Visualising Haskell

Matthew Hammond
2191335
M.Eng. Computer Science

Supervised by:
Vincent Rahli
University of Birmingham
School of Computer Science

Word Count:∼ 8766

Abstract

This report details the development of an online parser, evaluator, and visualiser for a subset of the Haskell language to help users learn concepts in functional programming that are often foreign to those coming from an imperative background. The visualiser allows a user to program their own functions, data types, and type classes, and then visualise them step by step until fully evaluated. The online tool ensures the user's program is correct by performing type checking and type inference on the given program, both before and during evaluation. We evaluate the visualiser's success against how well it aided a small group of students, and provide further insights on how it could be improved.

Source Code

The source code, and how to run this project can be found at:
https://git.cs.bham.ac.uk/projects-2023-24/mxh162

Acknowledgements

This project would not have been possible without the help of many people. Firstly, my supervisor, Vincent Rahli, who has helped guide me through the project, even when I insisted on making my own parser. Also, to my family, who have supported me throughout University, to the communities, CSS and DOÜVK, and the countless friends I've made, at the University of Birmingham, who without I would've burnt out long, long ago. Thank you everyone.

Contents

1 Introduction

Functional programming is an increasingly popular paradigm, from its inception in 1950 with Lisp, it was identified as an important paradigm for creating robust, testable, and scalable code [Hug89]. The 2015 review of Hughes' article by Hu et al. [HHW15] highlighted the explosion of functional features, especially higher order functions, in modern languages such as C#, C++, and Java, as well as how further research has led to improved performance of lazy evaluation. Other features, such as lazy evaluation (seen in Python's generator functions), have been slower to uptake despite potential benefits.

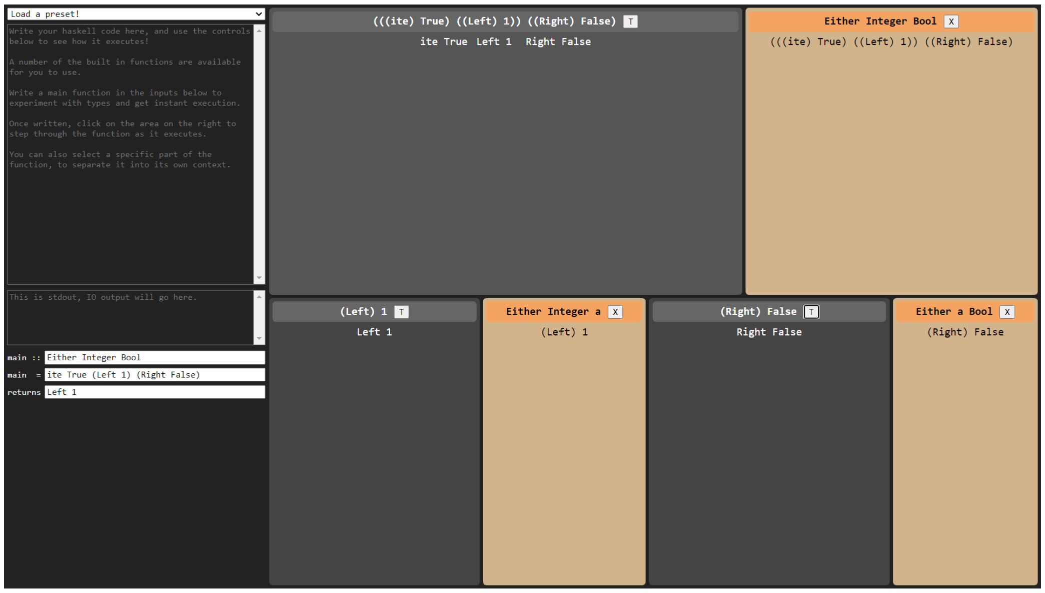
When looking at what functional programmers practice, it's no wonder why it's grown as a paradigm so much. The focus on pure functions (those without side effects, depending entirely on their inputs), and immutable data allows a programmer to write testable, reliable code, and may be the key to efficient parallel processing[Swa08]. A focus on these practices can lead to more maintainable code, with guarantees that changes to how a function works (of course, assuming the changes are correct) won't affect other functions at all, greatly reducing the complexity of a code base.

