Ictora - A Purely Functional Shader Language

Ellis Kesterton

12th March 2020

Contents

Chapter 1

Introduction

1.1 Background

In recent years, GPUs have been a subject of interest from a wide variety of fields. Their highly parallel architecture consisting of many cores complements the typical CPU which has fewer, but faster cores. The contrasting nature of these two processing units makes them an ideal duo for general purpose programming - highly parallel tasks can be offloaded to the GPU, while the CPU can take care of the more sequential jobs. This style of programming has been facilitated by tools such as NVIDIA's CUDA, which allows code to be written in a wide variety of standard programming languages, and through the addition of some minor annotations, allows the code to be ran on the GPU[17]. Yet, while general purpose programming on the GPU has seen great improvement recently, there has been little development in the area that GPUs were originally designed for: *graphics programming*.

The sequence of algorithms used to render a scene is known as the graphics pipeline. Originally, GPUs were simply a hardware implementation of this pipeline - compared to running the pipeline on the CPU, this yielded much greater performance, allowing massively more complex scenes to be rendered. However, an unfortunate side effect of moving the pipeline to hardware is that the algorithms are set in stone, leaving no room to customise the rendering process. To combat this, GPU manufacturers have made certain steps in the pipeline programmable: the programmer can upload a small piece of code to the GPU to be integrated into the pipeline. These small programs are known as shaders.

Unlike CPUs, there is no standardised processor architecture or instruction set for GPUs[30]. For our shaders, that would mean writing a slightly different program for every single GPU: it goes without saying that practically, this is unworkable. To enable programmers to write a single program compatible with many GPUs, GPU manufacturers embed a compiler in the graphics driver, which converts some high level shader language to the GPUs native machine code. Up until recently, the only two widely supported high level languages were GLSL[28] and HLSL[32], tied to their

respective graphics APIs, OpenGL and DirectX.

There are several issues with this approach. Perhaps the most notable is its resilience to change. To change the specification of GLSL or HLSL would mean updating hundreds of graphics drivers, a lengthy process that hardly supports active improvement of the shader languages. Development of new shader languages is also infeasible, as this would require the introduction of yet another compiler into the graphics drivers, something that GPU manufacturers are unlikely to be in support of. In addition, the compilation of these languages must occur at runtime, since the programmer does not know what GPU they are targeting. Shader languages must maintain a balance between being easy to parse and compile (to minimise the runtime compilation overhead), and being user-friendly and practical for the programmer.

To combat these issues, in 2015 Khronos Group (the organisation behind OpenGL and GLSL) proposed the introduction of a new shader language called SPIR-V. Unlike GLSL and HLSL however, this shader language is not intended to be programmed directly, rather targeted as an intermediate form (akin to LLVM) for new shader languages. It supports a binary representation which circumvents the overhead of parsing, and SPIR-V being much lower level makes it significantly quicker to compile. Khronos Group's influence has ensured the adoption of SPIR-V by making it the only supported shader language of their new graphics API, Vulkan. New shader languages can now take advantage of this by using SPIR-V as a compilation target. The compilation to SPIR-V takes place before the shader is run - this opens the door to more complex type systems and more intensive optimisations, which would've been too time consuming to perform at runtime. This dissertation concerns the development of a new shader language, Ictora, targeting the SPIR-V intermediate representation.

1.2 Project Description

The development of SPIR-V has opened up a lot of options for shader language design. Yet, despite SPIR-V opening so many doors, there has been almost no work in this area. This project entails the implementation and design of a compiler and interpreter for a new shader language, Ictora. Designed to tackle the downfalls of existing shader languages, Ictora will be purely functional with a strong type system - something that was unviable before the development of SPIR-V.

Ictora will be implemented in the dependently typed programming language and proof assistant, Idris. A complex project like a compiler is typically prone to many bugs, but by taking advantage of Idris's precise type system many of those bugs can be prevented at compile time. Following techniques pioneered by McBride[29], after type checking only well-typed terms will be represented, allowing the compiler to be represented by a total function - this project is also a demonstration that this technique scales into real world languages, and not just contrived examples.

1.3 Related Work

A considerable amount of effort has gone into writing compilers from GLSL and HLSL to SPIR-V[1], and cross compilers between HLSL and GLSL[21]. These projects allow existing GLSL and HLSL shaders to take advantage of some of the benefits of SPIR-V, such as a lower runtime overhead, compatibility with the Vulkan API, and compile-time optimisations. However, GLSL and HLSL compilers are still present in graphics drivers, and hence development is still stifled by the graphics driver compilation system.

Futhark[2] is another related project. It is a purely functional programming language designed to run on the GPU, but, unlike Ictora which is for writing shaders, Futhark is used to write general purpose GPU programs. Despite their differences, both Futhark and Ictora both share the common goal of aiming to make GPU programming more accessible, and closer to the comfort of traditional CPU programming.

[1] Available here: https://github.com/KhronosGroup/SPIRV-Cross
[2] Homepage: https://futhark-lang.org/

Chapter 2

Prerequisites

2.1 Lambda Calculi

2.1.1 An Overview of the Untyped Lambda Calculus

The untyped lambda calculus is perhaps the smallest programming language imaginable. It is an expression based language, where evaluation/execution involves continuously rewriting the expression into a simpler form. This is much like basic arithmetic, where one can rewrite the expression $(1 + 1) + 1$ as $2 + 1$, and then as 3 - we say the result of evaluating $(1 + 1) + 1$ is 3. A lambda calculus expression consists of only three simple constructs. Using the meta-variables M and N to represent arbitrary expressions, and x to represent an arbitrary name, we detail the constructs below:

Construct	**Syntax**	**Description**
Variable	x	A name representing another lambda calculus expression, much like variables in algebra.
Abstraction	$\lambda x\,.\,M$	A function which takes an argument of name x and has a body of the expression M. Any occurrences of the variable x in the body M refer to the argument of the function.
Application	$M\ N$	A call to function M with argument N.

Evaluation of a lambda calculus expression is fundamentally the repeated process of applying functions (abstractions), by substituting the argument into the body. For example, consider the following expression:

$$(\lambda x\,.\,x)\ y$$

We have the function $\lambda x\,.\,x$ applied to the variable y. To apply the function, we substitute y into the body of the abstraction for every occurrence of the argument name, x. We denote this substitution by the following notation:

$$x[x \mapsto y]$$

Reading the notation from left to right, we can read this as "in the expression x, substitute every occurrence of x for y". The result of this substitution is trivially the variable y.

The previous example only involved a single function application, but most interesting programs will involve a lot more. Below is the step by step evaluation of a slightly larger example:

$$(\lambda x\,.\,x)\ (\lambda y\,.\,y)\ z$$
$$(x[x \mapsto \lambda y\,.\,y])\ z$$
$$(\lambda y\,.\,y)\ z$$
$$y[y \mapsto z]$$
$$z$$

2.1.2 Variables and Names

Free and Bound Variables

Those familiar with first order logic may already be familiar with the notion of *free and bound variables* - the meaning is exactly the same in the context of the lambda calculus. A bound variable is one which is referring to the argument of one of the abstractions surrounding it. Every variable which is not a bound variable is a free variable: this is the set of variables who are not surrounded by an abstraction introducing an argument of the same name. For an example, see the following expression.

$$\lambda x\,.\,x\ y$$

The variable x is bound, since it is referring to the argument x of the surrounding lambda abstraction. On the other hand, the y variable is free, as there are no lambda abstractions surrounding it that introduce a parameter of y.

Bound variables always refer to the innermost lambda abstraction with that name. The following expression contains multiple lambda abstractions introducing the name x.

$$\lambda x\,.\,\lambda x\,.\,x$$

We say that the innermost parameter x *shadows* the previous one. The careful reader might observe that the name we give to bound variables does not change the meaning of the lambda expression. The following two expressions are identical, apart from the name we gave to the argument and variable.

$$\lambda x\,.\,x$$
$$\lambda y\,.\,y$$

The x and y are both bound to the same lambda abstraction, despite having different names. Semantically, there is no difference between these two expressions - they are both the identity function. When one expression can be converted to another by renaming only bound variables, the two expressions are said to be *alpha equivalent*. Renaming bound variables in an expression is called *alpha conversion*.

Revisiting Substitution

Upon understanding free and bound variables, a few substitution nuances come to light. First, consider how one might evaluate the following expression.

$$(\lambda x\,.\,\lambda x\,.\,x)\ y$$

With shadowing in mind, we know that the variable x is referring to the innermost lambda. When applying the outermost lambda to y, substituting y in for the rightmost variable x would not make sense, as the x it was referring to was not the outer lambda. More generally, substitution must only replace free occurrences of the target variable. In this example, the rightmost x is not free even after stripping the outer lambda off, and so substituting y should have no effect.

Another nuance of substitution is variable capturing. In the following expression, we substitute free x variables for free y variables:

$$(\lambda y\,.\,x)[x \mapsto y]$$

$$\lambda y\,.\,y$$

The y variable, originally free, now refers to the lambda expression. This is known as capturing, and is something we want to avoid as it changes the meaning of the expression. The workaround is simple: rename bound variables through alpha-conversion before performing the substitution. In our example, we rename the λy to λz before substituting, so that the variable y remains free:

$$(\lambda y\,.\,x)[x \mapsto y]$$

$$(\lambda z\,.\,x)[x \mapsto y]$$

$$\lambda z\,.\,y$$

2.1.3 Formalising Evaluation

Currently, our notion of evaluation is simply "apply functions until there are none left". While this is indeed the core idea, it is not a very formal or technical explanation. Formalising our notion of evaluation will help us refer to the concepts surrounding it later.

First, the process of applying a function in the lambda calculus is called β-reduction. Expressions of the form $(\lambda x\,.\,e_1)\ e_2$ can be β-reduced to $e_1[x \mapsto e_2]$ (using capture avoiding substitution). Expressions which we can directly β-reduce are referred to as β-redexes, short for β-reducible expression.

The lambda calculus can also be reduced in another way which we haven't discussed yet. Consider the following two expressions:

$$\lambda x\,.\,y\ x$$

$$y$$

Applied to arguments, both these expressions produce exactly the same result - yet, neither is β-reducible to the other. The two expressions are said to be *extensionally* equal. For example, suppose we were to apply both the expressions to a new variable z. The first expression would become $(\lambda x\,.\,y\ x)\ z$, which β-reduces to $y\ z$. The second expression becomes $y\ z$ - exactly the same! This equivalence is realised in the form of another reduction rule, η-reduction. Formally, an expression of the form $\lambda x\,.\,e_1\ x$ can be η-reduced to e_1, on the condition that x is not free in e_1. The inverse of this transformation is called η-expansion.

Finally, we introduce the concept of a normal form. When an expression no longer contains any β-redexes, we say that it is in β normal form[1]. Intuitively, an expression in β normal form is fully evaluated.

2.1.4 Adding Simple Types

An Infinite Loop

One problem with the lambda calculus as it stands is that the evaluation of certain expressions can leave us in an infinite loop. In general, this is an undesirable property - a lambda calculus program which never finishes evaluating never really produces a result. The presence of an infinite evaluation loop is equivalent to saying that the expression has no normal form. However much we β-reduce, we can never get rid of all of the redexes. For example, consider the following expression:

$$(\lambda x\,.\,x\ x)\ (\lambda x\,.\,x\ x)$$

If we β-reduce the only redex available, we are left with the same expression we started with! No matter what we do, we cannot reach a normal form. The expression has a β-redex, yet if we β-reduce it only creates another one. Below is another example of an expression without a normal form:

$$(\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)$$

Attempting to evaluate this expression results in an infinitely growing term!

$$(\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)$$

[1]Sometimes it is useful to define normal forms slightly differently, but for simplicity *normal form* will always mean complete reduction in this document.

$$(\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)$$
$$(\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)\ (\lambda x\,.\,x\ x\ x)$$
$$...$$

Introducing STLC

To eradicate these troublesome programs that do not have a normal form, Alonzo Church decided to categorise terms into different types. Unlike the untyped lambda calculus, where any syntactically valid program is allowed, Church's system additionally imposed some extra constraints. The constraints allow or disallow the construction of expressions based on their typing. This whole system is called the simply typed lambda calculus, or STLC for short.

Types in the STLC are defined by the following grammar:

$$\tau ::= \circ | \tau \to \tau$$

$\circ$ is the base type - it represents any atomic type, so it may help to think of it as an integer or boolean for example. $\tau \to \tau$ is the type of functions. The type of the left hand side of the arrow is the input to the function, and the type on the right hand side of the arrow is the result. The STLC syntax is identical to the untyped lambda calculus with the addition of a typing annotation on lambdas. Instead of $\lambda x\,.\,e$, we write $\lambda x : \tau\,.\,e$. This extra annotation defines the type of the argument that the lambda takes as its parameter. In general, whenever we see something of the form $e : \tau$, it means that e has type τ.

