguments on the left hand side and a `where` clause used, but also lambda abstractions, a `let` expression and `if` statements used. It is not, and does not make use of, a higher order function.

linear

```
-- linear interpolation
linear :: (Exp Float, Exp Float) -> (Exp Float, Exp Float) -> Exp Float
            -> Exp Float
linear (x0,x1) (y0,y1) x = y0 + (x - x0) * (y1 - y0) / (x1 - x0)
```

`linear` forms part of the deep embedding's AST, so it is pure but strict. There is no ad-hoc or parametric polymorphism. It is written in declaration style, with function arguments on the left hand side and no expression-style constructs used. It is not, and does not make use of, a higher order function.

main

```
main :: P.IO ()
main =
  let
      width   = 10000
      height  = 10000
      limit   = 1000
      radius  = 256
      img = A.map packRGB
          $ A.map (escapeToColour limit)
          $ mandelbrot width height limit radius ((-0.7) :+ 0) 3.067
  in
  writeImageToBMP "mandelbrot.bmp" (run img)
```

`main` is the outmost function of the program, written in the `IO` monad. The format of the function is pure and lazy. The `IO` monad preserves purity by design. The function does not use the `do` construct to force sequencing of operations, so the code is lazy in general. Haskell lazily evaluates the code until the last line. `writeImageToBMP` processes its argument at its beginning, so requires the evaluation of `run img`. This forces the evaluation of `img`, which forms the abstract syntax tree. `run` then runs all of the steps between

converting the syntax tree to getting the end GPU result strictly, and returns its answer to Haskell, to continue to work with lazily. So we can say that the code is lazy and pure, with one call to a strict function.

`main` contains no type polymorphism in its type signature. Each number variable is initialised without a concrete type, which leaves potential for type polymorphism in them, but `mandelbrot`'s lack of polymorphism forces each of their evaluations to a particular type.

The function is not declarative style, instead it is expression style, composed of a `let` statement.

The function does make use of higher order functions, in particular the Accelerate `map` function in two different contexts.

Summary of Results

The table below signifies whether a particular function had a high, middling or low level of an elegant property:

Function	Lazy/Pure	Type Polymorphism	Declaration Style	Higher-Order
mandelbrot	mid	low	mid	mid
complexOfPixel	mid	low	mid	low
dot	mid	low	high	low
step	mid	low	high	low
next	mid	low	high	low
escapeToColor	mid	low	mid	low
ultra	mid	low	mid	low
linear	mid	low	high	low
main	high	low	low	mid

Table 4.1: Assessment of Elegance for Static Mandelbrot Example.

Conclusion

Analysis of the elegance of the mandelbrot code showed similar patterns throughout.

With regards to laziness and purity, the code was only found to be lazy at the top level, which executed in the host language Haskell rather than the embedded language Accelerate. Once execution fell to the function `run`, it was handed to Accelerate's execution agent, which formed the AST, compiled to CUDA code, executed the underlying kernels and returned the result to Haskell strictly. This strict execution encompassed the majority of the code, including all of the GPU code. In contrast, purity was preserved, through both the IO

monad and Accelerate execution engine, which computes its results without side effects.

There was no use of type variables or type classes in the code to achieve either ad-hoc or parametric polymorphism. The code was not elegant in this sense.

Overall, the code was written in a mix of declaration style and expression style constructs, with some shorter, mathematical functions being written completely in declaration style.

In general, the code made very little use of higher-order functions. None of the implemented functions were higher-order, but `mandelbrot` and `main` utilized some, in particular use of Accelerate's higher-order functions, which utilized the GPU themselves.

Overall, the elegance of the program was low, particularly in the areas of laziness, type polymorphism and and higher-order functions.

4.4 Assessing Version 2: Dynamic GUI Example

4.4.1 Elegance

mandelbrot

```
mandelbrot :: forall a. (Num a, RealFloat a, FromIntegral Int a)
    => Int -> Int -> Acc (Scalar a) -> Acc (Scalar a)
    -> Acc (Scalar a) -> Acc (Scalar Int32) -> Acc (Scalar a)
    -> Acc (Array DIM2 (Complex a, Int32))
mandelbrot screenX screenY (the -> x0) (the -> y0) (the -> width)
        (the -> limit) (the -> radius) =
  A.generate (A.constant (Z :. screenY :. screenX))
             (\ix -> let z0 = complexOfPixel ix
                         zn = while (\zi -> snd zi       < limit
                                         && dot (fst zi) < radius)
                                    (\zi -> step z0 zi)
                                    (lift (z0, constant 0))
                     in
                     zn)
  where
```

Like its equivalent in Section 4.3.1, `mandelbrot`'s evaluation is done by the host language Accelerate, which forms the AST, evaluates it, translates it to CUDA, compiles the kernels, runs them, and returns the result to Haskell. This is a strict process done at the top level

call to `run`. For the same reasons, the code is also still pure.

The syntax of this function is mostly unchanged, aside from removal of the calls to `constant`, which are not needed due to the new types of `limit` and `radius`. As a result the function is still a merge of declaration style and expression style, with function arguments and a `where` clause, but also lambda abstractions and a `let` expression. It also still makes use of two Accelerate higher order functions, `generate` and `while`.

In terms of type polymorphism, we do see changes in this iteration of the code. The type signature replaces instances of `Float` with `Acc (Scalar a)`. This is for performance reasons due to the dynamic nature of the code [24]. Defining these inputs as Accelerate arrays, represented with the type constructor `Acc`, ensures that when the code is compiled to CUDA, they are represented as GPU arrays, rather than as part of the kernel. If they were part of the kernels, then every time they change, the kernel would need to be recompiled. This would reduce the performance benefits gained by leveraging GPUs. Memoisation in Accelerate ensures that the CUDA kernels are not recompiled if they don't need to be, e.g. when the dynamic parameters are represented as GPU arrays. The `Scalar` type constructor simply signifies to Accelerate that the Accelerate array contains a scalar value rather than being a multi-element array.

Despite the restriction to the `Acc` and `Scalar` constructs for performance, this version of `mandelbrot` is more polymorphic in its type signature than that in Section 4.3.1. The `Float` arguments are replaced with `Acc (Scalar a)`, with the restrictions `Num a`, `RealFloat a` and `FromIntegral a`. So the function operates on any type `a` that fits these restrictions, enclosed in the `Acc` and `Scalar` type constructors. These type classes are Accelerate type classes, which, aside from the expected numerical restrictions, enforce that `a` is a member of the `Elt` type class. The `Elt` type class is the class of those that are allowed inside an Accelerate embedded expression `Exp`. This restricts `a` to a smaller subset than similar Prelude type classes would.

Overall, we see no improvement in `mandelbrot` in terms of elegance, apart from in type polymorphism. The introduction of the type variable `a` enabled some ad-hoc polymorphism. There are still some restrictions to the polymorphism though, with the enforcement of `Acc` arrays and type class restrictions, in particular of the `Elt` class.

complexOfPixel

```
-- Convert the given array index, representing a pixel in the final
-- image, into the corresponding point on the complex plane.
complexOfPixel :: Exp DIM2 -> Exp (Complex a)
complexOfPixel (unlift -> Z :. y :. x) =
  let
```

```
        height = P.fromIntegral screenY / P.fromIntegral screenX * width
        xmin   = x0 - width  / 2
        ymin   = y0 - height / 2
        re     = xmin + (fromIntegral x * width)
                        / fromIntegral (constant screenX)
        im     = ymin + (fromIntegral y * height)
                        / fromIntegral (constant screenY)
    in
    lift (re :+ im)
```

`complexOfPixel` contains very little changes in the value level code, with similar small changes in using the `constant` function due to changes in the function's type signature. As a result there are no changes in the laziness, purity, programming style or level of higher-order functions.