It is no surprise that this has also led to a rise in functional programming within both online and university courses [UoW23; UoK23; UoB23; Cou], but the barrier to entry is still high. Singer and Archibald's 2018 paper [SA18], which analysed 161,000 interactions from over 3,000 Haskell learners, found around 3% of lines contained syntax errors, and 18% of syntactically correct expressions contained type errors.

1.1 Haskell Visualiser

The aim of this project has been a Haskell Visualiser, example are shown in Figure 1. Being able to see the expression, subsets of the expression, and their types makes it easier to understand. This enables the user to know why their expression is or is not well-typed, and changes to the expression are quickly realised for fast debugging. A number of preset examples exist, to show the syntax, what features of the Haskell language are implemented, and how they work.

This has been implemented with a custom programming language covering a subset of the Haskell syntax, who's grammar is described in Section 5.1. This language supports polymorphic functions, custom data types, and



```
Write your haskell code here, and use the controls
below to see how it executes!

A number of the built in functions are available
for you to use.

Write a main function in the inputs below to
experiment with types and get instant execution.

Once written, click on the area on the right to
step through the function as it executes.

You can also select a specific part of the
function, to separate it into its own context.
```

```
This is stdout, IO output will go here.
```

```
main :: Either Integer Bool
main  = ite True (Left 1) (Right False)
returns Left 1
```

```
(((ite) True) ((Left) 1)) ((Right) False)
ite True  Left 1  Right False

Either Integer Bool
(((ite) True) ((Left) 1)) ((Right) False)

(Left) 1
Left 1

Either Integer a
(Left) 1

(Right) False
Right False

Either a Bool
(Right) False
```

Figure 1: The Haskell Visualiser, showing the term "ite True (Left 1) (Right 'a')", and some type information.

function overloading through type classes, and is capable of inferring the type of expressions as they evaluate. As the aim was to lower the barrier to entry, a subset of the standard library has also been implemented for use in user's functions.

1.2 Report Structure

Section 2 provides the required background knowledge. Section 3 contains the specification, detailing what the project requires to be a success. Section 4 goes over the what and why of the design decisions, and Section 5 details how that design were implemented. Section 6 covers the testing, and Section 7 describes the outcome of the project. Lastly, Section 8 evaluates the overall outcome of the project.

2 Background

2.1 Haskell & Functional Programming

Functional programming differs from imperative programming in many ways. The building blocks of functional languages are their namesake, functions, and in their basis, the lambda calculus, there are no values except functions. Functional languages extend the lambda calculus with types, functions names, and more, but at their core can be represented by plain untyped lambda functions.

2.1.1 Lazy Evaluation

Many common languages are “strict”, using call-by-value evaluation strategy. When a value is parsed into a function, it is first evaluated, regardless of it is actually used or not. This comes with the benefits of reduced complexity, and more intuitive control flow. On the other hand, Haskell is a “lazy” language, using call-by-need evaluation strategy, only evaluating expressions when necessary. Computation is stored within “thunks”, which know how to compute their value, but are only evaluated when their value is needed. This freedom allows computation, that could otherwise never terminate, to be ignored or only partially computed, such as in Figure 3. In some cases, this can improve performance and memory usage, especially in the cases of non-terminating thunks or infinite lists, in practice this can vary, as there is a not insignificant overhead to store closures and computations [Jon92; Joh84]. However this complicates side effects such as IO. In code such as Figure 2 it's possible for neither, only one, or both print statements to execute depending on the rest of the code and what elements are evaluated. If the values are required, and evaluated, they will print. Otherwise, they will remain as unevaluated thunks.

Call-by-need evaluation strategy differs slightly from call-by-name evaluation strategy, as each argument is memoized to reduce duplicate computation

```
x = [print("First"), print("Second")]
```

Figure 2: A strict language would print ’First’ and ’Second’ whereas a lazy language would not.

```python
def foo(a):
    return 1
x = foo(loopforever())
```

Figure 3: A strict language would hang, whereas a lazy language would return a thunk that can evaluate to the value 1.

```python
def foo(x):
    return x * x
foo(input())
```

Figure 4: A call-by-name language would require two user inputs, whereas a call-by-need language would require one user input, and use it twice.

and side effects. It combines the benefits of call-by-value evaluation, the removal of duplicate code execution, and call-by-name evaluation, the removal of unnecessary code execution. Call-by-need evaluation also fixes cases where side effects are duplicated, such as in Figure 4

2.1.2 Type Classes & Monadic IO

Haskell does not allow for function overloading, where a function can have different type signatures, and different uses depending on the arguments passed in, such as in Figures 5 and 6. To conform with the strict type system, the type of the overloaded method is defined within the type class, and instances must adhere to it to compile.