Before we can begin defining those constraining rules, we need to introduce the notion of a typing context. Commonly denoted Γ, a typing context maps free variables to their types. One such example would be $\Gamma = \{x : \circ, y : \circ \to \circ\}$. Extending the context with another mapping is denoted as $\Gamma, x : \tau$.

Finally, we move onto the rules themselves. Formally, the rules are called typing judgements, and the combination of all the rules is called a type system. The STLC only contains three rules, and so we will step through them one by one. First, the rule for variables:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{T-Var}$$

Typing judgements are structured like any other logical rule - the premises are written above the horizontal line, and the conclusion below. Our rule *T-Var* can be translated into the following english sentence: *If a mapping from x to τ exists in the context Γ, then from the context Γ we can determine that the variable x has type τ*. While this just seems to be stating the obvious, it is necessary for a proper formulation. Next, we consider the rule which applies to the application of expressions.

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f\ x : \tau_2}\ \textit{T-App}$$

This rule is fairly intuitive. Applying a function of type $\tau_1 \to \tau_2$ to an argument of type τ_1 will give us a result of type τ_2. Finally, the rule for lambdas:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \,.\, e) : \tau_1 \to \tau_2}\ \textit{T-Var}$$

Lambda expressions bring their arguments into scope in the body, hence the extension of the context in the premise of *T-Lam*.

If we can prove that a term has a type using these rules, we say that the term is well typed. The combination of rules used to prove this fact is called a typing derivation. All well typed terms in the STLC have a normal form, regardless of what order the β reductions are performed - the STLC is said to be a *strongly normalising* language[41].

2.2 Dependently Typed Programming

2.2.1 Dependently Typed Lambda Calculus

In the vast majority of type systems, there is a clear distinction between terms which are values, and terms which are types[23]. A dependent type system makes no such division[27]. Much like functional languages make functions first class constructs, in a dependently typed language types are also first class - this means that types can be both arguments to functions, and the result of a function. Although the advantage of a system like this does not become clearly apparent until we consider user-defined data types, let us first briefly consider the simply-typed lambda calculus extended with dependent types, known as λ_Π.

As terms on the value level and type level are now considered equivalent, we only need a single syntactic category of constructs. The terms in λ_Π are detailed below:

$$\begin{aligned} e, \tau \ ::= \ & x \\ \mid \ & \lambda x : \tau \,.\, e \\ \mid \ & e_1\ e_2 \\ \mid \ & (x : \tau_1) \to \tau_2 \\ \mid \ & \star \end{aligned}$$

The first three constructs should be very familiar - variables, abstractions and applications are the same as in the STLC. Of more interest are the two last constructs,

$(x : \tau_1) \to \tau_2$ and $\star$.

First, lets consider the former, formally known as a Π type[2]. Π types are strangely reminiscent of the familiar arrow type we saw in the simply typed lambda calculus, where the only syntactic difference is the binding of the left hand side to a name (x in this case). The name refers to the value of parameter given, and much akin to how abstractions bind variables, the right hand side of the Π type may contain the bound variable. Practically, this means that return type of a function may change depending on the argument given - hence the name *dependent type*.

The other construct we are unacquainted with is the type of all types, $\star$ (alternatively known as a *universe*). Although we have no syntactic distinction between types and values, inhabitants of $\star$ are what one would refer to as a type. $\star$ is the type of itself, as well as the Π type we just introduced (and, if we were to extend our language with any more primitive types, $\star$ would be the type of those too). Although for our purposes this simple formulation is sufficient, it is worth noting that having $\star : \star$ makes this calculus susceptible to Girard's paradox[12]. In short, this means that the type system of this calculus is not logically consistent, a property that will soon prove quite useful. This issue is typically circumvented by *cumulative universe hierarchies* and *universe polymorphism*[19].

2.2.2 Curry-Howard Isomorphism

The Curry-Howard isomorphism is a remarkable relation between traditional logics and type systems: it shows an equivalence between types and propositions, and programs and proofs[38]. One of the simplest examples of this correspondence is between propositional logic and the simply typed lambda calculus (extended with pairs and sums). Compare the modus ponens rule from logic with the typing judgement for application from the STLC:

$$\frac{A \Rightarrow B \qquad A}{B} \Rightarrow \mathit{elim} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B}\ \mathit{T\text{-}App}$$

The similarity is obvious: implication in the logic rules corresponds with the arrow type from the STLC. This pattern continues when we compare other rules too, such as the $\Rightarrow$ -introduction and *T-Lam* show below.

$$\frac{\begin{matrix}[A] \\ B\end{matrix}}{A \Rightarrow B} \Rightarrow \mathit{intro} \qquad \frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash (\lambda x : A\,.\,y) : A \to B}\ \mathit{T\text{-}Lam}$$

Erasing the terms on any valid typing derivation in the STLC is exactly a natural deduction proof in propositional logic! In practice, this means that we can use the

[2]Other literature may refer to Π types as the *dependent function type*, however this is a bit of a mouthful.

STLC as a method of proving things - all we have to do is find a well-typed term whose type is the thing we want to prove. Suppose we would like to prove the following proposition:

$$A \Rightarrow (A \Rightarrow B) \Rightarrow B$$

We need to find a lambda calculus program whose type is as the same as the proposition (swapping out implications for arrow types). One such program can be seen below:

$$\lambda x : A \,.\, \lambda f : A \to B \,.\, f\ x$$

While the things we can prove in the STLC are quite limited, the relation actually extends to more logic and calculi: in fact almost all forms of logic have a an equivalent type system for the lambda calculus! Proof assistants take advantage of this correspondence by employing a rich type system, capable of proving many more things than are possible in the STLC. The type systems must be designed with care however, as the typing rules must maintain logical consistency for the proofs to be valid. Some features, such as the fix-point combinator, must be avoided because of this reason. Considering the typing judgement for fix, the reason becomes obvious:

$$\frac{\Gamma \vdash f : A \to A}{\Gamma \vdash fix\ f : A}$$

Logically, this says given a proof of $A \Rightarrow A$, we can deduce A. The former is a tautology, meaning we can prove anything - not a very useful logical system.

2.2.3 Idris

What Is Idris?

The language of choice for implementing Ictora is Idris[3]. It is a general purpose, purely functional language featuring dependent types, with Haskell-like syntax. Like many other dependently typed languages (such as Agda[7], Coq[5] and Epigram[29]), Idris doubles as a proof assistant under the Curry-Howard isomorphism. This allows the programmer to prove properties about their programs, the proofs verified by the Idris compiler[8]. Idris's interactive editor assists with the writing of these programs and proofs, using the powerful type system to guide implementation.

This section serves as a brief introduction to the Idris language, and realises much of the theory already discussed surrounding dependently typed programming. For more information on coding with Idris, Brady's *Type-Driven Development with Idris* [9] is an excellent resource.

[3]Homepage - https://www.idris-lang.org/

Encoding polymorphism

One of the beauties of dependent type systems is how simply they can encode many other programming language features: one such feature is polymorphism. Take the polymorphic identity function for example. In Haskell, one might write the following:

```
id :: forall a. a → a
id x = x
```

The type signature of this function abstracts over the type variable a, allowing the caller to (implicitly) instantiate a to type of their choosing. Polymorphism of this form is vital to writing reusable code, as now the same function can be reused in many different contexts. Fortunately for us, Haskell's type abstraction is in fact just a specialised case of the Π types present in dependently typed languages. In Idris, we can write the same function like so:

```
id : (a : Type) → a → a
id _ x = x
```

Instead of using a separate construct to abstract over types, the type can simply be made another parameter to the function. A call to this function would look like the following:

```
id Int 5
```

Of course, explicitly passing the type to every call of this function is tedious. Fortunately, since the value of a can be inferred from the latter parameter, we can make the parameter implicit by enclosing it in curly braces.

```
id : {a : Type} → a → a
id x = x
```

In fact, Idris can infer most implicit arguments - we can omit the type parameter all together if we wish:

```
id : a → a
id x = x
```

We can now call this function without passing the type explicitly:

```
id 5
```

Much more concise!

Abstracting More Than Types

While abstracting over types to simulate traditional polymorphic functions is useful, Idris's dependent types are a capable of a lot more. Consider the rather odd example below:

```
charOrInt : (x : Bool) → if x then Char else Int
charOrInt True = 'a'
charOrInt False = 42
```

The return type of this function depends on the value of the the boolean given: if the boolean is true, the function returns a character, otherwise it returns an integer. Like λ_{Π}, Idris makes no distinction between the type and value level, allowing us to compute the return type of the function using the standard *if* expression. Perhaps to make the example clearer, we could write an equivalent program which breaks out the type level computation into a separate function definition:

```
computeReturnType : Bool → Type
computeReturnType True = Char
computeReturnType False = Int

charOrInt : (x : Bool) → computeReturnType x
charOrInt True = 'a'
charOrInt False = 42
```

The Vectors

Simple algebraic data types[35] can be defined in Idris using the standard Haskell notation. Below are definitions for the natural numbers, optional values and lists (all of which are present in the standard library):

```
-- Defines the Peano naturals.
data Nat
    = Z     -- Zero is a natural
    | S Nat     -- Successor of a natural is also a natural


-- Defines an optional type, where the value may or may not
-- be present. It is parameterised by the type 'a' which
-- it (potentially) holds.
```

```
data Maybe a
    = Just a    -- Maybe either holds a value of that type
    | Nothing   -- Or holds nothing at all


-- Lists are defined in a standard linked fashion, and are
-- also parameterised by the type they hold.
data List a
    = Nil                 -- The empty list
    | (::) a (List a)     -- The operator (::), called "cons",
                          -- prepends an element to an existing
              -- list.
```

In addition, Idris also supports data types written in the more versatile GADT format[33]. The GADT notation enables us to take advantage of dependent types when defining the constructors, and also allows data types to be parameterised by any value, not just types. To define a GADT, one must write down the types signatures of the data type and its constructors manually. For example, first let's redefine the previous 3 data types in GADT notation:

```
data Nat : Type where
    Z : Nat
    S : Nat → Nat


data Maybe : Type → Type where
    Just : a → Maybe a
    Nothing : Maybe a


data List : Type → Type where
    Nil : List a
    (::) : a → List a → List a
```

It is fairly easy to see that any data type expressed with the former notation can also be expressed as a GADT. While one might think this makes the former notation redundant, it is still useful as a concise notation for defining simple data types.

The classic example of a data type which makes use of GADTs is length encoded lists. The idea is to define a data type very alike the existing `List`, which in addition to being parameterised by the type of the elements of the list, is also parameterised by the length of the list. Adding this extra information to the type level allows us to easily write more precise type signatures for our functions. First, let's focus on defining data type, which we will call `Vect`.

```
data Vect : (len : Nat) → (elem : Type) → Type where
```

The type has two parameters, a natural number representing the number of elements in the vector, and a type representing the type of the individual elements. Next we define the data constructors - these can be seen as the logical rules which through which one may construct the type.

```
Nil : Vect Z a
```

This constructor makes an empty vector, like it did for the `List` type. The length of an empty vector is zero, so the resulting type of this constructor is vectors parameterised by `Z`.

```
(::) : a → Vect len a → Vect (S len) a
```

The next constructor is cons. The signature is once again very similar to that of the `List` type, except we additionally encode that applying this constructor increases the length of the vector by one.