There is an improvement present in the type polymorphism, as the return type `Exp (Complex Float)` is abstracted to `Exp (Complex a)`, with a inheriting its type class restrictions from `mandelbrot`. `a` is wrapped in the `Complex` type constructor to signify a complex number in Accelerate. As this function is a calculation of a complex number, it is to be expected that it would return a complex type. It is wrapped in another type constructor, `Exp`, to represent an embedded expression that forms part of the AST.

Overall, there is no improvement in the elegance of `complexOfPixel`, apart from in type polymorphism. The introduction of a enabled some ad-hoc polymorphism, but there are still some polymorphism restrictions including representation as embedded `Exp` types and restriction to Accelerate type classes which force the `Elt` restriction.

dot, step, next

```
-- Divergence condition
dot :: Exp (Complex a) -> Exp a
dot (unlift -> x :+ y) = x*x + y*y

-- Take a single step of the recurrence relation
step :: Exp (Complex a) -> Exp (Complex a, Int32) -> Exp (Complex a, Int32)
step c (unlift -> (z, i)) = lift (next c z, i + constant 1)

next :: Exp (Complex a) -> Exp (Complex a) -> Exp (Complex a)
next c z = c + z * z
```

There is no change in the value level code for `dot`, `step` and `next`. As a result, most of the elegance remains the same. The code is strict, but pure. There is no use of higher-order

functions. Each of these functions remain in full declaration style, rather than a mix of declaration and expression style like most of the others.

Again there are changes at the type level, with each instance of `Float` abstracted to the type variable `a`, inheriting the type class restrictions from `mandelbrot`. They remain within the `Exp` constructors which form the AST.

Overall, there is no improvement in the elegance of these functions apart from in terms of type polymorphism. Introduction of the type variable `a` enabled some ad-hoc polymorphism, again restricted by Accelerate constructs including `Exp` and `Elt`.

escapeToColour

```
escapeToColour :: (RealFloat a, ToFloating Int32 a)
    => Acc (Scalar Int32) -> Exp (Complex a, Int32) -> Exp Word32
escapeToColour (the -> limit) (unlift -> (z, n)) =
  if n == limit
    then packRGB black
    else packRGB \$ ultra (toFloating ix / toFloating points)
      where
        mag     = magnitude z
        smooth  = logBase 2 (logBase 2 mag)
        ix      = truncate (sqrt (toFloating n + 1 - smooth)
                    * scale + shift) `mod` points
        scale   = 256
        shift   = 1664
        points  = 2048 :: Exp Int
```

The type signature of `escapeToColour` is quite different. The first argument representing the cutoff limit for colouring the mandelbrot set has been wrapped in the `Acc` and `Scalar` constructs. This is for the same reason as in `mandelbrot`. In Section 4.3.1, this parameter was set once when running the program, to generate the static image. In this example, it is dynamic, as the user can alter the limit in real time. As a result, for performance reasons, it must be an Accelerate array which exists on the GPU and outside of the kernel. There is one improvement in type polymorphism as `z` is abstracted from `Float` to `a`, with Accelerate type class restrictions.

There are few value level changes, aside from those forced by changes in the type signature, e.g. use of the `the` function to unpack the limit from its Accelerate array. It is still strict and pure, forming part of the Accelerate AST enabled by the `Exp`s in the type signature. It is still a mix of declaration and expression style, with function arguments on the left hand side and a `where` clause used, but also an `if` statement. No higher order functions are used.

Overall there is no improvement in elegance apart from the introduction of type variable `a`, restricted by Accelerate type classes.

ultra, linear and main

There are no type level or value level changes in `ultra` or `linear`. `main` has been abstracted to the boilerplate code of the dynamic example, which leverages other libraries such as Accelerate Gloss to render the GUI window.

Summary of Results

Function	Lazy/Pure	Type Polymorphism	Declaration Style	Higher-Order
mandelbrot	mid	mid	mid	mid
complexOfPixel	mid	mid	mid	low
dot	mid	mid	high	low
step	mid	mid	high	low
next	mid	mid	high	low
escapeToColor	mid	mid	mid	low
ultra	mid	mid	mid	low
linear	mid	mid	high	low

Table 4.2: Assessment of Elegance for Dynamic Mandelbrot Example.

Conclusion

Overall, the move to dynamic code caused little change to the mandelbrot code. This demonstrates the portability and modularity of the code, certainly an end goal of elegance as found in Section 2.1. As a result, there were no improvements in the elegance of the value level code in terms of laziness, programming style or higher order functions.

The main change in the mandelbrot code to allow for dynamic GUI interaction was changes to the type signature of functions. At the top level, dynamic variables were converted from concrete types to `Acc` arrays enclosing `Scalars`. While the author could have simply wrapped these around the concrete types, they instead wrapped ad-hoc type variables, with Accelerate type class restrictions enabling them to be propagated on. The improvements in type polymorphism at the top level propagate down through the code, with many of the embedded `Exp` types containing type variables with the same restrictions. This showed that the level of elegance possible for the implementation was not necessarily fixed, and could be improved.

While type polymorphism was improved, it was still restricted in this implementation. Some restrictions were analogous to those found in non-GPGPU programming, but others were

introduced through GPU concepts, particularly the `Acc` input arrays, deeply embedded `Exp` types, and restrictions to the `Elt` type class. Fully elegant code from a type polymorphism perspective would remove these restrictions specifically introduced by GPU concepts, and be restricted by non-GPU type classes in the same way as regular Haskell code. This abstraction was something that I would attempt when iterating the elegance of the implementation in terms of polymorphism.

4.5 Improving Elegance

4.5.1 Update to Declaration Style

The first property that I chose to iterate on was the style and syntax of the program. To update to declaration style I needed to update each of the following syntactic constructs:

- `let` expressions to `where` clauses
- Lambda abstractions to named functions with arguments on the left hand side
- `case` expressions to pattern matching in function definitions
- `if` expressions to guards on function definitions

Forcing these syntactic construct changes would lead to a full conversion to declaration style; away from longer expressions and towards shorter equations. The goal here was to see if Haskell GPGPU programming, specifically Accelerate programming, restricted the ability to satisfy this elegant property.

mandelbrot

The `mandelbrot` function was composed of both declaration style and expression style constructs, with lambda abstractions and `let` expressions being of the latter. `mandelbrot` was a call to the higher level function `A.generate`, whose second argument was both lambda abstraction and a let expression. I converted the lambda abstraction to a named function, `mandelLoop`, and wrote its implementation in a `where` clause. I also converted `mandelLoop`'s `let` statement to a `where` clause. There were two more lambda abstractions passed as arguments to the Accelerate function `while` (analogous to a loop construct in iterative programming). These represented the condition and updation constructs of a loop, so I named them as such and specified their implementation in the inner `where` clause.

These changes fulfilled the syntactic constructs of declaration style, but I made one more change, moving the `complexOfPixel`, `dot`, `step` and `next` functions from the existing `where` clause, to the global scope. I did this because declaration style programming favours

shorter equations over longer expressions, and having these functions in the general scope felt more in keeping with that. In keeping with the idea of modularity linked with declaration style, laziness and higher-order functions, they seemed more appropriate as modules at a global scope, especially considering their mathematical nature. This had implications on the type signature of each of these functions, as type class restrictions needed to be listed explicitly on each instead of inherited from `mandelbrot`. This later became useful to more clearly assess the type polymorphism when attempting to improve it. Another implication of this was that additional arguments, `screenX`, `screenY`, `x0`, `y0` and `width`, needed to be passed to `complexOfPixel` instead of inherited from `mandelbrot`.

```
mandelbrot
    :: forall a. (Num a, RealFloat a, FromIntegral Int a)
    => Int -> Int -> Acc (Scalar a) -> Acc (Scalar a)
    -> Acc (Scalar a) -> Acc (Scalar Int32) -> Acc (Scalar a)
    -> Acc (Array DIM2 (Complex a, Int32))
mandelbrot screenX screenY (the -> x0) (the -> y0) (the -> width)
    (the -> limit) (the -> radius) =
A.generate (A.constant (Z :. screenY :. screenX)) mandelLoop
where
    mandelLoop :: Exp DIM2 -> Exp (Complex a, Int32)
    mandelLoop ix = while condition updation initialization
      where
        condition zi = snd zi < limit && dot (fst zi) < radius
        updation zi = step z0 zi
        initialization = lift (z0, constant 0)
        z0 = complexOfPixel ix screenX screenY x0 y0 width
```