Haskell handles IO by using a variety of concepts from category theory, functors and monads. By restricting IO, and it's side effects, to within a particular data type that implements these classes. This doesn't solve the

```java
static int foo(int x, int y) {
    return 2 * x + y;
}
static float foo(float x, float y) {
    return x + 2 * y;
}
```

Figure 5: An int passed into "foo" will have different behaviour to a float.

```haskell
class Fooable a where
    foo :: a -> a -> a
instance Fooable Integer where
    foo x y = 2 * x + y
instance Fooable Float where
    foo x y = x + 2 * y
```

Figure 6: Figure 5 rewritten using Haskell's type class system.

problem of indeterminate ordering, but does force the programmer to consider IO whenever handling it.

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
```

Listing 1: The Functor and Monad typeclass.[1].

"fmap" allows functions that were originally not designed for wrapped types, to work as intended, and the ">>=" and ">>" function allows for cleanly passing wrapped types around and handling error values. By using these operators side effects that affect the system the program runs on, including potential errors, are contained and handled appropriately. A variety of other functions exist with the standard library to help use these classes easily.

These classes are also used elsewhere, especially for simple error handling. For example, in Listing 2, if the first call to "head" returns Nothing, it is propagated onwards. Only if both the calls to "head" return values does it return a tuple of the values.

```haskell
head :: [a] -> Maybe a
head [] = Nothing
head (x:_) = return x

dualHead xs =
    head xs >>=
    \x -> head (tail xs) >>=
```

[1]GHC's dialect has the Monad class depend on Applicative, which depends on Functor. This differs from the Haskell 2010 Language Report which this project is based off.

```
    \y -> return (x,y)
```

Listing 2: Safely fetch the first two elements of a list.

Numerous other built-in functions use the Maybe and Either types for malformed arguments, such as for safe division, which then must be handled by the caller through pattern matching or the class methods. This allows many errors, such as division by zero which would throw an error in most other languages, to be handled safely before the program ever runs. The programmer is unable to simply overlook the error and cause a program crash during runtime.

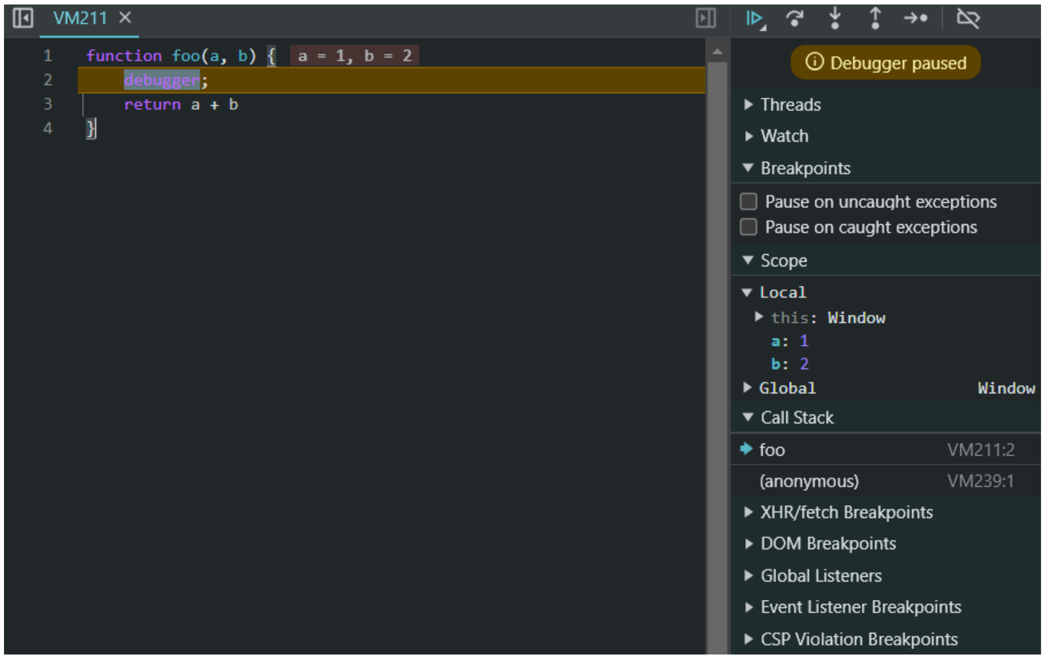
2.2 Visualisation

Various visualisers exist online for visualising algorithms, data structures, as well as general programming languages. Countless YouTube videos exist breaking down algorithms and data structures step by step to explain how they work, and sites like [Algo] have a wide array of algorithms and data structures that can be viewed.

More focused, visualisers can also show general programming languages. Especially for imperative languages, the current memory can be tracked [CSC], showing what values exist in memory, and what variables represent what values. This is especially common in debuggers, such as the one's built into modern web browsers to aid with JavaScript development, such as the one shown in Figure 7. These types of visualisers can greatly aid in development, as the exact line of code where the problem occurs can be quickly detected.

Visualisers come in many different styles, some such as Google Chrome's debugger are largely plain and text-based, aiming to be user friendly without knowledge of how the language works behind the scenes. The extreme end of this are visualisers like [CBN], which, at every step of evaluation, shows a valid Haskell expression. This visualiser is especially adept at visualising lazy evaluation, as each step transforms into a new valid with only the relevant changes, allowing the user to see which thunk has been evaluated.

Other visualisers opt for a graph based approach, such as the [SPARTAN] visualiser by Waugh Ambridge. These types of visualisers are often much lower level, in this case showing individual applications, bindings, as well as showing when and which thunks are being evaluated. These types of visualisers require more in depth knowledge of the basis of functional pro-