Functions defined over our length encoded vectors must mandatorily reason about the length of those vectors involved. First, consider a function which concatenates two vectors together:

```
append : Vect n a → Vect m a → Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

The type signature specifies that the length of the resulting vector must be the sum of the lengths of the two input vectors - a fact which is automatically verified by the compiler. A boon of writing more precise type signatures is that it prevents many implementation errors - the more precise the type, the more sure the programmer can be in the correct implementation. For example, without encoding the lengths of the lists on the type level, one could write a definition of *append* (which passes the type checker) which always returns the first list passed. However, our append defined over length encoded vectors would catch this error at compile time, complaining that the length of the resulting list was n and not $n + m$.

Next, the classic `head` function, which returns the first element of a list. On standard lists the definition may look something like the following:

```
head : List a → Maybe a
head [] = Nothing
head (x :: xs) = Just x
```

Since we do not know whether the list is empty or not, we must return an optional value - in the case of the empty list, there is no first element. But suppose we know the list to be non empty? Perhaps we just created it, or it was the result of a function which always returns a certain amount of elements? It does not make sense to account for the fact that the list might be empty, since we, the programmer, know it isn't! Dependently typed programming is about bridging the gap between what the programmer knows, and what the compiler knows. We need to inform the compiler that if we know a list to be non-empty, then the head function need not return an optional value. Using our definition of vectors, we may define this alternative head function:

```
head : Vect (S n) a → a
head (x :: xs) = x
```

Rather than take in any list at all, we require that the length of the input list must be the successor of some n. Every number is the successor of another except zero, so our function only takes precisely those lists with a non-zero length. Naturally, a list with a non-zero length has at least one element, and so we do not need to match on the constructor for creating empty lists.

Theorem Proving

Dependently typed languages make excellent use of the Curry-Howard isomorphism. Since proofs can be represented as types, and dependently typed systems make no distinction between types and values, proofs can be passed around and constructed like any other object. The result is a powerful theorem prover. This is where Idris shines, as all of this is possible using principles we are already familiar with.

To prove something, we first need to define the proposition that we are trying to prove - a good starting point is the existence of an element in a list. Since propositions are represented by types, we begin by defining a data type. The proposition relates elements and lists, and so we parameterise the data type by their types.

```
data Elem : a → List a → Type where
```

Constructing a value of `Elem 2 [5, 2, 8]` would be a proof that `2` is contained in the list `[5, 2, 8]`. Next, we must define the logical rules which constitute when this proposition is true. If the element is the first thing in the list, then it is indeed contained in the list:

```
    Here : Elem x (x :: xs)
```

We could continue to define constructors for the 2nd, 3rd, 4th... elements, but that would require an infinite amount of code. Instead, we make use of inductive principles

- if we know an element is in a list, we can extend that list by a new element and the original element is still contained in the list.

```
    There : Elem x xs → Elem x (_ :: xs)
```

Now we have a method of proving that a list contains an element, let's put it into action. We will prove that the number `42` is in the list `[99, 36, 42, 2]`. We begin by writing this down as a type signature (remember, proofs are just types!).

```
ourproof : Elem 42 [99, 36, 42, 2]
ourproof = ?theproof
```

The `?theproof` on the right hand side of the function definition is known as a *hole*. It represents an incomplete definition, and is useful for incrementally writing a function without having to worry about the type checker complaining. Using an interactive editor plugin for Idris, we can ask Idris to do several things with a hole. First, we can ask it for the type of the expression that should be used to replace it - if we asked Idris the type of `?theproof`, it would simply reply `Elem 42 [99, 36, 42, 2]`. The other thing we can ask Idris to do is to try and find any value which would type check in its spot, known as a proof search. For example, if we did a proof search on a hole with type `Nat`, Idris would probably come back with `Z` (0). While the proof search isn't particularly useful when dealing with mundane types such as integers, when we're trying to prove things, we don't usually care how it is done - any type-checking implementation is a valid proof. In the case of our little proof, doing a proof search would actually fill in a valid proof automatically! We step through manually for demonstration purposes anyway.

We only have two potential options as the next step - apply one of the `Elem` constructors. A quick glance tells us that `Here` would not make sense, as `42` is not the first element of `[99, 36, 42, 2]`. Hence, we apply the `There` constructor instead.

```
ourproof : Elem 42 [99, 36, 42, 2]
ourproof = There ?theproof
```

We then ask for the type of the hole again. Idris tells us that we are looking for a value of type `Elem 42 [36, 42, 2]`. Once again, we analyse potential constructors that could help us make the value we are looking for. In this case, `There` is the only valid option again:

```
ourproof : Elem 42 [99, 36, 42, 2]
ourproof = There (There ?theproof)
```

Repeating the same process, the `Here` constructor is finally suitable, and so the proof

is complete.

```
ourproof : Elem 42 [99, 36, 42, 2]
outproof = There (There Here)
```

Totality

Since we may put arbitrary computations on the type level (including calls to recursive functions), one may wonder whether the type-checking for Idris is decidable. Type checking for dependently typed languages is intertwined with evaluation, and indeed, if we were to put a computation that infinitely looped in a type signature, the type checker may not terminate. This prompts the introduction of the important concept of totality in Idris. While it is well known that checking whether an arbitrary function terminates is an undecidable problem, we can still devise algorithms which correctly identify a strict subset of terminating functions (known as totality checkers). While there will exist functions which do terminate but are not recognised, in practice this is rarely a problem. In Idris, the totality checker only allows structurally recursive functions, which are those whose recursive calls have parameters which are "smaller" than the current inputs.

Totality is also important for logical consistency too. Allowing non-terminating functions allows anything to be proven - a simple proof of falsehood using a non-total function is shown below[4].

```
whoops : Void
whoops = whoops
```

[4]In Idris, the type `Void` is equivalent to $\bot$ in logic.

Chapter 3

Design

3.1 Existing Language Criticisms

A large part of the motivation for this project was personal dissatisfaction with existing shader languages. The design of Ictora is heavily influenced by these criticisms, aiming to avoid the same problems while retaining their strengths. Hence, having a thorough understanding of these other languages is vital. While other more niche shader languages do exist, we will primary discuss GLSL[32] and HLSL[28] due to their widespread use.

The compilation of GLSL and HLSL at runtime evidently limits the design space. As already mentioned, the languages must maintain a balance between user friendliness and being easy to compile (to minimise runtime overhead). Features that are normally performed at compile time are lacking - for example, GLSL provides no way to share code between shaders, meaning the same algorithms are often implemented over and over in different shaders. HLSL improves over this slightly by allowing C-style includes, but that is still a long way from the usability of a modules system. The type systems in both languages are acceptable, lacking more advanced features such as polymorphism and algebraic data types, but still covering the basic cases. However, the major flaw is that since the compilation is performed at runtime, so is the type-checking. If the shader is part of a much larger application, checking for type errors involves waiting until the shader is used! This issue detracts from the advantages of having a type system at all, as the rapid development cycle of fixing type errors is not possible.

Compatibility is a large annoyance with existing shader languages. HLSL is the worst offender, only being compatible with the closed-source DirectX, which itself is only available on Microsoft operating systems. Khronos Group's GLSL is not much better either, which is only supported by OpenGL. Neither of these shader languages are natively compatible with Vulkan or Apple's Metal. In practice, this means that the same shader may have to be re-written several times over in slightly different languages, in

order to support the multiple graphics APIs you may want to target[1].

GLSL and HLSL fall short in another form of compatibility too. The graphics pipeline has multiple programmable segments, each with their own individual shader program. The passing of information between the different shaders in the pipeline is core to creating complex graphical effects. Of course, the shader programs must be written in a compatible way - if the first shader in the pipeline passes on some vertex coordinates, the next shader along must be written to expect those coordinates as an additional input. Despite the importance of this compatibility, both GLSL and HLSL choose to define the different shader stages independently, being written in separate files. Admittedly, this approach does have some merit. Shader stages can be mixed and matched, for example, the same vertex shader could be used with two different (but compatible) fragment shaders. This saves a slight amount of storage space and computer memory, and is presumably the reason why GLSL and HLSL originally chose to keep stages separate. In reality, the saved memory is negligible on modern machines. Furthermore, it is rare that an entire shader may be reused in completely different pipelines, and so this advantage is rarely put into effect. Instead, a unified definition of all shader stages would be preferable. Defining all stages in the same file allows type-checking to enforce compatibility, something that is not possible with the separate definitions (even with the help of a validator). This eliminates a potent runtime error, helping increase usability.

Debugging is a tricky aspect of graphics programming - mistakes by the programmer are rarely recognised by an error, and instead the problem must be deduced by what is being rendered incorrectly. While this is mainly the fault of the graphics APIs, having confidence in the behaviour of shaders can help limit the scope of where the mistake could've occurred, making debugging considerably easier. The prominent shader languages definitely fall short in this area. Testing a shader involves integrating into a larger application, which is not particularly helpful since the application could contain bugs of its own. Documentation for builtin functions can only go so far, and sometimes the programmer simply wants to test whether their algorithm is performing as expected or not. Debuggers do exist, but anecdotal experience has proven them to be unpleasant and buggy - GLSL-Debugger[2] is endorsed by OpenGL themselves, yet their website claims that it is a major work in progress and does not support newer versions of OpenGL.

3.2 A Functional Language

3.2.1 Why Functional?

The most stark difference between the design of Ictora and existing shader languages is the choice to follow a functional paradigm. All other existing shader languages, most notably GLSL and HLSL, are procedural languages resembling C - why depart

[1] Supporting multiple graphics APIs is commonplace in the video game industry, where DirectX is often used for Windows builds, and OpenGL used for other platforms.

[2] Available at: http://glsl-debugger.github.io/

from the tried and true design? The answer to this question is manyfold.

Firstly, a functional language brings something new to the shader ecosystem. There are no existing functional shader languages, and so Ictora naturally develops a use case for those programmers who prefer programming in functional style, whereas development of a new procedural language would likely just be overshadowed by existing languages. Functional shader programming also comes with all of the commonly cited benefits on general purpose functional programming: code reuse, strong type systems and equational reasoning to name a few. Furthermore, these strengths all naturally help tackle weaknesses previously outlined with existing shader languages.

Technical implementation details aside, functional programming is also rather befitting when considering its context within the GPU pipeline. The pipeline is the combination of many mathematical transformations, equivalently viewed as the composition of several pure functions, one for each stage. Composition is a core concept in functional programming, and purity aligns excellently with the functional paradigm where all functions are pure by default. Avoiding the need for input and output removes much of the implementation complexity that would be present in a general purpose programming language (where IO is indispensable).

3.2.2 The First Functional Shader Language

You may wonder why has no one has undertaken a similar project before. Being such a great fit in theory, why has noun already developed a functional shader language? Foremostly, development of new shader languages has only became feasible very recently, with the development of SPIR-V in 2014. Attempting to target the higher level existing languages (GLSL and HLSL) would be extremely unpleasant, given their technical limitations and evolving specification. Most of the effort since the development of SPIR-V has been put into developing cross-compilers from GLSL and HLSL to SPIR-V, so that they can be used in conjunction with Vulkan, which solely supports SPIR-V. Development of new languages would certainly be put on the back foot, since GLSL and HLSL are already extremely widespread.

Technical considerations are also a deterrent from undertaking the development of a functional shader language. The environment under which shaders are executed is very minimal, and so naturally language designers have gravitated to lower-level languages which map easily onto assembly and IR. Functional languages are often implemented with complex runtime systems, which is simply not possible on the GPU (or at least the components which are exposed to shaders). These difficulties and how Ictora tackles them are expanded upon in the next section.

3.3 Compiling Lambdas to the GPU

3.3.1 Typical Implementation

Before discussing how we might attempt to compile Ictora to the GPU, let's first consider how we might achieve the same thing on the CPU. Fortunately, this a well researched area. Suppose we are trying to compile a Haskell-like language down to a pseudo-assembly.

The most obvious approach is to convert each top-level Haskell function to a matching assembly procedure. At first, this appears to work well.