complexOfPixel

In `complexOfPixel` I first updated the arguments and type signature due to the additional arguments brought over from `mandelbrot`. I also brought over the type restrictions on `a` from `mandelbrot`. The only change to declaration style was updating the `let` statement to a `where` clause, which was a straightforward conversion.

```
complexOfPixel :: forall a. (Num a, RealFloat a, FromIntegral Int a)
    => Exp DIM2 -> Int -> Int -> Exp a -> Exp a -> Exp a -> Exp (Complex a)
complexOfPixel (unlift -> Z :. y :. x) screenX screenY x0 y0 width =
  lift (re :+ im)
    where
      height = P.fromIntegral screenY / P.fromIntegral screenX * width
      xmin   = x0 - width  / 2
```

```
    ymin    = y0 - height / 2
    re      = xmin + fromIntegral x * width
                        / fromIntegral (constant screenX)
    im      = ymin + fromIntegral y * height
                        / fromIntegral (constant screenY)
```

dot, step, next

The functions `dot`, `step` and `next` were already written in declaration style as shown in my analysis. The only change required was to copy the type restrictions from `mandelbrot` on `a`.

```
dot :: forall a. (Num a, RealFloat a, FromIntegral Int a)
    => Exp (Complex a) -> Exp a
dot (unlift -> x :+ y) = x*x + y*y

step :: forall a. (Num a, RealFloat a, FromIntegral Int a)
    => Exp (Complex a) -> Exp (Complex a, Int32) -> Exp (Complex a, Int32)
step c (unlift -> (z, i)) = lift (next c z, i + constant 1)

next :: forall a. (Num a, RealFloat a, FromIntegral Int a)
    => Exp (Complex a) -> Exp (Complex a) -> Exp (Complex a)
next c z = c + z * z
```

escapeToColour

The only expression style construct in `escapeToColour` was an `if then else` statement, which tested whether the iteration depth had reached its limit before calculating the colour. As per declaration style programming, these should be changed to guarded expressions. However, this was not possible, as the `if then else` construct here was not Haskell's, which is analogous to guards, but instead used `RebindableSyntax` to overload the construct to represent Accelerate's `?` function. I replaced the construct explicitly with the `?` function to illustrate this. `?` is of the following type signature:

```
(?) :: Elt t => Exp Bool -> (Exp t, Exp t) -> Exp t
```

As this function was not accounted for in "History of Haskell"'s distinction between expression style and declaration style programming, it required judgment to decide which style this construct fit into. While updating to `?` was a move away from expression style, it was not close enough to declaration style. Declaration style favours short equations using pattern matching or guarded expressions. Ultimately the `?` expression is analogous to the `if then else` construct, as evidenced by the overloaded syntax. This `?` function determines its outcome based on an `Exp Bool`, which is Accelerate's expression of a `Bool`, therefore

should be treated as such. In elegant Haskell we would expect this `Bool` to be processable by guards. This showed up one limitation of declaration style Accelerate programming; we are restricted to expression style checking of the `Bool` rather than checking by guards.

```
escapeToColour
    :: (RealFloat a, ToFloating Int32 a)
    => Acc (Scalar Int32)
    -> Exp (Complex a, Int32)
    -> Exp Word32
escapeToColour (the -> limit) (unlift -> (z, n)) =
n == limit ? (packRGB black,
        packRGB $ ultra (toFloating ix / toFloating points))
      where
        mag     = magnitude z
        smooth  = logBase 2 (logBase 2 mag)
        ix      = truncate (sqrt (toFloating n + 1 - smooth)
                        * scale + shift) `mod` points
        scale   = 256
        shift   = 1664
        points  = 2048 :: Exp Int
```

ultra

`ultra` contained nested uses of the `if then else` construct. As with `escapeToColour`, I replaced these with `?` to remove this construct explicitly but was limited in full conversion to declaration style. I also moved the sub-expression `interp` to global scope, for the same reasons as moving |mandelbrot|'s sub-expressions listed above.

```
ultra :: Exp Float -> Exp Colour
ultra p =
  p <= p1 ? (
    interp (p0,p1) (c0,c1) p,
    p <= p2 ? (
      interp (p1,p2) (c1,c2) p,
      p <= p3 ? (
        interp (p2,p3) (c2,c3) p,
        p <= p4 ? (
          interp (p3,p4) (c3,c4) p,
                  interp (p4,p5) (c4,c5) p))))
  where
    p0 = 0.0     ; c0 = rgb8 0   7   100
```

```
    p1 = 0.16    ; c1 = rgb8 32  107 203
    p2 = 0.42    ; c2 = rgb8 237 255 255
    p3 = 0.6425  ; c3 = rgb8 255 170 0
    p4 = 0.8575  ; c4 = rgb8 0   2   0
    p5 = 1.0     ; c5 = c0
```

interp

`interp` contained a `let` statement, which I updated to a `where` clause.

```
-- interpolate each of the RGB components
interp :: (Exp Float, Exp Float)-> (Exp Colour , Exp Colour)
            -> Exp Float -> Exp Colour
interp (x0,x1) (y0,y1) x =
        rgb (linear (x0,x1) (r0,r1) x)
      (linear (x0,x1) (g0,g1) x)
      (linear (x0,x1) (b0,b1) x)
  where
      RGB r0 g0 b0 = unlift y0 :: RGB (Exp Float)
      RGB r1 g1 b1 = unlift y1 :: RGB (Exp Float)
```

linear

`linear` was already a declaration style expression, so did not need to be updated.

Conclusion

Overall, the Accelerate mandelbrot code was highly portable to declaration style. Accelerate was well suited to declaration style, as evidenced by the fact that some functions were already written in it, and some others could be converted either through changing syntactic constructs or moving functions to global scope. The flexibility of the syntax meant that Accelerate could possibly fit other definitions in relations to style and syntax, as well as my own. This made the finding more generalisable.

However, the code was restricted in terms of the flexibility of `if` statements, which could not be swapped to guard expressions due to their underlying representation. This showed a real limitation on the syntax caused by Accelerate GPU constructs. So although the syntax was quite flexible, this showed up a limitation on that flexibility, meaning that the code could not be made fully elegant in the area of style/syntax.

4.5.2 Improving Type Polymorphism

The next elegant property that I chose to improve was the type polymorphism of the program. Based on my research, these conversions would fall under three broad categories:

- Abstracting concrete types to type variables, where possible.
- Making type variables parametrically polymorphic, i.e. unconstrained, where possible.
- Where type variables must be ad-hoc polymorphic, i.e. constrained by type classes, ensuring the fewest constraints on them possible are applied.

These abstractions would make functions more general, increasing their modularity and portability. Type restrictions were expected throughout, particularly in the use of type classes, Haskell's "most distinctive characteristic" [3]. The goal was to see whether GPGPU programming restricted types in a way that non-GPGPU programming would not, just as it had restricted the ability to use guards on GPU `Bool` expressions.

mandelbrot

The first types that I looked to abstract were `screenX` and `screenY`'s `Int` types. They could not be abstracted further, as they were used to specify the array dimensions passed to `A.generate`. Accelerate array dimensions must be specified with `Int`s. This is a reasonable requirement as an array dimension cannot traditionally be of a floating point length.