```
function foo(a, b) {  a = 1, b = 2
    debugger;
    return a + b
}
```

Figure 7: Google Chrome's debugger, showing the current state in code, including bound variables, including implicit variables, such as "this".

gramming, lambda calculus, as they remove most of the abstraction and simplicity gained from the programming language syntax.

2.3 User Study

2.3.1 Motivation

To gauge how useful such a visualiser would be, an online survey was designed to work out how effective the current methods of teaching functional programming had been. Singer and Archibald [SA18] had shown that there were significant challenges to learning functional programming, with frequent errors preventing code execution. The survey would help validate the need for more tools and visualisers within functional programming. It would also be useful to know which methods were commonly used to help debug programs, and to analyse them to work out how they help, and how they could be improved.

The survey was aimed towards students either currently in their 2^{nd} year (when the Functional Programming module is taught at the University of Birmingham), and those in later years who had already taken the module. 14 students responded, 3 in their 2^{nd} year and 11 in later years.

2.3.2 Results

Not all the questions in Appendix B were required, but enough responses were recorded to show the need of a tool to help learn and debug Haskell. All the answers are also recorded in Appendix B.

Which tools/visualisers have you used for testing and debugging Haskell?
If users were already familiar with an existing suite of tools to help learn Haskell, the usefulness of a new one would be greatly diminished. Of the responses, 10 participants said they hadn't used any tools for helping with Haskell. 3 participants used GHCi, an interactive shell built by the GHC contributors. It is capable of showing the principle type of expressions, and is the easiest way to run functions as scripts. 1 participant reported using the Haskell extension available within the VSCode marketplace. This extension is widely used, with over half a million downloads, and comes with a large number of functions including type checking expressions as the user implements them, allowing errors to be detected without having to attempt compilation.

Which tools/visualisers have you used for testing and debugging other languages?
In contrary to Haskell, tools among other languages are much more common. 5 participants reported using a variety of VSCode extensions for languages, which includes linters, debuggers, and type checkers. A number of other debuggers, GDB for C, PyCharm, mypy, ruff, and more for Python, and a variety of Java tools were mentioned by them and other respondents. Only 3 participants didn't report using any tools to test and debug, a sharp drop from Haskell. It's clear that Haskell has a much smaller array of tools available when compared to other languages, and those that do exist are used much less frequently and are often less comprehensive than other tools.

How did you find the transition from object-oriented imperative languages to a functional language? (1-5, hard-easy)

The average response to this question was 3.4 out of 5, indicating that the transition wasn't really easy, but also wasn't really hard. As shown below, I think this is because some concepts, such as higher order and anonymous functions, are already present in a number of popular languages, and are fairly well understood, whereas the more complicated concepts can typically be ignored until the user needs to implement a function using them.

How well would you say you understood the following concepts? (1-5, not well-very well) (Additional comments in Table 2)
Participants were asked about higher order functions, anonymous functions, monads, lazy evaluation, partial application, and pure functions. As expected, the concepts common to other languages, higher order and anonymous functions, on average scored higher, 4.2 and 4.3 out of 5 respectively. When leaving out participants who responded that they did not know what the concept was, the remaining concepts, in order from most to least well understood, were lazy evaluation (4.0/5), partial application (3.6/5), pure functions (3.5/5), and monads (2.8/5). When asked to elaborate on difficulties, several respondents reported they only felt like that had a surface level understanding of concepts. "*I...never fully understand WHY they worked or were a certain way.*", "*I still didn't find them [monads] intuitive.*". This could be due to users being unable to see how these concepts actually evaluate, which could be rectified with a step by step visualiser, such as what this project aimed to achieve.

How likely are you to use a functional language in the future? (Answers in Table 3)