```
add : (Int, Int) → Int
add (x, y) = x + y
```

```
add:
    iadd arg1, arg2
    mov res, arg1
    return
```

Functions can be passed to other functions by simply passing their address. However, we run into a problem when we try to compile functions which return other functions. We cannot simply return an address, as the function that we return may be different depending on what argument was passed! Consider the definition of a curried addition:

```
add : Int → Int → Int
add x = λy ⇒ x + y
```

If we were to pass 5 as the first argument, the function should return the function $\lambda y \Rightarrow 5 + y$. If we were to pass 42 instead, we return a different function, specifically $\lambda y \Rightarrow 42 + y$. We cannot model the return result as an address pointing to some static code, since the function's body is not known until the argument is passed. One option is to generate the assembly code at runtime, and then return a function pointer to it - while this would work, runtime code generation is complex and inefficient. Fortunately, more approachable solutions exist.

First, observe that the structure of the resulting function is identical aside from the substituted integer. We always want to return code of the following form:

```
returnFunc:
    iadd arg1, <value for x>
    mov res, arg1
```

```
    return
```

The idea is to always return a pointer to the same static code, and additionally return an environment which tells us the values for these "missing" values. In our case, we return the address of `returnFunc` and a block of memory which tells us the value for `x`. The amount of values that are mapped in the environment is not known to the callee, and so we must heap allocate the environments. This idea is known is a closure.

Closures are underlying almost every compiled functional language implementation[34] in one form or another. Some compilers make use of abstract machines, see SECD machine[1], to provide an easy method of compiling lambda calculus programs to a closure representation.

3.3.2 GPU Limitations

Compiling a functional language down to SPIR-V on the GPU is much less pleasant than traditional CPU implementations. As SPIR-V wishes to maintain compatibility with as many GPU architectures as possible, it only exposes a very small subset of features which are sure to be present (and efficient) on every piece of hardware. Presumably due to this reasoning, SPIR-V provides no support for dynamically allocated memory or function pointers - two features that are core to the standard implementation of a functional language. Furthermore, SPIR-V functions prohibit recursion entirely as even the existence of a call stack is unassured!

Needless to say, this causes serious complexity. The classic implementation using closures is impossible, as it requires the closure's environment is allocated on a heap, which we do not have. Many alternative ideas were attempted, but all were thwarted by some missing feature.

The most powerful observation to realise here is that SPIR-V is not actually Turing-complete: we do not have access to a resizing call stack or dynamic heap memory, and so all of the storage space we require must be declared up front in terms of variables. SPIR-V is in fact very computationally similar to a finite automata[3], and hence Ictora must also be memory bounded and non-Turing-complete. This knowledge majorly influences the design of Ictora and its compilation. Features such as general recursion are in fact fundamentally impossible, and so their compilation need not be considered.

This marks the second important observation - most typed lambda calculi without general recursion are strongly normalising, meaning their evaluation always terminates. STLC, System F[3] and the elementary affine lambda calculus[11] are all examples of this. If we base Ictora upon any of these calculi, then we can take advantage of the strong normalisation and evaluate the calculus at compile time - previously impossible due to the uncertainty of termination when general recursion is involved. This revelation means that we do not actually need to represent lambda expressions

[3]The slight difference being that some SPIR-V programs do not terminate.

on the GPU at all, as they can be reduced at compile time. The resulting program is simply a mathematical expression void of lambda terms, trivial to convert to SPIR-V.

3.4 Language Features

3.4.1 Constructs

The design of the Ictora language itself is fairly straight forward. It extends the simply typed lambda calculus with several comfort features and adds primitives required for writing shaders. We detail the individual features below:

Primitive Maths A large part of shader programming is the manipulation of numbers and vectors, and so Ictora provides a range of primitive operations for manipulating them.

Top-Level Functions Being able to declare global functions enables the programmer to modularise and re-use common pieces of code in multiple places. Furthermore, it enables us to write multiple shaders stages in the same file, defining each as a separate function.

Let Bindings Allows matching on intermediate values and reduces code duplication, like top-level functions do. It differs from complete functions since the scope is only local.

Pairs Useful for returning multiple values from a single function. This feature is essential, as shader stages need to return two things: their raw output, and auxiliary information to pass on to the next stage.

Pattern Matching Let bindings support matching on patterns, which can be used to extract components of composite types. This is an elegant alternative to defining lots of primitive functions, and extends nicely if the language were to be extended with algebraic data types.

For reference, an example program is shown below which demonstrates most of these features:

```
vert : Vec4 → (Vec4, Float)
vert = fn pos ⇒
           let [_, y, _, _] = pos
           in (map_x (add 0.3) pos, y)

frag : Float → Vec4
frag = fn green ⇒ [0.2, green, 0.2, 1.0]
```

An Ictora program is centred around two function definitions: `vert` defines the vertex shader, and `frag` defines the fragment shader (other shader types do exist, but they are significantly more niche). The type signature of the shader functions is notable. A vertex shader always has a type of `Vec4 →(Vec4, a)`, and a fragment shader always has a type of `a →Vec4` - The `a` can be any type of the programmer's choice, as is the information passed along the pipeline. Importantly, both definitions must choose the same type for `a` - this enforces their compatibility, something not possible in GLSL or HLSL.

3.4.2 Syntax

Ictora uses very conventional syntax for a functional language, aiming to be as familiar as possible to those familiar with popular general purpose functional languages. Function applications are denoted by white space, lambda expressions are introduced with the `fn` keyword, and so on. Ictora is white space sensitive too, however it is not as strict as some other languages like Haskell. The rule of thumb is that continuations of an expression can be placed on a newline, but must have a non-zero indent. For example, the following is syntactically valid:

```
f : Float → Float
f = fn x ⇒
    x
```

While this is not:

```
f : Float → Float
f = fn x ⇒
x
```

This is so we can easily distinguish between the definition of a new top-level function and the continuation of an existing definition on a newline.

Since most of the syntax is self-explanatory, we simply detail the grammar below (whitespace rules are omitted for simplicity):

⟨*type*⟩ ::= ⟨*type2*⟩
 | ⟨*type2*⟩ '->' ⟨*type*⟩

⟨*type2*⟩ ::= ⟨*identifier*⟩
 | (⟨*type*⟩)
 | (⟨*type*⟩, ⟨*type*⟩)

⟨*pattern*⟩ ::= ⟨*identifier*⟩
 | '_'

| '(' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ ')'
| '[' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ ']'

⟨*expr*⟩ ::= 'fn' ⟨*identifier*⟩ '=>' ⟨*expr*⟩
| ⟨*expr2*⟩

⟨*expr2*⟩ ::= ⟨*expr2*⟩ ⟨*expr3*⟩
| ⟨*expr3*⟩

⟨*expr3*⟩ ::= ⟨*identifier*⟩
| ⟨*number*⟩
| '(' ⟨*expr*⟩ ')'
| '(' ⟨*expr*⟩ ':' ⟨*type*⟩ ')'
| 'let' ⟨*pattern*⟩ '=' ⟨*expr*⟩ 'in' ⟨*expr*⟩
| '(' ⟨*expr*⟩ ',' ⟨*expr*⟩ ')'
| '[' ⟨*expr*⟩ ',' ⟨*expr*⟩ ',' ⟨*expr*⟩ ',' ⟨*expr*⟩ ']'

⟨*func_defn*⟩ ::= ⟨*identifier*⟩ ':' ⟨*type*⟩ ⟨*newline*⟩ ⟨*identifier*⟩ '=' ⟨*expr*⟩

⟨*program*⟩ ::= ⟨*func_defn*⟩
| ⟨*program*⟩ ⟨*func_defn*⟩

3.5 Type Checking

3.5.1 Typing Rules

Before we can define the typing rules, we must first define the semantics of pattern matching. This is done through the definition of the partial function *patternVars*. Formally, the result of $patternVars(pat, A)$ is the variable-type mappings that should be added to the context if we match a value of type A against the pattern pat. Cases for which *patternVars* is not defined are type errors - it does not make sense to match with the pattern of (x, y) with a value that isn't a pair, for example.

$$
\begin{aligned}
patternVars(\_, A) &= \emptyset \\
patternVars(x, A) &= x : A \\
patternVars((p_1, p_2), (A, B)) &= patternVars(p_1, A) \\
&\quad \cup\ patternVars(p_2, B) \\
patternVars([p_x, p_y, p_z, p_w], Vec4) &= patternVars(p_x, Float) \\
&\quad \cup\ patternVars(p_y, Float) \\
&\quad \cup\ patternVars(p_z, Float) \\
&\quad \cup\ patternVars(p_w, Float)
\end{aligned}
$$

With the definition of pattern matching concluded, the typing rules are detailed below.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textit{T-Var} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\ \textit{T-App}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x\,.\,e) : A \to B}\ \textit{T-Lam} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) : A}\ \textit{T-Ano}$$

$$\frac{}{\Gamma \vdash \text{number} : \textit{Float}}\ \textit{T-Lit} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : (A, B)}\ \textit{T-Pair}$$

$$\frac{\Gamma \vdash e_1 : \textit{Float},\ e_2 : \textit{Float},\ e_3 : \textit{Float},\ e_4 : \textit{Float}}{\Gamma \vdash [e_1, e_2, e_3, e_4] : \textit{Vec4}}\ \textit{T-Vec}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \cup \textit{patternVars}(pat, A) \vdash e_2 : B}{\Gamma \vdash \textit{let}\ pat = e_1\ \textit{in}\ e_2 : B}\ \textit{T-Let}$$

3.5.2 Type Inference

Why Infer Types?

Type inference was considered a core feature during the design of Ictora. In general, type inference is the process of automatically deducing the type of an expression without being explicitly told. In functional languages, this means that the arguments to lambda expressions do not need to be annotated with their type - this makes the language significantly easier to read and write. Compare the two equivalent functions written In Ictora, one which annotates each lambda with its type, and the other that doesn't:

```
-- With annotations
f = fn g : Int → Int ⇒ fn x : Int ⇒ g x

-- Without annotations
f = fn g ⇒ fn x ⇒ g x
```

However, not all type systems have a decidable algorithm for complete type inference - System F, the fully polymorphic lambda calculus, is one such example. In those cases, we often write algorithms which can infer the types of the most common expressions, but require a little manual annotation in the more difficult cases.

A Bidirectional Approach

Bidirectional type checking is a general technique for constructing a partial type inference algorithm from raw typing judgements. It revolves around two mutually recursive functions: one which infers the type of an expression, and another which checks that an expression has a given type. These two functions are said to synthesise and inherit types, respectively. In a nutshell, the algorithm traverses the AST, checking types when it knows what the type should be, and inferring types when it doesn't.

Algorithms derived in this manner have many advantages[10]. Firstly, high quality error messages are easily implemented. The recursive nature of bidirectional type checking means that the source of the error can be pinpointed, and hence an accurate error message produced - something that cannot be said about unification based approaches. The approach also adapts remarkably well to the addition of new language features, making it ideal for languages whose specification is evolving. Of course, every technique comes with drawbacks. Bidirectional type checking algorithms require some type annotations (which can be in the form of a top level function type declaration), whereas algorithms like Hindley-Milner[31] can manage with none at all. However, annotating the types of top level functions are often all that is required, and even in languages where full type inference is present (such as Haskell), most experienced programmers provide type annotations regardless. All of these reasons influenced the decision to use bidirectional typing in Ictora.

Inference Rules

The algorithm is described with rules which look very similar to typing judgements, except instead of the usual colon for declaring an expression has a type, we use $\uparrow$ and $\downarrow$ instead. Inference rules are represented by $\uparrow$, and checking rules are represented by $\downarrow$. More formally:

$\Gamma \vdash e \uparrow A$	In context Γ, we can infer the type of e to be A.
$\Gamma \vdash e \downarrow A$	In context Γ, we can verify that a valid typing for expression e is indeed A.

The rules themselves describe a syntax-directed algorithm - implementing the algorithm is as simple as looking for a rule whose conclusion matches our goal, and recursing on the premises. The full bidirectional typing rules are described below.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \uparrow A}\ BT\text{-}Var \qquad \frac{\Gamma \vdash e_1 \uparrow A \to B \qquad \Gamma \vdash e_2 \downarrow A}{\Gamma \vdash e_1\ e_2 \uparrow B}\ BT\text{-}App$$

$$\frac{\Gamma \vdash x : A \vdash e \downarrow B}{\Gamma \vdash (\lambda x\,.\,e) \downarrow A \to B}\ BT\text{-}Lam \qquad \frac{\Gamma \vdash e \downarrow A}{\Gamma \vdash (e : A) \uparrow A}\ BT\text{-}Ano$$

$$\frac{}{\Gamma \vdash \text{number} \uparrow \mathit{Float}}\ \mathit{BT\text{-}Lit} \qquad \frac{\Gamma \vdash e_1 \uparrow A \qquad \Gamma \vdash e_2 \uparrow B}{\Gamma \vdash (e_1, e_2) \uparrow (A, B)}\ \mathit{BT\text{-}Pair}$$

$$\frac{\Gamma \vdash e_1 \downarrow \mathit{Float},\ e_2 \downarrow \mathit{Float},\ e_3 \downarrow \mathit{Float},\ e_4 \downarrow \mathit{Float}}{\Gamma \vdash [e_1, e_2, e_3, e_4] \uparrow \mathit{Vec}4}\ \mathit{BT\text{-}Vec}$$

$$\frac{\Gamma \vdash e_1 \uparrow A \qquad \Gamma \cup \mathit{patternVars}(\mathit{pat}, A) \vdash e_2 \uparrow B}{\Gamma \vdash \mathit{let}\ \mathit{pat} = e_1\ \mathit{in}\ e_2 \uparrow B}\ \mathit{BT\text{-}Let}$$

$$\frac{\Gamma \vdash e \uparrow A}{\Gamma \vdash e \downarrow A}\ \mathit{BT\text{-}CheckInfer}$$

3.6 Putting It All Together

3.6.1 Interpreter

One of the main deliverables for this project is an interpreter. While the strong type system of Ictora catches programs with type errors, programs are still prone to semantic mistakes, such as misunderstanding the purpose of a built-in function. The main method of tackling these semantic errors is through testing - in the context of shaders, this means integrating the shader into a full graphics application and running it, and checking the result is as expected. This workflow is unideal, and so Ictora includes an interpreter so that programmers can test their programs without running them entirely. The purity of the language allows snippets of a full program to be extracted and tested in isolation, without worrying that the surrounding code influences the result. Furthermore, interpreters serve as a very practical form of documentation: sometimes textual documentation isn't always clear, and the best way to understand is to try the function with some test inputs. The interpreter facilitates all of this behaviour, and is core to increasing the debuggability of the Ictora language.

The interpreter's design is very straight forward. It has two main commands: typing in an expression evaluates it, brining it to normal form, and prefixing the expression with `:t` returns its type. To allow the programmer to debug their shader programs conveniently, the interpreter can also be called with a command line argument of an Ictora file, which is loaded into scope. The user can then call the functions in their shader to test them easily, without the need to manually copy them into the interpreter.

3.6.2 Compiler

The compiler is the main result of the project. It takes an Ictora source file, compiles it to SPIR-V, and writes the resulting SPIR-V assembly code to a file. The compilation process is just the composition of the many ideas we have already discussed.

1. Parse the Ictora source file, exiting early with an error message if parsing fails.
2. Type check the program using the bidirectional type inference algorithm. Additionally, as part of the type checking process, ensure the existence of the *vert* and *frag* entry points and ensure they have type signatures which are compatible with each other. If type checking throws an error, we print the error message and exit early.
3. Extract the definitions for the shader entry points as a single expression, collapsing other top level functions they make use of into *Let* bindings.
4. Normalise/Evaluate the the entry point expressions.
5. Convert the expressions to SPIR-V code.
6. Integrate both of the shader entry points into a single SPIR-V module.
7. Call the external SPIR-V assembler, writing the final result to a binary file.

The beauty of this compiler pipeline is that if the program passes type checking, the compiler is guaranteed not to encounter an error further down the pipeline. This allows for cohesive error reporting, and also helps keep the code clean as error messages can be produced in a centralised place.

Chapter 4

Implementation

4.1 Syntax and Parsing

4.1.1 The AST

Central to the implementation of any programming language is the definition of the abstract syntax tree. It allows the compiler or interpreter to easily manipulate the structure of the code, without dealing directly with the textual representation. As Idris is a functional language, we naturally represent the AST through a series of recursive data types. First, we define the data type representing types:

```
data STy : Type where
    NamedTy : Name → STy
    (⇝) : STy → STy → STy
    PairTy : STy → STy → STy
```

Primitive types are universally represented by their name, checked later during type checking to ensure that the name is valid. This is preferred over defining individual constructors for each primitive, as doing so would require the parser to be modified upon the addition of a new data type. Moreover, if Ictora were to be extended with custom data types, the AST would have to separately account for arbitrary user defined type names. The arrow and pair types are both primitive type constructors which combine two existing types.

Expressions are defined as follows, with less interesting constructors being omitted for brevity:

```
data Expr : Type where
    Var : Name → Expr
    Lam : Name → Expr → Expr
```

```
    App : Expr → Expr → Expr
    Ano : Expr → STy → Expr
    Let : Pattern → Expr → Expr → Expr
    ...
```

Note that lambdas are not annotated with their type, which will instead be inferred through the bi-directional type checking. The `Ano` constructor allows the programmer to manually annotate an expression with its type, helping the inference algorithm resolve ambiguous typings (fortunately, this is rarely necessary). Ictora also supports a basic form of pattern matching in let bindings, which is used to dismantle composite types such as pairs and vectors. Patterns are also defined recursively, allowing nested pattern matching:

```
data Pattern : Type where
    PEmpty : Pattern
    PVar : Name → Pattern
    PPair : Pattern → Pattern → Pattern
    ...
```

`PEmpty` discards the matched value (written as an underscore), `PVar` binds the value to a name, and `PPair` matches on a pair. Implementing pattern matching allows for a unified method of extracting components of composite types, and removes the need for a plethora of primitive functions to extract the various parts of pairs and vectors.

A program is simply a list of names mapping to expressions.

```
Program : Type
Program = List (Name, Expr)
```

4.1.2 Total Parsing

Parsing is a very well researched area, and the common techniques are well understood. The most common of those techniques for writing a parser in a functional language is parser combinators[22]. At its simplest, we view parsers as functions, which are combined together with each other using higher-order functions known as combinators. This declarative representation leads to very concise and understandable parsing algorithms. Unfortunately, the flexibility of parser combinators does leave them susceptible to some irksome issues, namely non-termination and ambiguity. As the parsers can be combined freely, it is very easy to construct a parser which falls into an infinite loop[1], especially when handling left-recursive grammars. Fortunately for us, recent research has developed an augmentation of traditional parser combinators which ensures non ambiguity and termination through dependent types and conin-

[1]This is especially important in a proof language like Idris which requires functions to terminate.

duction principles[2, 13]. In this project, we use *TParsec*[2], a port of the popular Agda library *Agdarsec*[2].

As the main grammar for expressions is large and complex, let's instead consider the parsing of types as an example. Below is the grammar we're trying to parse.

⟨*type*⟩ ::= ⟨*type2*⟩
 | ⟨*type2*⟩ '->' ⟨*type*⟩

⟨*type2*⟩ ::= ⟨*identifier*⟩
 | (⟨*type*⟩)
 | (⟨*type*⟩, ⟨*type*⟩)

The grammar is broken down into two non-terminals so arrows unambiguously associate to the right. Attempting to implement the parser using an ambiguous grammar would result in a type error, and so a working parsing implementation is a proof of unambiguity. Although with such a small grammar this fact is trivial, the static guarantees provided by *TParsec* are very helpful when dealing with larger and more complex grammars. The translation of the above grammar into Idris code is quite straight forward.

```
parseType : All (Parser' STy)
parseType = fix _ $ λrec ⇒

            let parsePair = Combinators.map (uncurry PairTy) $
                            and (between (char '(') (char ',') rec)
                                (between whitespace (char ')') rec)

                parseType' = alts [ map NamedTy identifier
                                  , parens rec
                                  , parsePair ]

                parseArrow = cmap (⇝) $ between whitespace
                                                whitespace
                                                (string "→")

            in chainr1 parseType' parseArrow
```

The recursive calls are encoded using a specialised fix-point operator which is always guaranteed to terminate. We use the `chainr1` combinator to parse a non-empty list of types separated by the right associative arrow operator. As you can see, the implementation does not require any manual intervention to prove termination or unambiguity - these are properties are automatically inferred by the type system. Parsers for expressions, patterns and programs are all also implemented using the same technique.

[2]https://github.com/gallais/idris-tparsec

4.2 Type System

4.2.1 Concrete and Syntactical Types

Implementation of the type system makes heavy use of dependent types. We distinguish between well-typed and ill-typed terms by embedding the typing rules on the type level. Not only does this help prove the correctness of the type checker, but can also be used to show that any transformations performed on the AST do not violate type soundness. This approach intertwines formal verification techniques with the implementation, resulting in an elegant formulation which matches closely with the theory.

Although we have already defined syntactical types, many invalid types are representable since we use a string to represent all named types. The presence of syntactical types is still important for the parser, but in addition we define a more concrete representation for our type checking purposes. We will later formalise the relationship between our two definitions of types, but the correspondence should be obvious.

```
data Ty : Type where
    FloatTy : Ty
    (⇝) : Ty → Ty → Ty
    PairTy : Ty → Ty → Ty
    ...
```

The definition is the same as our syntactical types, except each primitive has its own constructor. To formalise the relation between these two types, we take advantage of the Curry-Howard isomorphism and define a data type parameterised by both. An element of this type is a proof that the two types are equivalent.

```
data IsType : STy → Ty → Type where
    TFloat : IsType (NamedTy $ unpack "Float") FloatTy
    TArrow : IsType t a → IsType u b → IsType (t ⇝ u) (a ⇝ b)
    TPair  : IsType t a → IsType u b → IsType (PairTy t u) (PairTy a b)
    ...
```

4.2.2 Typing Contexts

Names or Indices?

An interesting decision is whether to use a named or indexed representation for variables. Named representations match what we are parsing, and so we can use our already defined AST throughout the entire program. Semantics can be defined in terms of the same syntax that the program is written in, removing the need for multi-

ple layers and conversions. However, De Bruijn indices are significantly easier to work with when defining semantic operations such as substitution[6][3]. Furthermore, most literature defines the more complex algorithms in terms of indices too, such as the NbE algorithm we implement later[16, 25]. Taking all of this into account, a named representation was chosen: the extra implementation difficulty is surmountable, and the result is a simpler and more understand model.

Contexts and Lookups

Give our choice of a named representation, typing contexts are an ordered mapping between names and types.

```
data Context : Type where
    Empty : Context
    (::) : (Name, Ty) → Context → Context
```

New variables are extended to the left of the context, which shadow previous variables with the same name. We define when a name is in scope with the `Lookup` type.

```
-- "Lookup name con a" is a proof that lookup "name" in context "con"
-- gives a type "a".

data Lookup : Name → Context → Ty → Type where
    Here  : Lookup x ((x, a) :: _) a
    There : Not (x = y) → Lookup x con a → Lookup x ((y, _) :: con) a
```

The two constructors account for all possible locations of the variable. The `Here` constructor can be used when the mapping is the first element of the context, and the `There` constructor can be used to extend the context used in the proof. We carefully design the `There` constructor to account for shadowing - we can only extend the context of a `Lookup` if we do not shadow the variable we are interested in, expressed in the form of the `Not (x = y)` constraint.

Fresh Names

A definition of name freshness is also an important concept. A fresh variable is one which does not already appear in the context, and hence can be appended to the context without risk of shadowing other variables.

```
-- "Fresh name con" means that "name" is fresh in the context "con"

data Fresh : Name → Context → Type where
```

[3]Formally verified capture avoiding substitution can be quite tricky to implement.

```
    FreshEmpty : Fresh name Empty
    FreshExt   : Not (x = y) → Fresh x con → Fresh x ((y, _) :: con)
```

Context Weakening

Another important concept in our implementation of typing contexts is the notion that one context is *weaker* than the other. In essence, any expression which makes sense under a weaker context will also make sense under a stronger one. This means that every variable lookup accessible in the weaker context must also be present in the strong one - this definition is translated very literally into Idris code.

```
-- "Weaker con con'" is a proof that the context "con" is weaker than
-- the context "con'"

Weaker : Context → Context → Type
Weaker con con' = {x : Name} → {a : Ty} →
                  Lookup x con a → Lookup x con' a
```

Our definition of weaker is not strict, as every context is weaker than itself (a reflexive relation).

```
weakerRefl : Weaker con con
weakerRefl = id
```

4.2.3 Typing Rules

Intrinsic vs Extrinsic Typing

When defining well typed terms in a proof language, there are two dominant techniques: intrinsic and extrinsic typing[4]. An intrinsic representation parameterises expressions by their type, and embeds the typing rules in the constructors[4, 29]. To help realise this idea more concretely, we give an example below.

```
-- "Expr con a" means an expression which when under the context "con"
-- has type "a"

data Expr : Context → Ty → Type where
    Var : Lookup name con a → Expr con a
```

[4]Not to be confused with intrinsic and extrinsic semantics[36], which concerns whether semantic meaning is defined for all terms or only well typed terms. Personally, I prefer the terms internal and external typing to avoid this confusion, but to remain consistent with existing literature intrinsic and extrinsic will be used.

```
    App : Expr con (a ⇝ b) → Expr con a → Expr con b
    Lam : (name : Name) → (a : Ty) → (body : Expr ((name, a) :: con) b)
       → Expr con (a ⇝ b)
```

This representation is exceptionally concise. Functions which manipulate the AST must mandatorily preserve well-typedness, as intrinsic typing makes it impossible to construct an ill-typed term. In literature, this approach is dominant[25, 37, 39] as implementations typically require significantly less lines of code than their extrinsic counterparts. However, when used in the context of a real-world compiler, some downsides become apparent.

As an intrinsically typed AST can only represent well typed expressions, it cannot be used as the direct output of parsing (as the programmer may write an ill typed but syntactically valid program). The most pragmatic solution to this problem is to to maintain two separate definitions of the AST. One AST will be purely syntactical (like the one we defined for Ictora), and will be used as the output of parsing. The other AST will be the intrinsically typed representation, like the one seen above. Conversions between the two forms of AST will take place when a specific representation is required. While this approach does work, it requires near duplication of the same AST - additions of new language constructs involve writing similar constructors twice. Furthermore, much more complex typing rules become unrepresentable with this method. Calculi where types may contain value level constructs (such as λ_{Π}) would require the `Expr` data type to be parameterised by itself, resulting in an inconstructible data type. While the current Ictora type system *is* intrinsically representable, future versions may not be.

Instead in Ictora we chose an extrinsically typed representation, which considers the typing derivation as a separate entity. While this method typically compels a lengthier implementation, it overcomes the disadvantages of intrinsic typing outlined previously, and hence is preferred. We define the data type representing a typing derivation below.