The next abstraction that I attempted was the abstraction of the `Acc (Scalar a)` types. These were types bounded by the type class restrictions on `a` (e.g. numeric types), wrapped in the `Acc` and `Scalar` type constructors. As the code operated on `a` itself, a possible abstraction was to change each occurrence of type `Acc (Scalar a)` to just `a`. It is possible to remove `Acc` and `Scalar`, as seen in the static example analysed earlier. However, this has implications on the performance. Representing the arguments as `Acc (Scalar a)` ensures that the compiled CUDA code treats them as GPU arrays. Otherwise, they are represented as elements within the compiled CUDA code, meaning that each time they change, the CUDA code must be recompiled [24]. These parameters change each time the user interacts with the GUI, which would mean recompilation of the CUDA kernel on every interaction, which would drastically reduce performance. So to effectively leverage GPUs in Accelerate, we are restricted in the types of these arguments.

In terms of restrictions on `a` itself, `a` must be of the `RealFloat` and `FromIntegral` type classes as these restrictions are required by `complexByPixel`. The type class restriction `Num a` however, is redundant so can be removed.

The type `Acc (Scalar Int32)` can be abstracted further to `Acc (Scalar b)`. `b` must

stay inside these type constructors for the same performance reasons as `a`. `b` must be of the `Ord` type class so that it can be compared in the `mandelLoop` condition, which is a reasonable restriction that would also be present in non-GPU code. Both `b` and `Exp b` are tied to the `P.Num` type class (the prelude's Num class, as opposed to Accelerate's), due to a restriction of `step`. This is more restrictive than traditional non-GPU code, as we need the restriciton on both `b` when it is wrapped in a GPU expression, and unwrapped from it.

```
mandelbrot
    :: forall a b. (RealFloat a, FromIntegral Int a,
        Ord b, P.Num b, P.Num (Exp b))
    => Int -> Int -> Acc (Scalar a) -> Acc (Scalar a)
    -> Acc (Scalar a) -> Acc (Scalar b) -> Acc (Scalar a)
    -> Acc (Array DIM2 (Complex a, b))
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- Each dynamic type must be `Acc (Scalar a)` instead of just `a`.
- `a` and `b` must be part of Accelerate type classes, which enforce membership of the `Elt` type class.
- Both `b` and its GPU expression must be members of `P.Num`.

complexOfPixel

`complexOfPixel`'s first argument, of type `Exp DIM2`, represents a two-dimensional GPU expression. This cannot be abstracted further as it is passed down from `mandelbrot` representing the 2 dimensional array in which to store the result. The dimensions of the array must match so that `complexOfPixel` can correctly index into the array.

The next two arguments, of type `Int`, represent the screen width and height. These arguments were earlier required by `mandelbrot` to be of type `Int` when constructing the 2D array.

The final three arguments; `x0`, `y0` and `width`, are of type `Exp a`. These represent embedded scalar GPU expressions, which have been extracted from the scalar arrays in `mandelbrot`. This is required for these values to be represented as GPU expressions in the backend. We cannot extract a value `a` from an expression `Exp a`, as this represents running the GPU expression at the top level. The whole idea here is that the entire GPU expression is compiled and run at the top level, rather than executing many smaller GPU expressions within Haskell code, causing overheads of host-device memory transfer. So these values must remain as `Exp a`.

The type restriction `Num a` is redundant and can be removed. `a` is used in calculations, and

the restriction `RealFloat a` must remain so that these can be floating point calculations. Additionally, `FromIntegral Int a` must remain so that the `Int` values `x` and `y` can be brought to type `a` in these calculations. It is important to note that these type classes are from Accelerate rather than the prelude, and as well as the expected restrictions they also enforce that `a` be of the type class `Elt`, meaning a type that can be enclosed in a GPU expression. This is an extra restriction on the type `a` that would not be present in non-GPU code.

The result of `complexFromPixel` is of type `Exp (Complex a)`. We would expect this value to be a complex type in non-GPU code (as the function is calculating a complex number), but we see the additional restrictions on `a` and the `Exp` type constructor that we saw for the function arguments.

```
complexOfPixel :: forall a. (RealFloat a, FromIntegral Int a)
    => Exp DIM2 -> Int -> Int -> Exp a -> Exp a -> Exp a -> Exp (Complex a)
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- Each dynamic type must be `Exp a` instead of just `a`.
- `a` must be part of Accelerate type classes, which enforce membership of the `Elt` type class.

dot

With `dot`, we see that the declaration style improvements enable improvements in type polymorphism. The type restrictions `RealFloat a` and `FromIntegral Int a`, brought over from `mandelbrot`, are not needed so can be removed. This increases the polymorphism of `a`. `Num a` is still required, which is shorthand for `(Elt a, P.Num (Exp a))`. I.e. `a` must be able to be embedded in a GPU expression, and that GPU expression must be of the prelude's `Num` class. The restriction of `P.Num` is something that we would see in non-GPU code, but again there are extra restrictions due to it being an embedded GPU expression. The argument to the function is a `Complex` GPU expression. The complex element of this is analogous to a restriction that we would see in non-GPU code, as this function is designed to process a complex number.

```
dot :: (Num a) => Exp (Complex a) -> Exp a
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- Each dynamic type must be `Exp a` instead of just `a`.
- `a` must be part of Accelerate type classes, which enforce membership of the `Elt` type class.

step

Again we see that declaration style improvements have enabled improvements in type polymorphism, as the `RealFloat a` and `FromIntegral Int a` restrictions can be removed. Additionally, each instance of `Complex a` can be abstracted to `a`, as the restriction to complex numbers is not required in this function, so we can make it more general. We still require that `a` is of the `Num` class and enclosed in an `Exp` constructor for the same reasons as in `dot`.

Additionally the `Int32` type is too restrictive and can be abstracted to type `b`. `b` must be of the `Elt` class so that it can be enclosed in a GPU expression, and its expression must be of the `P.Num` class due to requirements in mandelbrot, so this is captured with `Num b`. Additionally, `b` must also be part of `P.Num`, as it is unlifted from the GPU expression and used in a calculation before being lifted back in.

```
step :: (Num a, Num b, P.Num b) => Exp a -> Exp (a, b) -> Exp (a, b)
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- Each dynamic type must be enclosed in a GPU `Exp`.
- `a` and `b` must be part of Accelerate type classes, which enforce membership of the `Elt` type class.
- Both `b` and its GPU expression must be members of `P.Num`.

next

This type signature can be made much more polymorphic. The body of the function is simple linear arithmetic, utilising the `(+)` and `(*)` constructs. The type class restriction required for these operations is that `a` belongs to `P.Num`. So each of the other restrictions, including both the `Complex` and `Exp` constructors, can be removed.

```
next :: (P.Num a) => a -> a -> a
```

GPU programming does not restrict the type polymorphism of this function. It has been generalised and can be utilized by other functions, including non-GPU ones. In terms of type polymorphism, it is an elegant function.

escapeToColour

`escapeToColour`'s type signature cannot be made more polymorphic.

`escapeToColour`'s first argument is of type `escapeToColour`. This is restricted to the `Acc` and `Scalar` constructors for the same performance reasons as `mandelbrot`, as it is also a dynamic parameter to a function called at the top level.

The second argument is of type `Exp (Complex a, Int32)`. This argument needs to be of type `Exp (Complex a, b)` to accept the return type of `mandelbrot` in the code, which itself is restricted by the return type of `complexOfPixels`.

In both arguments the wrapped values are restricted to `Int32` as they are used in `Int32` calculations.

The return type is `Exp Word32`. This cannot be abstracted further as it is the type returned by `packRGB`, and is propagated higher to render the result.

```
escapeToColour :: (RealFloat a, ToFloating Int32 a)
    => Acc (Scalar Int32) -> Exp (Complex a, Int32) -> Exp Word32
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- The dynamic type must be `Acc (Scalar a)` instead of just `a`.
- The second type is restricted to a GPU `Exp`.
- The number types are restricted to `Int32` due to the library implementation.
- The return type is restricted to `Word32` due to the library implementation.

ultra

The type signature for `ultra` cannot be made more polymorphic due to restrictions caused by the `interp` function. The argument of type `Exp Float` cannot be abstracted further as it needs to be passed to the restricting `interp` function, and the return type `Exp Colour` cannot be abstracted further as it is returned by the restricting `interp` function.