6 respondents said they were likely, 4 said they were unlikely to use functional programming in the future, and 2 said they would use some concepts in their imperative programming. The elegance and correctness of functional programs was noted as a positive, as well as being able to emulate the notation of mathematics within programs. The lack of tooling, and familiarly with imperative object-oriented programming was noted as reasons to not continue with functional programming.

2.3.3 Analysis

Despite the small number of participants, it's clear that Haskell lacks the extensive tooling that is present in many other popular languages. Many of the responses reflect my own feelings regarding functional programming. This validates the aim of this project, and its success will be partially dependant on how users are able to use the visualiser to improve their understanding of Haskell.

3 Specification

A variety of measures will be used to determine if the project is a success. To ensure a robust system, these will be implemented incrementally, and tested for reliability and correctness, as well as usability by the end user.

3.1 Functional Requirements

1. The tool must be able to parse user input into the correct expression, and parse as they would expect a Haskell expression to.
2. The tool should report appropriate errors on malformed inputs, including both syntax and type errors, so the user can debug and modify their code.
3. The tool must be able to evaluate the expression correctly, and evaluate as they would expect a Haskell expression to.
4. The tool must be able to visualise expressions correctly, to accurately represent the expression.
5. The tool must be able to visualise the type of expressions correctly, to accurately represent the type of the expression.

3.2 Non-Functional Requirements

1. The tool should parse user inputs within a short period of time, so the user can quickly see changes they make take effect.
2. The tool should not slow down when visualising large expressions, as Haskell expressions can become very large while they evaluate.
3. The tool should be easy to understand by the user, to ensure they can learn from it.

4 Design

When analysing the problem, it was clear there were 3 distinct parts:

- Parsing user input.
- Evaluating the expression.
- Visualising the expression.

By implementing each part individually and then tying them together with a well defined interface, each part could be individually tested, and changes to one would not affect the others.

4.1 Language

The first design decision to make was which language to implement the visualiser in. Most languages are perfectly suitable for the job, with extensive libraries and tooling, but picking the one for the task early would allow me to research existing solutions to the further sections.

An obvious choice would be to use Haskell, with a variety of existing tools for parsing Haskell within itself, such as ghc-vis for visualising data structures. However, despite making a visualiser for it, my knowledge with it's built-in library and syntax was not great. I was going to learn a lot about Haskell while making the visualiser, but I would encounter many hurdles if I was to use it. While the user of such a visualiser would likely already have Haskell installed, a beginner would need to install the project.

My next choice was JavaScript, a language that I am much more familiar with. It has a much larger range of libraries and online information. JavaScript also has the advantage of being the primary language on the web, allowing extremely easy access by going to a site hosting the visualiser. To reduce complexity, I decided to keep it as a static site, with all the logic on the client-side. This meant I didn't have to produce a server and handle communication between it and the client.

4.2 Lexing and Parsing

Within JavaScript, a variety of community made parsers exist [ANTLR; OhmJS] that can take a specific grammar and generate a parser. However,

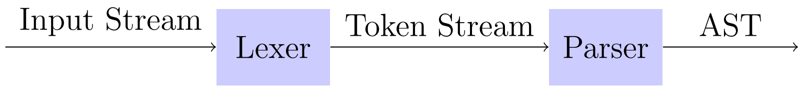


Figure 8: The flow of data within a parser.

these tools parse into a representation that would need to be further converted into an implementation that could work with the other sections of the project.

Therefore I decided I would make my own parser. The main requirements was an easy to implement, and extensible parser so features of Haskell could be implemented incrementally. The visualiser would primarily be used for small code snippets, so while efficiency was important, it was not the most crucial aspect.

To satisfy these requirements, I investigated various types of parsers. The most common distinctions being LL(k) and LR(k) parsers, or top-down and bottom-up, where "k" are the number of lookahead tokens. LL parsers are typically easy to manually create, but are much harder to run in linear time as they may have to backtrack when reaching dead-ends while parsing. LR parsers are the opposite, typically requiring specialised tools to create the tables necessary to parse within linear time with no backtracking or guesswork.[SGC07].