```
-- "HasType con e a" means under the context "con", expression "e" has
-- type "a"

data HasType : Context → Expr → Ty → Type where
    TVar : Lookup nom con a → HasType con (Var nom) a

    TLam : HasType ((x, a) :: con) body b
        → HasType con (Lam x body) (a ⇝ b)

    TApp : HasType con f (a ⇝ b) → HasType con x a
        → HasType con (App f x) b

    TAno : HasType con e a → IsType t a → HasType con (Ano e t) a

    TLit : HasType con (Lit n) FloatTy
```

```
    TPair : HasType con x a → HasType con y b
        → HasType con (Pair x y) (PairTy a b)

    TLet : HasType con e1 a → PatternAdds p a extras
        → HasType (extras ++ con) e2 b
        → HasType con (Let p e1 e2) b

    ...
```

`HasType con e a` is equivalent to $con \vdash e : a$ in traditional typing notation. Each constructor of this type corresponds to a typing judgement - premises of the judgement are parameters to the constructor, and the conclusion is the return type.

Pattern Matching

One rule of particular interest is the `TLet` rule. The context under which the body of a `Let` expression is evaluated under depends on what pattern we match on: therefore we need a method of relating patterns to the variables they add to the context. As an example, consider the following two let expressions, identical aside from the pattern they match on.

```
-- Variable pattern
let x = (42.0, -42.0) in ...

-- Pair of variables pattern
let (x, y) = (42.0, -42.0) in ...
```

The first expression adds a single variable `x : (Float, Float)` to the typing context; while the second expression adds two variables, `x : Float` and `y : Float`. In addition, some patterns only make sense when matched on specific types. Trying to match `(x, y) = 1.0` should obviously be a type error. To capture this relation, we define a new data type: `PatternAdds`.

```
-- "PatternAdds pat a extras" means that using pattern "pat" to match on
-- a value of type "a" adds the entries in "extras" to the context.

data PatternAdds : Pattern → Ty → Context → Type where
    PTEmpty : PatternAdds PEmpty a Empty

    PTVar   : PatternAdds (PVar name) a ((name, a) :: Empty)

    PTPair  : PatternAdds p1 a con1 → PatternAdds p2 b con2
        → PatternAdds (PPair p1 p2) (PairTy a b) (con2 ++ con1)
```

```
    ...
```

Bidirectional Typing Rules

Unlike general purpose typing derivations, the bidirectional typing rules do not necessarily need to be defined as a data type. In this project, their purpose is not to mandate what terms are well typed (which is the job of the normal typing rules), but to describe a syntax directed algorithm for inferring the typing derivation of terms. Such an algorithm could be simply realised as a function from terms to an optional typing derivation, where each case corresponds to a bidirectional rule. Regardless, we choose to define the bidirectional typing rules for two reasons. First, as a simple proof that the inference algorithm does indeed follow the bidirectional rules we have already designed. Second, to aid in the production of high quality error messages (which will be expanded on shortly).

The bidirectional inference rules can be represented in a very similar manner to the normal typing rules. We define two mutually inductive data types: `Infer` defines the rules for synthesising the type of a term, and `Check` defines the rules for inheriting a type.

```
-- "Infer con e a" is a proof that under the context "con", we can
-- infer that expression "e" has type "a" using the bidirectional
-- typing rules (synthesis).

-- "Check con e a" is a proof that under the context "con", we can
-- verify that expression "e" is indeed a valid instance of the
-- type "a" (inheritance).

data Infer : Context → Expr → Ty → Type where
    BTVar : Lookup name con a → Infer con (Var name) a

    BTApp : Infer con f (a ⇝ b) → Check con x a
        → Infer con (App f x) b

    BTAno : IsType t a → Check con e a → Infer con (Ano e t) a

    BTLit : Infer con (Lit n) FloatTy

    BTPair : Infer con x a → Infer con y b
        → Infer con (Pair x y) (PairTy a b)

    ...


data Check : Context → Expr → Ty → Type where
    BTLam : Check ((name, a) :: con) body b
```

```
        → Check con (Lam name body) (a ⇝ b)

    BTCheckInfer : Infer con e t → Check con e t
```

Next, we prove the correctness of the bidirectional typing rules by showing we can produce a typing derivation given we managed to infer or check the type.

```
inferToRules : Infer con e a → HasType con e a

checkToRules : Check con e a → HasType con e a
```

The implementations of both of these functions is trivial, and has hence been omitted.

When Type-Checking Goes Wrong

It may seem we finally have all the tools we need to start writing the type checker. We could define the inference and checking algorithms which our bidirectional rules outline, returning either a proof that the term has successfully been inferred/checked, or a string containing the error message. Such an implementation would contain the following definitions.

```
tryInfer : (con : Context) → (e : Expr)
    → Either String (a ** Infer con e a)

tryCheck : (con : Context) → (e : Expr) → (a : Ty)
    → Either String (Check con e a)
```

One flaw with this implementation is the use of a string for error messages. Firstly, it makes error messages effectively uninspectable after they have been created. Suppose, later on in the compilation process we wanted to order the error messages in terms of importance - reidentifying string error messages is crude and difficult. The classic solution to this is to use a data type to represent error messages instead, which is then only converted to a string for printing out.

```
data Error : Type where
    UndefinedVar : Name → Error
    TypeMismatch : Ty → Ty → Error
    ...

tryInfer : (con : Context) → (e : Expr)
    → Either Error (a ** Infer con e a)

...
```

However, from a correctness perspective this is still a little imprecise. The type signature of `tryInfer` ensures that we only return the successful `Right` branch when we have a proof of successful inference, however there is no proof that we only return return the erroneous `Left` branch on uninferable terms. Conceivably, the type checking algorithm could in fact return type errors when the term is in fact inferrable! To ease this understanding, consider the following obviously incorrect, but well-typed implementation of the `tryInfer` algorithm.

```
tryInfer _ _ = Left (UndefinedVar "whoops!")
```

While this example is conspicuously contrived, the same form of mistake could manifest itself much more covertly into a mostly correct implementation. Following the Type-Define-Refine cycle[9], we must augment the type signature of the function to be more precise, making the incorrect implementation an Idris type error. In this case, we need to ensure that we only use the `Left` branch when it is provably impossible to construct a valid derivation. The common approach to this problem in literature[40] is to make the type checking function a decider, where the erroneous cases are not handled by a type error, but a proof that a derivation is impossible. Impossibility (or negation) in Idris is written as a function returning `Void`, much like in propositional logic, where $\neg p \iff p \Rightarrow \bot$. Modifying our inference function to use this approach would result in the following type signature.

```
tryInfer : (con : Context) → (e : Expr)
    → Either ((a ** Infer con e a) → Void) (a ** Infer con e a)
```

While this approach does indeed ensure that only uninferable terms are rejected, instead of the pragmatic error messages, we are left with the opaque function returning `Void`. There is no indication to why the inference may have failed, and hence generating specific error messages is impossible! Neither of these two well-understood methods are suitable for this project, as they both compromise in either practicality or correctness.

We need to represent the erroneous case by a type which is transparent, allowing good error messages to be derived, but also provably uninhabitable if the term has a valid derivation. After attempting a variety of solutions, by far the most effective was to define *well-typed errors*. We develop an error data type which not only includes information that is useful when printing out the error, but also information which proves the existence of the error. For example, in the case of an undefined variable, a proof that the variable does not exist in the context is required. This form of error is easily inspected since it is a data type, and can therefore be subject to case analysis. To prove the second property of interest, which is that it is only inhabitable by terms which do not have a valid derivation, we additionally prove that for any term the coexistence of a type error and valid derivation is impossible. Below is a snippet of the error data type for inference.

```
-- "InferError con e" means that under the context "con", it is not
```

```
-- possible to infer the type of "e" using the bidirectional inference
-- rules.

data InferError : Context → Expr → Type where
    IEVarUndefined : Not (a ** Lookup name con a)
        → InferError con (Var name)

    IEAppL : InferError con f → InferError con (App f _)

    IEAppR : Infer con f (a ⇝ b) → CheckError con x a
        → InferError con (App f x)

    IEAppNotFunc : Not (IsArrow a) → Infer con f a
        → InferError con (App f _)

    ...
```

The data type is parameterised by a typing context and expression, which are the inputs for which the bidirectional inference algorithm could not find a valid derivation. The constructors can be divided into two categories: those that describe errors that would be present in a normal `TypeError` data type (such as undefined variables), and those that propagate the error to an encompassing term. An example of the latter is the `IEAppL` constructor, which states that an inference error on the left hand side of an application implies an inference error for the whole term. To aid your understanding, Figure 4.1 visualises both the AST of an ill-typed term and its corresponding error derivation.

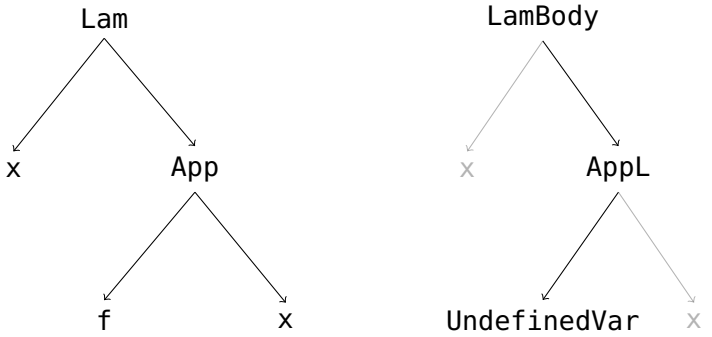

Figure 4.1: On the left, the AST of an ill-typed expression (the variable `f` is undefined). On the right, the structure of the error derivation.

The beauty of this construction is that it satisfies the original constraints with ease. First, it is inhabitable by precisely those terms which do not have a derivation, ensuring the implementation does not accidentally return an error for a term which is in fact well-typed. In addition, producing accurate error messages with this approach is trivial and remarkably effective. As depicted above, the location of the error is precisely encoded, allowing for an accurate description of exactly why and where the error occurs. The textual representation of the error in Figure 4.1 is as follows.

```
In the body of the lambda expression...
On the left hand side of the application...
Undefined variable "f"
```

Of course, there is always room for improvement. One might argue a better error message would suggest a potential solution, perhaps searching the context for similarly named variables since `f` maybe have been a typo (in fact, the production of human-friendly error messages is a topic of research itself[20, 26]). The beauty of this construction is that it is highly adaptable to whatever error message you may want to generate: all relevant information is already embedded into the error type, and so no changes to the type checking algorithm would be needed!

Proving that this error can only be parameterised by uninferable terms is as simple as showing its disjunction with the `Infer` data type.

```
inferErrorCorrect : InferError con e → Infer con e a → Void
```

The type inference procedure can finally be written, whose implementation is trivial due to the the heavy typing constraints.

```
tryInfer : (con : Context) → (e : Expr)
    → Either (InferError con e) (a ** Infer con e a)
```

4.3 Evaluation and Normalisation

4.3.1 Normal Forms

Before concerning ourselves with how we might evaluate our expressions, we must first define precisely what we are trying to achieve. As we know, evaluation of the lambda calculus is simply repeated reduction - a term which no longer has any reductions of interest is said to be in normal form. In our case, we would like to remove all β-redexes and η-expand where possible, as to make the expression as simple to convert into SPIR-V as possible. The resulting expression is said to be in long β-η normal form.

There are two ways we might choose to define a normal form in Idris. One method, outlined in the Lambda chapter of the PLFA book[40], is to define the small-step evaluation semantics as a data type. The data type relates two terms, serving as a proof that the first evaluates to the second; constructors are the reduction rules. Normal forms are simply defined as terms for which we can no longer apply a reduction step - equivalently, when it is no longer possible to construct an instance of the evaluation

data type with our expression as its first parameter. The beauty of this approach is that it is a simple translation of theory into code, as normal forms are represented by definition.

The second approach is to define a predicative data type on well-typed terms which limits what constructors may be present, in order to restrict the expressions to only normal forms. As β-normality is not structurally inductive, we additionally define a notion of neutral terms to aid in implementation. While this implementation is slightly less formal, it comes with practical advantages, particularly its compatibility with the NbE[16, 25] algorithm we will discuss later. It is for precisely this reason that this approach was chosen for Ictora. The definitions are detailed below.