```
ultra :: Exp Float -> Exp Colour
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- The argument and return type are restricted by GPU restrictions in `interp`.

interp

The type signature for `interp` cannot be made more polymorphic.

The function takes a pair of `Exp Colours`, which cannot be abstracted further as they are operated on by Accelerate's `unlift` function, unlifted into the `RGB` constructor. Unlifting into the `RGB` constructor requires type `Exp Colour`.

The function also takes three `Exp Floats`, which cannot be abstracted further as they are passed to `linear` alongside other `Exp Floats` unlifted from the colours, and `linear` requires that all arguments be of the same type to perform its calculation.

```
interp :: (Exp Float, Exp Float)
    -> (Exp Colour , Exp Colour) -> Exp Float -> Exp Colour
```

In summary, we see the following additional restrictions on type polymorphism caused by GPU programming:

- Each dynamic type must be enclosed in a GPU `Exp`.
- The number type is restricted to `Float` due to the library implementation.

linear

This type signature can be made much more polymorphic. The body of the function is simple linear arithmetic, utilising the `(+)`, `(*)` and `(/)` constructs. The type class restriction required for these operations is that `a` belongs to `P.Fractional`. So the restrictions that the arguments be `Exp Floats`, or even just `Floats`, is unnecessary.

```
linear :: (P.Fractional a) => (a, a) -> (a, a) -> a -> a
```

GPU programming does not restrict the type polymorphism of this function. It has been generalised and can be utilized by other functions, including non-GPU ones. In terms of type polymorphism, it is an elegant function.

Conclusion

Overall, it was possible to improve the type polymorphism for this mandelbrot program. This was best seen in `next` and `linear` where there were very little type class restrictions, apart from single Prelude type classes intended to enable polymorphism over calculations. Smaller polymorphism improvements were made in other functions, while certain functions, like `complexOfPixel`, could not be abstracted at all.

Overall, the restrictions in type polymorphism throughout the program manifested as the following problems:

- Top level dynamic parameters enforced as `Acc (Scalar a)`.
- The lifting and unlifting of values enforcing parameters as `Exp a`.
- Restriction to Accelerate type classes enforcing the additional restriction `Elt a`.
- Library specific implementations returning types which could be more general, e.g. `Int32`.

The attempted iteration of type polymorphism of the mandelbrot code showed that Accelerate GPU constructs reduce the elegance possible in code in terms of type polymorphism.

4.5.3 Higher-Order Functions

The analysis of the `mandelbrot` code showed that it made very little use of higher-order functions. One notable higher-order function which was used was `while`, whose function signature is as follows:

```
while :: forall e. Elt e
    => (Exp e -> Exp Bool) -> (Exp e -> Exp e) -> Exp e -> Exp e
```

"while" is notable for because:

- It is an Accelerate/GPU function.
- It is a higher-order function, as its first two arguments are functions.

The first argument takes a `Exp e` and returns a `Exp Bool`. The second takes a `Exp e` and returns a `Exp e`. These are the condition and updation of the `while` loop construct.

So although the mandelbrot implementation does not make much use of higher-order functions, the use of `while` shows that they can be used in Accelerate programs. In particular, it shows that higher-order functions dealing with GPU expressions can be used.

While the use of higher-order functions is dependent on the implementation and the amount of abstraction possible/desired, Haskell programmers traditionally make use of higher-order Prelude functions which capture common patterns of computation. Some notable ones include `zipWith`, `map`, `filter`, `scan` and `fold` [25]. When we examine the Accelerate documentation, we can see that the GPU equivalent of these higher-order functions exist in Accelerate [26]. We can compare Accelerate's hackage listing with the Haskell prelude's one to see that accelerate enables equivalent higher-order functions for GPU arrays and expressions [27]:

```
-- Prelude function:
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
-- Accelerate function:
zipWith :: forall sh a b c. (Shape sh, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c) -> Acc (Array sh a) -> Acc (Array sh b)
        -> Acc (Array sh c)

-- Prelude function:
map :: (a -> b) -> [a] -> [b]
-- Accelerate function:
map :: forall sh a b. (Shape sh, Elt a, Elt b)
    => (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)
```

```
-- Prelude function:
filter :: (a -> Bool) -> [a] -> [a]
-- Accelerate function:
filter :: (Shape sh, Elt e)
       => (Exp e -> Exp Bool) -> Acc (Array (sh :. Int) e)
       -> Acc (Vector e, Array sh Int)
```

We can see that the prelude's higher-order functions have Accelerate equivalents, where GPU arrays take the place of generic arrays, and GPU Expressions take the place of ad-hoc type variables. The types are restricted by Accelerate conventions as discussed earlier, but the higher-order functions work with these type restrictions to allow them to be performed efficiently on the GPU. Additionally, standard higher-order Prelude functions can be leveraged in a function dealing with more traditional types.

So higher-order functions can be leveraged in an Accelerate program just as in a traditional Haskell program, on both GPU expressions/arrays and traditional variables/arrays. Accelerate does not get in the way of passing functions as arguments to other functions, and processing them. In fact, it seems to encourage them.

An interesting note is that the mandelbrot implementation is low on higher-order functions. This restriction is implementation specific, as the implementation is built up of functions required specifically for mandelbrot calculations. An observation can be made that this implementation makes less use of higher-order functions than Jones' non-GPU Haskell implementation [16]. The Accelerate implementation is forced to be fundamentally different from Jones' not due to Accelerate's relationship with higher-order functions, but its relationship with laziness. Laziness is at the core of Jones' implementation, on top of which he composes Haskell prelude functions. As I will examine in Section 4.5.4, this is not possible in Accelerate, which restricts the composition of functions in this implementation.

4.5.4 Laziness and Purity

Inspecting the laziness and purity of the Mandelbrot implementation revealed the same result throughout each function, that the code was pure, but not lazy. To assess this final property of elegance with regards to GPU programming, the next step was to try to improve this property in the implementation, to deeper understand the implications of GPU programming on it.

Unlike declaration style and type polymorphism, which had not been maximised, and higher-order functions, which were applicable in general but not to this problem, laziness and purity had already been maximised.

Analysing Accelerate solely under the property of purity, it is clear that Accelerate implementations are pure. In the Accelerate language, "Accelerate operations are pure and thus do not mutate arrays" [24]. At the top level, "run is a pure function at the level of the user-level Accelerate API". When we examine the mandelbrot code, "run" occurs at the top level of the mandelbrot set generation. "run" is the top level function of Accelerate which returns the output of the GPU operation, which can be interleaved into regular Haskell code. Assuming that the regular Haskell code outside of "run" is pure, and knowing that the code inside of "run" is pure, then the purity of the code cannot be improved. So purity is not a property of Haskell that needs to be sacrificed to leverage GPU programming.

While the code is pure, much of it is not lazy. The Haskell code is evaluated lazily until evaluation of `run` is required, which hands over execution to Accelerate. Accelerate processes the AST, compiles it to CUDA code, runs this CUDA code, and returns the result to Haskell as the output from `run`. Any code that needs to leverage the GPU, forms part of this AST and thus is executed strictly by CUDA. CUDA's strict, sequential C-based implementation means that the author cannot leverage laziness in composing modules as Jones did in his functional pearl [16]. Instead of deciding the depth to visualise with `take`, the code uses `while`, a structure more commonly used in imperative languages. So the use of Accelerate has restricted the program's elegance with regards to laziness, restricting the modularity as described by Hughes [4] and making the programmer leverage certain imperative concepts rather than functional ones.