```
data Normal : (e : Expr) → HasType con e a → Type where
    NfNe   : Not (IsArrow a) → Neutral e te → Normal {a} e te

    NfLam  : Normal body tbody → Normal (Lam x body) (TLam tbody)

    NfPair : Normal x tx → Normal y ty
          → Normal (Pair x y) (TPair tx ty)


data Neutral : (e : Expr) → HasType con e a → Type where
    NeVar : Neutral (Var _) _

    NeApp : Neutral f tf → Normal x tx
         → Neutral (App f x) (TApp tf tx)

    NeLit : Neutral (Lit _) _

    NeFst : Neutral x tx
        → Neutral (Let (PPair (PVar name) PEmpty) x (Var name))
                  (TLet tx (PTPair PTVar PTEmpty) (TVar Here))

    ...
```

While the intuition on why the rules are defined as so may not be blindingly obvious, their correctness is checked when we come to perform the conversion to SPIR-V which has no support for lambda expressions.

4.3.2 Structural Normalisation

A function to normalise a lambda calculus term should take any arbitrary well-typed expression, and return an expression of the same type with the additional proof of it being in normal form.

```
normalise : (e : Expr) → HasType con e a
```

```
          → (e' : Expr ** ht' : HasType con e' a ** Normal e' ht')
```

Given the simplicity of lambda calculus evaluation, the implementation of this function initially appears trivial. Given a separate definition of capture-avoiding substitution, case analysing and recursively normalising sub-expressions does indeed yield the correct result. However, such an implementation will not pass Idris's termination checker as it is not structurally recursive. The difficulty arises in the case of applications, as the substitution of a term in normal form into another term in normal form is not necessarily normal itself. For example, the lambda term $(\lambda f\,.\,\lambda x\,.\,f\ x)\ (\lambda x\,.\,x)$ is the application of two terms in normal form, which when β-reduced results in the non-normal expression $(\lambda x\,.\,(\lambda x\,.\,x)\ x)$. This hinderance means we must re-call the normalisation procedure of the result of the β-reduction, ruining structural recursion. In pseudo-code, normalisation of applications looks something like the following:

$$normalise\ (App\ f\ x) = normalise\ (apply\ (normalise\ f)\ (normalise\ x))$$

Since the termination of the STLC is a well known proven result [41, 14], we could simply assert termination to Idris. However, this is hardly in the spirit of dependently typed programming, and leaves our implementation susceptible to human error (the algorithm will only terminate if implemented correctly). Instead, formal proofs of termination for the evaluation of the STLC were researched.

For an algorithm to pass Idris's termination checker, all recursive calls must act on at least one sub-part of the inputs to the function. This means that if we can show a strict decrease in some property of the expressions after performing reductions, we have ourselves a structurally recursive algorithm. Unfortunately, such property of the STLC is not at all obvious. One might think that after a β-reduction, the number of redexes in the expression is strictly less than before, as we've just removed one. Unfortunately, this is not the case. In fact, β-reducing may increase the total number of redexes across the term, as the substitution may place lambdas in places that were previously irreducible. Furthermore, if this were the case then the same logic could be used to prove the termination of the untyped lambda calculus, which we know to be false. This last observation is important, as to avoid whatever proof we come up with also being applied to the obviously non-terminating untyped lambda calculus, we must make use of their only difference: the typing.

This observation was also made by Girard[18], who came up with the notion of a term's degree, which takes into account both the number of reductions *and* the types of the terms being applied in said reductions. As he proves, the degree of a term strictly decreases after β-reduction, giving us the property we were looking for. However, in practice this did not translate well. After numerous attempts to integrate this idea into our existing normalisation algorithm, it became clear that even a successful implementation would make the code incomprehensible and unclear.

Fortunately, this is not the first time that a structurally recursive approach to normal-

isation has been desired. The are two prominent techniques described in literature, hereditary substitutions[24] and normalisation by evaluation[16] (sometimes referred to as NbE). The former augments the substitution procedure to also β-reduce any new redexes it might create, making β-reduction normal form preserving. The latter takes a different approach to normalisation altogether, which will be detailed in the next section. While both approaches have their merits, NbE is considerably more efficient and so is the chosen method for a real-world compiler.

4.3.3 Normalisation by Evaluation

The implementation of a functional language within another functional language can make use of an efficient technique for evaluation known as embedding. The fundamental idea is strikingly simple - convert terms of the AST into terms of the host language, and take advantage that the host language itself knows how to evaluate lambda expressions. The following pseudocode function outlines this idea (glossing over some difficulties):

$$
\begin{aligned}
&embed\ (Var\ name) = name \\
&embed\ (Lam\ name\ body) = \lambda name \Rightarrow\ embed\ body \\
&embed\ (App\ f\ x) = (embed\ f)\ (embed\ x) \\
&embed\ (Lit\ n) = n \\
&...
\end{aligned}
$$

Embedding yields excellent performance[16], and completely removes the need to define things like the awkward capture-avoiding substitution (as all evaluation is handled by the host language).

The weakness with this approach, however, is that some terms in the host language are a lot more opaque than those represented in an AST. Suppose we embed an AST whose root node is a *Lam*: naturally, this would get embedded to a lambda expression in the host language. The problem is that lambda expressions in the host language cannot be simply printed out or inspected, like they could in the AST representation. We have no way of "looking inside" the host lambda and seeing its body, we can only apply it as a black box. This is a serious issue, as the compilation of Ictora requires the normalised AST to be recovered so it can later be converted to SPIR-V. Therefore we need a method of *reifying* expressions in the host language back to an AST representation.

That is the essence of NbE: embed expressions into the host language for efficient evaluation, and then reify the result back into the AST.

4.4 SPIR-V in Idris

4.4.1 A SPIR-V Library

With SPIR-V and Idris both being fairly uncommon and niche languages, it is not unexpected that there are no existing libraries to help us generate SPIR-V. Unfortunately, SPIR-V is not particularly pleasant to output manually, as it has many subtle restrictions that would trip up a standard solution. To maintain good decoupling and clear code, a library was developed to help generate SPIR-V in a non-intrusive way. Since the only purpose of the library is to be used in conjunction with Ictora, to keep matters simple, we only support the small subset of SPIR-V which is needed for the compiler.

SPIR-V programs are written in single static assignment (SSA) form, binding the result of operations to identifiers. These identifiers must be unique across the entire module. Types and constants are incrementally built up, and are also required to be unique - it is not allowed to define two different identifiers which map to 32 bit integers, for example. A snippet of a disassembled SPIR-V program is shown below:

```
        %void = OpTypeVoid
           %3 = OpTypeFunction %void
       %float = OpTypeFloat 32
     %v4float = OpTypeVector %float 4
        %uint = OpTypeInt 32 0
      %uint_1 = OpConstant %uint 1
%_arr_float_uint_1 = OpTypeArray %float %uint_1
%gl_PerVertex = OpTypeStruct %v4float %float %_arr_float_uint_1
%_ptr_Output_gl_PerVertex = OpTypePointer Output %gl_PerVertex
          %_ = OpVariable %_ptr_Output_gl_PerVertex Output
         %int = OpTypeInt 32 1
       %int_0 = OpConstant %int 0
%_ptr_Input_v4float = OpTypePointer Input %v4float
         %pos = OpVariable %_ptr_Input_v4float Input
%_ptr_Output_v4float = OpTypePointer Output %v4float
        %main = OpFunction %void None %3
           %5 = OpLabel
          %18 = OpLoad %v4float %pos
          %20 = OpAccessChain %_ptr_Output_v4float %_ %int_0
                OpStore %20 %18
                OpReturn
                OpFunctionEnd
```

The design of the SPIR-V library revolves around a state monad. The use of a monad allows us to encapsulate the subtleties of SPIR-V generation, allowing implementations using the library to solely focus on the more important parts. The state maintained by the monad is represented by the following type:

```
record BuilderState where
    constructor MkBuilderState
    types : TypeMap
    nextId : Nat
    staticCode : List Instruction
```

First, the `TypeMap` serves as a lookup table for types we have defined. Before defining a type, we first lookup the type in the map to check if we have defined it before, and if so we use the existing definition. This prevents all duplicate type definitions. To efficiently generate new identifiers, we keep track of a natural number in the monad's state. A new identifier is generated by converting the counter to a strinct, and then incrementing the counter. This is a simple and effective way of ensuring that each identifier we generate is unique. Lastly, the record stores a list of "static instructions". This mainly a SPIR-V technicality: some instructions must be placed at the top of the file, but are not necessarily known until we have generated the rest of the code. By storing these instructions as state, we can append to them whenever necessary throughout the generation of the rest of the program.

Aside from this monad, the majority of the library is concerned with the definition of SPIR-V operations as a data type. The definition is rather mundane and large, and so has been omitted.

4.4.2 Lambda Calculus to SPIR-V

With a SPIR-V library in place, the last part of compilation can be completed - turning our Ictora expressions into SPIR-V. The majority of the groundwork has already been completed, so this is a fairly seamless process. We make use of strong normalisation by only defining the conversion for expressions in normal form - since every expression can be normalised, by composition we can convert any expression to SPIR-V. This simplifies our implementation, as our function does not actually need to concern itself with lambda expressions and applications.

Before we move on to the central definition, we must define a few auxiliary data types. The first, `NoArrows` is a proof that a type does not contain any function types.

```
data NoArrows : Ty → Type where
    NAFloat : NoArrows FloatTy
    NAVec : NoArrows VecTy
    NAPair : NoArrows a → NoArrows b → NoArrows (PairTy a b)
    ...
```

It is inhabited by every type except the arrow type, `(a ⇝b)`. Additionally, we define a data type representing that all elements in the typing context satisfy the `NoArrows`

predicate:

```
data BasicCon : Context → Type where
    BCEmpty : BasicCon Empty
    BCCons : NoArrows a → BasicCon con → BasicCon ((_, a) :: con)
```

These two definitions can help us ensure that no high order functions are present during compilation - normalisation ensures that all functions are applied in the expression itself, but does not prevent the presence of functions in the typing context: that is the purpose of these definitions. To keep track of variables in scope, we additionally define an environment which provides values for types in the typing context.

```
data Env : Context → Type where
    Nil : Env Empty
    (::) : Id → Env con → Env (_ :: con)
```

We finally reach the central definition:

```
buildExpr : Env con → (e : Expr) → (te : HasType con e a)
         → Normal e te → NoArrows a → BasicCon con
         → Builder (Id, List Instruction)
```

The type signature is quite descriptive. As input, we take in an environment. This is so we know what value to provide SPIR-V with when we encounter a free variable. Has previously mentioned, we only consider typed expressions in normal form, simplifying the cases we have to consider. Since SPIR-V does not allow the definition of lambda expressions, the expression we compile and its context must not contain any function types. Finally, as result, we return the SPIR-V identifier of the expression we have just generated, and the list of SPIR-V instructions we produced along the way.

The implementation of this function is a simple matter of case analysis. The full definition is very large, so let's just consider the small snippet which converts pairs as an example.

```
buildExpr {a = PairTy a b} env (Pair x y) (TPair tx ty)
                           (NfPair nx ny) (NAPair nax nay) bsc = do

    (lId, lInstrs) ← buildExpr env x tx nx nax bsc
    (rId, rInstrs) ← buildExpr env y ty ny nay bsc

    res ← freshId
    t ← type (PairTy a b) (NAPair nax nay)
    pure (res, lInstrs ++ rInstrs ++
               [MkInstrWithRes res $ OpCompositeConstruct t [lId, rId]])
```

The algorithm is verbose, but fundamentally simple. The monad provided by the SPIR-V library allows the definition to be expressed in a simple manner - although we haven't discussed every definition and function used in this example, the general technique can still be understood. Sub-expressions are recursively built, and then glued together with a final SPIR-V instruction. This pattern is continued through the definition of `buildExpr`, resulting in a readable conversion which is easy to debug and modify.

Chapter 5

Evaluation

5.1 A Comparison with GLSL

A good way to evaluate the success of this project is where it began - existing shader languages. Dissatisfaction with existing shader languages was the primrary driving force behind the project, with criticisms of GLSL and HLSL heavily influencing design decisions. Now, we provide a side by side comparison of Ictora and GLSL version 330. Undeniably, GLSL is a considerably more mature language and hence is more featureful (particularly in terms of breadth) - this is to be expected, and instead we are more interested in the theoretical capabilities and design of each language.

5.1.1 Syntactical Differences

First, we compare an equivalent pair of vertex and fragment shaders in GLSL and Ictora. First, the GLSL program. It is split across two files, `vert.glsl` and `frag.glsl`.