4.5.5 Final Results

Summary of Results

Function	Lazy/Pure	Type Polymorphism	Declaration Style	Higher-Order
mandelbrot	mid	mid	high	mid
complexOfPixel	mid	mid	high	low
dot	mid	mid	high	low
step	mid	mid	high	low
next	mid	high	high	low
escapeToColor	mid	mid	mid	low
ultra	mid	mid	mid	low
linear	mid	high	high	low

Table 4.3: Assessment of Elegance for Final Iteration.

Evaluation of Results

The assessment and iteration of the code showed the following implications of Accelerate GPGPU programming on elegance:

Accelerate GPGPU code is pure, is not lazy, restricts type polymorphism, restricts declaration style programming and allows higher order functions. GPGPU programming therefore limits the elegance of Haskell.

As discussed previously, Accelerate's execution agent takes control of the program at the top level `run` command, processing the AST, compiling the appropriate CUDA kernel, running it and returning the result back to Haskell. The Haskell runtime has no way of enforcing laziness and purity due to the handing over of execution. Despite this, Accelerate's developers make a conscious effort to keep the execution pure, preserving a key elegant property of Haskell. The same is not true for laziness, as the CUDA code is compiled and run strictly before handing the result back to Haskell.

We see restrictions on the type polymorphism of the program, primarily through the forcing of `Exp` types and their enclosed types belonging to `Elt`. Some polymorphism is still possible, one example being ad-hoc polymorphism adhering to `Elt` within `Exp`. In some instances, polymorphism without restriction to these constructs is possible. This is where values do not have to be `lifted` or `unlifted` out of and into GPU expressions. In this case, functions are more generalisable and can be used by both GPU and non-GPU code. In general though, this is not the case, and the programmer may have to write GPU expression-specific functions.

Declaration style could achieved throughout most of the program, with many expression style constructs being updated to declaration style, and longer function bodies being updated to shorter equations. This showed the flexibility of Accelerate in terms of programming style. Declaration style could not be fully achieved though, with restrictions on guarded expressions and pattern matching. This is due to Accelerate's deep embedding. The value of the pattern match would only be known by Haskell once the expression has been evaluated, i.e. once Accelerate has returned its result to Haskell, which would be reliant on the pattern match, resulting in a circular dependency [28]. This same problem occurs with `if` statements within the deep embedding, so Accelerate overwrites the syntax to represent its own conditional operator which is handled at CUDA runtime. Pattern matching is more general than `if` though, as it works on more than just Boolean operators, so cannot be replaced by a single equivalent function. As a result, the expression style `if` statement is available to us while guards and pattern matching are not. GPGPU programming through deep embedding has restricted the elegance of the syntax in this way.

Analysis also showed that higher-order functions may be leveraged in Accelerate GPU

programming, including equivalents to higher-order Prelude functions in Accelerate that receive and return embedded expressions. This specific implementation used very few higher-order functions in comparison to the elegant, sequential implementation analysed, but that was due to the latter's use of laziness being central to its modularity, which could not be leveraged in Accelerate [16]. This is a practical example of Hughes' argument in "Why Functional Programming Matters", on how laziness and higher-order functions together are the glue that enable modularity in functional programming [4]. Without laziness some of this modularity was unavailable, but in general higher-order functions are still available to be leveraged where possible.

Overall, the following elegant properties of Haskell hold for GPGPU programming:

- The code is pure.
- The code uses higher-order functions, where possible.

The following limitations to elegance of Haskell GPGPU programming were observed:

- The code is not lazy.
- Type polymorphism is restricted.
- Declaration style programming is restricted.

5 Evaluation

5.1 Choice of Library

The evaluation of results showed that purity and higher-order functions could be leveraged in Accelerate, while there were restrictions to laziness, type polymorphism and declaration style programming. The positive results hold for GPGPU programming in general; if a property is possible in Accelerate, then it is possible in GPGPU programming. However, if a property is not possible in Accelerate, this may be a library specific limitation that is not present in another library, therefore not a limitation on GPGPU programming in general. It is important to evaluate the properties found not to hold more generally, using knowledge from Section 2.2 and Section 2.3 to understand if these properties could have been preserved with a different choice of library.

Laziness is a property found not to hold for Accelerate programming, due to Accelerate's deep embedding and handing off to the CUDA runtime, which executes strictly. This is not unique to Accelerate. Section 2.3 showed that Obsidian and Nikola operate under the same premise. They are deeply embedded in Haskell, with their syntax building an AST, which gets translated to CUDA and executed before handing the result back to Haskell. This is a sensible requirement; for Haskell to leverage GPUs, it must hand its execution to tools which leverage the GPU. So code that runs on the GPU cannot be lazily evaluated by Haskell, and is evaluated by the underlying GPU technology. GPGPU tools' implementation as an extension of C means that they get executed strictly. The lack of laziness is not due to the choice of the Accelerate library, and is a more general result of GPGPU programming.

Type polymorphism is a property found to be restricted by Accelerate programming. In particular, variables needed to be enclosed as `Acc` arrays or `Exp` scalars. These enclosed types needed to be of the type class `Elt`, i.e. types that were valid as part of GPU expressions in the AST. In Section 2.3 we saw a similar construct in Nikola, where every value in the type signature of a Haskell function needed to be have an `Exp` type constructor to make it a Nikola function. However Nikola sought to be less restrictive with types, enabling general Haskell function compilation onto GPUs. Obsidian was more restrictive than Accelerate, enabling kernel computations on arrays of values of Integers, Floats,

Booleans, arrays or tuples. Nikola's approach shows that less restrictive approaches are possible with regards to type polymorphism, but the lack of support for Nikola for the past ten years means that this can currently be leveraged in theory but not in practice.

Declaration style is a property found not to hold for Accelerate programming. Though most of the functions of the mandelbrot implementation could be iterated to declaration style, those containing `if` statements could not be iterated to ones which pattern matched on guards. This is due to the deep embedding, and is a problem that the authors of Accelerate are currently working on [28]. Pattern matching occurs at Haskell runtime, but Haskell cannot know the outcome of the pattern match until after the patterns have been evaluated, i.e. after the deeply embedded AST has been evaluated and the result returned to Haskell, at which point it is too late to evaluate the pattern. This is a shortcoming of deep embeddings in general, rather than specifically Accelerate or GPGPU programming. Yet, it is one that Accelerate's developers are working on, defining a general method for pattern matching in deeply embedded languages, using Accelerate as a specific EDSL to illustrate this method. So while this particular shortcoming in terms of declaration style is currently unavoidable in deep embeddings, therefore unavoidable in Haskell GPGPU programming, it soon will be avoidable in Accelerate specifically, and other EDSLs which choose to implement this method.

This analysis shows that Accelerate was an appropriate choice of library to tackle the question of elegance in Haskell GPGPU programming. Use of Accelerate showed that purity and higher-order functions were possible. Restrictions in laziness would have shown in any other Haskell GPGPU library due to the requirement of deep embedding and execution by GPGPU tools. Very little restrictions were shown to declaration style programming, and those that were found are actively being worked on by Accelerate's developers to solve a more general problem. So use of and further investigation of Accelerate has shown that declaration style is possible in theory in GPGPU programming, and will soon be possible in future. Accelerate was only found to be overly-restrictive in its type polymorphism, restricting types within the `Exp` construct in a way that Nikola does not. However, Nikola's lack of maintenance and low userbase makes it an unreasonable choice for GPGPU programming in practice, and its type signature still requires `Exp` type constructors to signify a deep embedding.

5.2 Definition of Laziness

Background research in Section 2.1 and Section 2.4.2 informed a working definition of elegance. This definition consisted of four key properties; laziness/purity, type polymorphism, declaration style programming and higher-order functions. Having applied this definition to research, it was now possible to evaluate the definition itself to assess if the

same properties were still relevant.