```
/* FILE: vert.glsl */

#version 330 core

layout (location = 0) in vec4 pos;
out float greenVal;

void main()
{
    vec4 pos2 = pos;
    pos2.x += 0.3;

    greenVal = pos.y;
    gl_Position = pos2;
}
```

```
/* FILE: frag.glsl */

#version 330 core

in float greenVal;
out vec4 outColor;

void main()
{
    outColor = vec4(0.2, greenVal, 0.2, 1.0);
}
```

And now the equivalent Ictora program is detailed below.

```
vert : Vec4 → (Vec4, Float)
vert = fn pos ⇒
           let [_, y, _, _] = pos
           in (mapX (add 0.3) pos, y)


frag : Float → Vec4
frag = fn green ⇒ [0.2, green, 0.2, 1.0]
```

Syntactically, the Ictora program is considerably more concise. The type signature of the entry points is used to neatly convey the inputs and outputs to the shaders, avoiding the need for the in-out declarations present in the GLSL shader. The higher-order functions such as `mapX` allow vectors to be modified in place, without the need of numerous intermediate assignments. Perhaps the biggest difference between the layout of the two languages is that both shaders are defined in the same file in Ictora. While the primary reason for this design is so the type checker can enforce compatibility between the vertex and fragment shaders, it also has the advantage of making shader programs much more readable - the whole story can be quickly understood by opening only a single file.

5.1.2 Not Just a Pretty Face

The Ictora language also boasts some practical advantages over the traditional GLSL shader - namely, the enforced shader compatibility by the type system. To demonstrate the usefulness of this, we contrive two incompatible shaders, and compare where in the workflow the problem is identified. In keep things simple, we will base these new shaders on the one we compared above.

Suppose that the programmer makes the mistake of defining the wrong inputs for the

fragment shader. Instead of having the input as a float, they mistakenly take in a `Vec4` instead. We modify the GLSL fragment shader as follows:

```
#version 330 core

in vec4 greenVal;
out vec4 outColor;

void main()
{
    outColor = greenVal;
}
```

The fragment function in the Ictora shader also gets changed in an equivalent way:

```
frag : Vec4 → Vec4
frag = fn green ⇒ green
```

Considering the fragment shaders on their own, both these changes are valid. However, it has made the fragment shader incompatible with the original vertex shader (which outputs a `Float` rather than a `Vec4`). Let's see how the programmer ends up identifying this mistake for both Ictora and GLSL.

First, GLSL. As GLSL does not need to be compiled beforehand, the programmer will test their shaders by running a graphics application which makes use of them. This is already unideal, since the application may not make use of the shaders immediately. Imagine if you're designing a graphical effect for the final boss of a video game - aside from playing through the entire game, the only reasonable way to test the shader is to modify the application code to shortcut to the final boss: not an efficient workflow. The issue is finally identified when the fragment and vertex shaders are linked together with the following error message[1]:

```
error: vertex shader output `greenVal' declared as type `float', but
       fragment shader input declared as type `vec4'
```

As you were hopefully expecting, Ictora identifies this problem much more quickly. Ictora is a compiled language, and so the first step to testing the shader is seeing if it passes the type checker. In this case, it doesn't with the following error:

```
Incompatbile vertex and fragment shader definitions. 'vert' propogates a
```

[1]There is actually a further caveat: the error message is only printed in the case that the programmer specifically checks the linking status for errors. In the case that they do not check for errors, they are instead presented with a white screen with no indication to what could've gone wrong. However, this was categorised as more an OpenGL issue rather than a problem with GLSL itself.

```
  'Float' down the pipeline, whereas 'frag' is expecting a 'Vec4'.
```

Much better!

5.2 A Good Technique?

5.2.1 Functional Shaders

The biggest difference between Ictora and existing shader languages is the choice of paradigm. While GLSL and HLSL are procedural C-like languages, Ictora is purely functional. The functional paradigm appeared to be an excellent fit in theory, with the only caveat being the difficulty of compilation - in hindsight, did this hold true? I am pleased to say that it certainly did. In fact, it turned out to be a better match than originally envisioned. As already demonstrated, Ictora shaders are extremely concise: a big part of why this is possible is the functional paradigm. Representing shaders as functions works remarkably well, as almost all operations that need to be performed are pure functions themselves. Function purity also allows algorithms to be isolated and tested in an interpreter, without worrying about whether other pieces of code may effect the output. This allows for easy testing, not possible with other procedural shader languages. Input and output, a notoriously awkward part of functional programming, is not necessary in shaders.

Originally, it was supposed that the cost of these advantages was a much more difficult implementation, and worse runtime performance. The former proved to be true - devising a compilation method to the hostile GPU environment required quite a bit of creativity. However, after the appropriate technique was discovered, implementation was actually fairly straight forward. Furthermore, since the implementation method circumvents compiling lambda expressions to the GPU altogether, runtime performance was a complete non-issue. The biggest downside was overcome, and the project is proof that functional programming is in fact ideal for shader programming, given an appropriate compilation technique is used.

5.2.2 Idris and Dependent Types

Early on in the design process, the language of implementation was switched from Haskell to Idris. The reasoning behind this was that the dependent type system would allow greater precision, and hence less implementation errors. The switch came with a cost however: Idris is a significantly less mature language than Haskell. At the time, the advantages of Idris seemed to outweight the disadvantages, but it is worth revisiting this decision in hindsight.

The dependent types of Idris have indeed proven substantial worth. They have enabled a self-verifying implementation, through the use of extremely precise type sig-

natures. Graphics applications are notoriosly difficult to debug, and so having confidence in the compilers correctness has been extremely useful. However, the use of Idris had some downfalls too. As expected, the relative immaturity of the language meant that high quality and well documented libraries were not always readily available. Generating SPIR-V from Idris involved the writing of a custom library, whereas there are several libraries already availble in Haskell[2]. In addition, as the project grew in size so did the time it took for Idris to type check files - this rendered the interactive editor useless in more complex definitions, making the rapid type-driven-development cycle very difficult. Compile times were unpleasant, with the final result taking approximately 9 minutes to compile from scratch on a powerful machine.

With important points on both sides of the argument, it is difficult to come to a conclusion to whether the use of Idris had an overall positive impact on the project. If the project were written in a different language, perhaps more headway could've been made on project extensions. Equally, perhaps the quicker initial development would've been counteracted by solving bugs which Idris's type system prevented. One thing that can be said for certain, however, is that the choice of language has made the project particularly interesting, providing many intriguing insights.

5.3 Future Developments

The project has many avenues for future development. Development of a programming language is a large task, and Ictora only scratches the surface - below is a list just a few of the many possible extensions to the project.

More Builtins Over time GPUs have adapted to their common uses, and provide bespoke instructions to perform common tasks efficiently on the hardware (e.g. linear interpolation). SPIR-V exposes many such operations, yet Ictora only makes use of a select few. Ictora could be extended with many more of these operations, making it more versatile and production-ready.

Algebraic Data Types Currently, Ictora does not support user-defined data types. While ADTs would see less use in a shader than they would a general purpose program, they are still a worthwhile addition which makes programming more pleasant.

Polymorphism Extending Ictora with polymorphism would allow even greater code reuse. Since all polymorphic functions would be evaluted at compile time, this would have no performance implications - an absolute win. In practice, this would involve moving to a more powerful type system such as System F. While this significantly complicates type inference, it is a well-documented problem[15] and so is not unreasonable.

More Shader Types Ictora could be extended to support more types of shaders other than vertex and fragment. While more niche, geometry and tesselation shaders

[2]One such library is available here: https://hackage.haskell.org/package/spir-v

still see use in advanced graphics applications, and so supporting them makes Ictora appeal to a wider audience.

Chapter 6

Conclusion

The project was a success. A working implementation of a functional shader language has been implemented, which can be integrated into real world graphics programs. It has tackled the common disadvantages present in existing shader langauges, a key design goal. Errors which would normally not be discovered until runtime are identified at compile time instead: a much more pleasant experience for the programmer. Moreover, Ictora provides debugging facilities in the form of a stand-alone interpreter, allowing programmers to iterate and test their algorithms without the need to develop an entire application to serve as a sandbox.

Additionally, the project serves as a case study into the viability of functional programming for graphics shaders. It has identified the difficulties with this approach, but more importantly it has shown how those difficulties can be overcome. I believe that functional programming is excellent fit for writing shaders, and that the successful implementation of this project is proof.

Ictora is also a demonstration that formally verifying programming language implementations using dependent types is a technique which extends beyond contrived examples, and is an effective technique for fully-fledged languages too. Furthermore, some of the original ideas used to implement Ictora may be applicable to others embarking on similar projects. Techniques such as the *well-typed errors* discussed in section 4.2.3 would be useful for any formal implementation of a typed programming language.

To conclude, the project has achieved the original goals set out and more. Its success has inspired me to continue studying in this area, and I hope it also does for others.

Bibliography

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, 10(14), 2003.

[2] Guillaume Allais. Agdarsec total parser combinators. *Journées Francophones des Langages Applicatifs 2018*, page 45, 2017.

[3] Thorsten Altenkirch. A formalization of the strong normalization proof for system f in lego. In *International Conference on Typed Lambda Calculi and Applications*, pages 13–28. Springer, 1993.

[4] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *International Workshop on Computer Science Logic*, pages 453–468. Springer, 1999.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[6] Stefan Berghofer and Christian Urban. A head-to-head comparison of de bruijn indices and names. *Electronic Notes in Theoretical Computer Science*, 174(5):53–67, 2007.

[7] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[8] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[9] Edwin Brady. *Type-driven development with Idris*. Manning Publications Company, 2017.

[10] David Raymond Christiansen. Bidirectional typing rules: A tutorial, 2013.

[11] Paolo Coppola, Ugo Dal Lago, and Simona Ronchi Della Rocca. Elementary affine logic and the call-by-value lambda calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 131–145. Springer, 2005.

[12] Thierry Coquand. An analysis of girard's paradox. 1986.

[13] Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 285–296, 2010.

[14] Kevin Donnelly and Hongwei Xi. A formalization of strong normalization for simply-typed lambda-calculus and system f. *Electronic Notes in Theoretical Computer Science*, 174(5):109–125, 2007.

[15] Joshua Dunfield and Neelakantan R Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. *ACM SIGPLAN Notices*, 48(9):429–442, 2013.

[16] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In *International Summer School on Applied Semantics*, pages 137–192. Springer, 2000.

[17] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[18] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

[19] Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions draft. In *International Joint Conference on Theory and Practice of Software Development*, pages 241–256. Springer, 1989.

[20] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1019–1028, 2010.

[21] Lukas Hermanns. Cross-compiling shading languages. Master's thesis, Technische Universität Darmstadt, 2017.

[22] Graham Hutton and Erik Meijer. Monadic parser combinators. 1996.

[23] Mark P Jones. Typing haskell in haskell. In *Haskell workshop*, volume 7, 1999.

[24] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, pages 3–10, 2010.

[25] András Kovács. A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus. Master's thesis, Eötvös Loránd University, 2017. `https://github.com/AndrasKovacs/stlc-nbe/blob/separate-PSh/thesis.pdf`.

[26] Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. *ACM SIGPLAN Notices*, 42(6):425–434, 2007.

[27] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010.

[28] Ricardo Marroquim and André Maximo. Introduction to gpu programming with glsl. In *2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, pages 3–16. IEEE, 2009.

[29] Conor McBride. Epigram: Practical programming with dependent types. In *International School on Advanced Functional Programming*, pages 130–170. Springer, 2004.

[30] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 9, 2010.

[31] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[32] Craig Peeper and Jason L Mitchell. Introduction to the directx® 9 high level shading language. *ShaderX2: Introduction and Tutorials with DirectX*, 9, 2003.

[33] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *ACM SIGPLAN Notices*, 41(9):50–61, 2006.

[34] Simon L Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[35] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[36] John C Reynolds. The meaning of types from intrinsic to extrinsic semantics. *BRICS Report Series*, 7(32), 2000.

[37] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 284–298, 2020.

[38] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[39] PD van der Walt. Reflection in agda. Master's thesis, 2012.

[40] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. 2019. Available at `http://plfa.inf.ed.ac.uk/`.

[41] Beta Ziliani. Strong normalization for simply typed lambda calculus. 2012.