Laziness and purity were two concepts linked together in the definition, informed by the background research. "A History of Haskell" noted the link between laziness and purity, and the idea of lazy evaluation leading to unreliable side-effects, noting that "laziness kept us pure" [3]. Analysis of the mandelbrot code, Accelerate library and Haskell GPGPU programming in general showed that these concepts should not be linked as the same property. The level of elegance for this property throughout the code was middling, but this was really an average of two properties; the code was pure, but it was not lazy. This does not contradict any concepts found in the background research - laziness implies purity, but purity does not imply laziness. This elegant property should be separated into two properties in the working definition for elegance; *"An elegant program is lazy in 100% of its evaluations"* and *"An elegant program is pure in 100% of its evaluations"*.

In terms of the validity of these individual properties themselves, they remain valid properties of elegance. Purity's importance as found in Section 2.1 and Section 2.4.2 was reinforced by analysis of Accelerate, which found that its developers ensured that its computations remained pure, making it a "purely functional embedded language" [24]. Purity has beneficial implications in Accelerate; immutable arrays mean that device arrays can be shared, reducing the memory required. An array only needs to be transferred to the device on first use, as it cannot be changed ahead of later uses. Concurrent evaluation is also always sound due to purity. Indeed McDonnell argues that "purely functional embedded languages represent a good programming model for making effective use of massively parallel SIMD hardware" [24]. The benefits of purity also show in the ongoing work on embedded pattern matching, where "the construction of the AST for the embedded program has to be a pure function" for it to be successful [28]. Purity's importance to the elegance of Haskell, as analysed under both discussions of the design committee and also its importance to fractals, has been reinforced when looking at its role in leveraging GPGPU programming in Haskell.

While purity was central to GPGPU programming in Haskell, laziness was mostly absent, only appearing in regular Haskell code and being swapped for strict execution when the CUDA runtime took control. This calls into question whether laziness is a necessary elegant property in Haskell if GPGPU programming can be achieved effectively without it. While the functionality could be achieved, analysis of it when acknowledging the background research shows that there is a sacrifice in Haskell's elegance to achieve it. Hughes previously attributed laziness and higher-order functions as two elegant properties that were the "glue" of functional languages, enabling concise modularity [4]. It was also attributed to the modularity of Jones' elegant implementation of generating the Mandelbrot Set in Haskell, where we see a practical example of the separation of a generator and a selector as described by Hughes [16][4]. The Accelerate code does not have laziness at its core, as this is restricted by GPGPU programming. Instead, it achieves the functionality with an imperative

`while` construct. Without laziness at its core, the generator/selector model falls away, as does the combination of laziness and higher-order functions as the glue of the program. This affected the elegance judgement of the program for the property of higher-order functions, even though higher-order functions are present in Accelerate. The absence of laziness in GPGPU programming negatively affected the elegance of the implementation studied, therefore making laziness a valid property of elegance.

As well as laziness, higher-order functions were deemed to be a property of which the implementation was particularly lacking. Although it was low in this particular implementation, it was shown to exist in Accelerate, with the lack of it in this implementation being due to a lack of laziness. As the implementation was achieved effectively without higher-order functions, the same question arises as for laziness: is it a necessary elegant property? The analysis showed that it is, and that a less concise and less modular implementation in Accelerate was due to lack of laziness and higher-order functions in the code. Higher-order functions are as crucial as laziness in making an elegant, concise and modular program as discussed by Hughes [4]. Based on the same logic as laziness being a valid elegant property, so too is use of higher-order functions. The developers of Accelerate are clearly aware of this importance, ensuring to provide implementations of prelude higher-order functions for the programmer to leverage in a GPU context. But, without laziness, a programmer cannot fully utilize their elegance.

A fourth aspect of the definition of elegance was the ability to achieve declaration style programming as defined in History of Haskell [3]. This was a choice to but some formal definition on the property of elegant style and syntax, which is usually an intuitive property attributed to Haskell's elegance. The choice was grounded in a distinction made in styles by members of the original Haskell committee, but was somewhat arbitrary. Seeing this style leveraged in other literature which referenced elegant style and syntax showed the validity of this choice [16]. The properties of declaration style enabled some grounded analysis of the syntax of the mandelbrot example, Accelerate, GPGPU programming and deep embeddings, most notably the distinction between utilizing `if` statements while being unable to utilize pattern matching. The background research coupled with analysis of the implementation showed the importance of having some rigid property of elegant style and syntax as part of the definition. However, the choice of exactly which syntactic elements is still subjective, so this property of the working definition of elegance is the most malleable. A future iteration of elegance may emphasise expression style, or some mix of syntactic components, as elegant syntax, provided that this property has some credibility.

The fifth property of elegance was the use of type polymorphism. This property was informed by discussions of type classes, Haskell's "most distinctive characteristic", in History of Haskell [3]. Type classes were a key contribution of Haskell to functional programming, and were a property that didn't exist a the time of Hughes' Why Functional Programming

Matters. They enable overloading of operations and type polymorphism. Polymorphism was also a key feature of Jones' functional pearl composing fractals [16]. Polymorphism was thus defined as an elegant property. This property was tested for in the mandelbrot implementation by assessing whether additional restrictions were placed on type polymorphism by Accelerate - if so, the elegance was restricted. This was common across the implementation, with type signatures typically being enclosed in the `Acc` or `Exp` type constructors, and enclosed types belonging to the class `Elt`. But this restriction is desired by the authors. As part of current work on embedded pattern matching in Accelerate, `Bool` is being removed from the `Elt` class, due to evaluation of boolean expressions being replaced by pattern matches on conditions [28]. The set of types now "contains only real machine types: signed and unsigned integers ..., floating point numbers ..., and SIMD vectors of these types". It is clear that the authors have purposely restricted the polymorphism of `Elt` with the aim of representing real GPU types. When looking at the restrictions from this end, we can see that the authors are leveraging type classes to ensure that functions can only process appropriate GPU types. The `Exp` type constructor, which represents an embedded expression, is also necessary to distinguish between an embedded type and non-embedded type, allowing types to be lifted into and unlifted from the deep embedding, thus allowing them to be used for different functionality. We see its presence in Nikola as additional evidence of its necessity to the deep embedding required. So the restriction of type polymorphism in elegance, as it stands, should be removed from the working definition. It stands to reason from the research that some property of elegance in Haskell relate to type classes and type polymorphism, but it is difficult to strike the balance between desired restrictions and parametric polymorphism in the general case.

5.3 Research Method

The first step of the research method was to carry out background research. This included background research on elegance in Haskell, Haskell GPGPU programming, and fractals. These areas were researched to form a well founded working definition for elegance, choose and understand an appropriate GPGPU library, and select a suitable problem domain specifically. Following this, elegance was defined and Accelerate was chosen. Research into fractals and Accelerate showed that two different Mandelbrot Set examples had been written by the developers. It was decided to assess the elegance of these, note a difference in elegance, and iterate on the more elegant one. This would demonstrate practically the limitations of elegance in Haskell GPGPU programming.

Upon reflection, one substantial improvement that could have been made to the research method was to answer aspects of the question in the background research stage. Assessment of the elegance in the background research stage would have revealed certain answers earlier,

including Accelerate's relationship with purity and laziness. This would have guided the implementation stage better, giving more direction to assessing each of the elegant properties rather than seeking to fully understand by analysing the implementation.

Additionally, the project would have benefited from writing the implementation from scratch, rather than iterating on existing examples. Guided by background research of the elegance achievable in Accelerate, an implementation could have been written which seeks to maximise elegance from the start. This would be guided by the elegant, non-GPGPU, Haskell-based approach implemented by Jones [16]. This would enable an elegant design from first principles, saving some time in assessing and iterating already existing code. This could be written from scratch utilizing Stack or Nix for package management, mitigating the time lost in package and project setup in Cabal. Additionally, this While the project was completed and the research question was answered on time, saving time using this method could have enabled progress on some of the Future Work in Section 5.4.

While time could have been saved in certain areas of the project, the research method enabled the question to be answered. Research into elegance using the chosen sources led to a sound working definition for elegance, and research into GPGPU libraries in Haskell enabled the one to be chosen which, upon reflection, allowed the most generalisable results. Assessment and iteration of the code did enable the question to get answered, and learnings on Haskell, GPGPU programming, Accelerate and deep embeddings in general. Fractals, in particular, were an ideal problem domain under which to examine elegance and GPGPU programming, due to their suitability to both, and led to results which could be generalised to broader Haskell GPGPU programming.

5.4 Future Work

5.4.1 Testing with Obsidian

Background research before the implementation, and evaluation afterwards, showed that Accelerate was the most suitable library with which to assess the elegance of GPGPU programming. Nikola's lack of support and maintenance ruled it out as a suitable contender for modern GPGPU programming. Obsidian, a lower level EDSL, was ruled out due to its closer ties to imperative GPGPU constructs. While this project showed that elegance is mostly possible in GPGPU programming, it may be beneficial to see if a similar level of elegance is possible with another library, i.e. Obsidian. This would contribute to the literature in demonstrating whether there is flexibility in library choice to achieve elegant Haskell GPGPU programming. In terms of the current research, the future work could make use of the current working definition of elegance, findings about restrictions of GPGPU programming, findings about restrictions of EDSLs and a best case measure of elegance for

GPGPU programming in Haskell.

5.4.2 Pattern Matching in Accelerate

As discussed, there is ongoing work in implementing embedded pattern matching in Accelerate. When this is implemented, future work could re-examine the elegance of Accelerate, to understand if declaration style may be now fully leveraged in Haskell GPGPU programming.

5.4.3 Evaluation of Boilerplate Code

The evaluation and iteration of the Mandelbrot code at each stage of the process involved working with the core code that generated the set, not concerned with how it was visualised. The second, dynamic, example visualised the set with an interactive GUI, which required boilerplate code to achieve this with Accelerate. While evaluation of the core Mandelbrot code answered questions about GPGPU programming, evaluation of the boilerplate code could make a good case study on evaluation of the elegance of graphical programming in Accelerate, which is a more specific domain of GPU programming. This could lead to equivalent discoveries on the capabilities and restrictions on Haskell GPU programming in the graphics domain as were discovered in the general purpose domain.

6 Conclusion

It is clear from the research that Haskell GPGPU programming is not fully elegant. Limitations exist in laziness, type polymorphism and syntax. However, limitations in type polymorphism exist by design, and as a result of deeply embedded expressions, a property much more general than to just GPGPU programming. Similarly, the limitations in syntax exist as a general result of deep embedding, a limitation that is being actively overcome in Accelerate with general implications on deeply embedded languages.

The only unique restriction that GPGPU programming places on Haskell is on laziness. This is not enough of a restriction to discount Haskell as a solution to the difficulties of GPGPU programming. An elegant program is the ideal case, and in the real world many programs will not be fully elegant. That a GPGPU program can be mostly elegant shows that it leverages most of the properties that make Haskell worthwhile. Haskell GPGPU programming leverages the performance efficiencies of GPUs and the core properties of Haskell. While there are sacrifices in both, they are worthwhile for a programmer that wishes to leverage both at once.

Accelerate is a powerful library and EDSL that enables GPGPU programming to a high standard, leveraging Haskell to high potential. It remains supported over a decade after its release, and the authors continue to improve its features, including those which directly affect elegance such as embedded pattern matching. Overall, it is a useful and ever-improving tool for a functional programmer who seeks to gain performance increases in their code, or a GPGPU programmer that seeks to leverage the benefits of functional programming.

The definition of elegance in this dissertation remains a working definition. Purity, laziness and higher-order functions play a key role, as does syntax - albeit with a more flexible and subjective interpretation. Background research also indicates that type classes and polymorphism have a part to play in the elegance of a program, but it is difficult to bound a restriction on what is “too restrictive” or “too polymorphic”. In any case, the working definition of elegance provides ample groundwork on assessing the usefulness of domain specific programming in Haskell, understanding if they leverage its key properties enough to justify its use over other high level languages.

Bibliography

[1] Paul Hudak and Joseph H Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5):1–52, 1992.

[2] Simon Marlow et al. Haskell 2010 language report. 2010.

[3] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.

[4] John Hughes. Why functional programming matters. *The computer journal*, 32(2): 98–107, 1989.

[5] Cuda toolkit, May 2021. URL `https://developer.nvidia.com/cuda-toolkit`. Accessed: 2021-11-31.

[6] Cuda c programming guide, Nov 2021. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 2021-11-31.

[7] Joel Svensson. *Obsidian: GPU Kernel Programming in Haskell*. PhD thesis, Citeseer, 2011.

[8] Obsidian, 2016. URL `https://hackage.haskell.org/package/Obsidian`. Accessed: 2021-12-10.

[9] Joel Svensson, Koen Claessen, and Mary Sheeran. Gpgpu kernel implementation and refinement using obsidian. *Procedia Computer Science*, 1(1):2065–2074, 2010.

[10] Accelerate, 2020. URL `https://hackage.haskell.org/package/accelerate`. Accessed: 2021-12-20.

[11] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14, 2011.

[12] Cuda package, 2021. URL https://hackage.haskell.org/package/cuda. Accessed: 2021-12-20.

[13] Mainland/nikola, 2013. URL https://github.com/mainland/nikola. Accessed: 2021-12-15.

[14] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 67–78, 2010.

[15] Benoit B Mandelbrot. Fractal geometry: what is it, and what does it do? *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 423(1864): 3–16, 1989.

[16] Mark P Jones. Functional pearl composing fractals. *Journal of Functional Programming*, 14(6):715–725, 2004.

[17] Will Mayfield, Justin Eiland, Taylor Hutyra, Matt Paulsen, Bryant Wyatt, et al. Fractal art generation using gpus. *arXiv preprint arXiv:1611.03079*, 2016.

[18] Acceleratehs/accelerate, 2022. URL https://github.com/AccelerateHS/accelerate. Accessed: 2021-04-06.

[19] Accelerate website, 2017. URL https://www.acceleratehs.org/. Accessed: 2021-12-20.

[20] Accelerate on r/haskell, 2021. URL https://www.reddit.com/r/haskell/search/?q=accelerate. Accessed: 2021-12-29.

[21] Nikola on r/haskell, 2021. URL https://www.reddit.com/r/haskell/search/?q=nikola. Accessed: 2021-12-29.

[22] Acceleratehs/accelerate-examples, 2022. URL https://github.com/AccelerateHS/accelerate-examples. Accessed: 2022-04-06.

[23] Accelerate mandelbrot tutorial, 2016. URL https://www.acceleratehs.org/examples/mandelbrot.html. Accessed: 2021-12-20.

[24] Trevor McDonell. *Optimising Purely Functional GPU Programs*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2015.

[25] Higher order functions - learn you a haskell, 2011. URL http://learnyouahaskell.com/higher-order-functions. Accessed: 2022-04-05.

[26] Data.array.accelerate - hackage, 2022. URL https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html. Accessed: 2022-04-05.

[27] Prelude - hackage, 2022. URL https://hackage.haskell.org/package/base-4.16.1.0/docs/Prelude.html. Accessed: 2022-04-05.

[28] Trevor L McDonell, Joshua D Meredith, and Gabriele Keller. Embedded pattern matching. *arXiv preprint arXiv:2108.13114*, 2021.

A1 Appendix

A1.1 dependencies.txt

List of cabal installations required:

gloss
gloss-accelerate
accelerate-debug
criterion
fclabels
accelerate
test-framework
network-http
network-conduit
HTTP
http-types
test-framework-providers
test-framework-hunit
test-framework-quickcheck2
statistics-0.15.2.0
accelerate-llvm
accelerate-llvm-native
accelerate-io
accelerate-io-codec
accelerate-io-bmp
colour-accelerate