The choice was clear, an LL parser would fit perfectly. Care would have to be taken to keep the time complexity down, but the ease of use and extensibility would more than make up for it.

While plain text can be parsed into Haskell directly, it is much easier to pre-process the data with a lexer, converting the input stream into a stream of tokens. This removes extraneous characters, such as insignificant whitespace, and groups characters into labelled tokens. The set of possible tokens is much easier to match on than the set of characters, as the tokens mark the start and end of each construct. This means the lexer's purpose is to divide up the input, and the parser's purpose is only to validate it, separating the problem into two easier problems.

4.3 The Internal Representation

As the users input would require additional functionality than just executing, it had to be converted into an internal representation that could be analysed, stepped through, and visualised. Compilation was off the table, as that would lose important information associated with visualising, such as identifier names, and would be very difficult to step through and visualise. Type information can also be discard after successful compilation, as values of one type are never passed into a function that cannot take them, and non-exhaustive pattern matching is generally avoided.[2]

Other interpreted languages, such as Python, Java, and C#, are compiled into a bytecode and then interpreted via a virtual machine. Bytecodes are a compact and efficient way to store and run an interpreted program, as extraneous characters, such as whitespace, can be stripped, and identifiers simplified. However, in my case, converting the users input into bytecode would lose too much information like compiling, as type information can be removed and identifiers simplified. As the goal is to visualise expressions similarly to the user's input, preserving identifier names is a necessary requirement to ease understanding.

The easiest approach I could think of was an object-oriented style to represent types and thunks. This would allow some aspects of each system to be implemented and tested together, before adding more features later, allowing them to built on top of a strong foundation. By defining an interface for types and an interface for thunks, e.g a method to apply a beta reduction given the symbol and replacement, a thunk can have any type, any child thunk, e.t.c and handle its case without needing to know about its parent's, or children's information. Another benefit of constructing the types and thunks in a tree structure is that the child thunks and types are still fully formed expressions. This would allow for the user to analyse parts of expressions without errors occurring.

4.4 Visualising

As described in Section 2.2, two classes of visualisers identified were text-based, and graph-based visualisers, each with distinct advantages and disadvantages.

Graph-based visualisers are more visual, and can smoothly transition between states. This allows the user to easily see the changes from one state to another, as they can focus on that part. A number of libraries also exist for visualising graphs, such as [cytoscape.js], that are capable of rendering

[2]Unless unsafe functions are used, but they come with many other risks.

a manipulable graph within a website. In many ways, an abstract syntax tree is already a graph, and, for this project, each thunk could be converted into a node in the diagram, with edges connecting to its children. However, although easier to display as a graph, this style of visualisation not necessarily easier for the user to interpret, as the graph is visually very different from the program the user inputs.

Text-based visualisers aren't as easy to create smooth transitions with, but keep the states very readable. For example, when a term is substituted into an expression, the user will be able to see what the new expression looks like without having to decode a graph, and the user can more easily understand what has happened without understanding exactly how functional programming works. However, few libraries exist, so a way to visualise the expression would have to be implemented manually.

Despite the extra work, I decided to implement the visualiser with a text-based approach, similarly to [CBN]. This is for the aforementioned reasons, as ease of use was one of the main design goals for the visualiser.

```
function any(...fs) {
    // Attempt to lex against each choice, returning all
   matches.
    return (input) => fs.map(f => lexer(f)(input));
}
```

Figure 9: The lexer corresponding to the choice rule, returning all matches.

```
varid = tok(diff(all(small, many(any(small, large, digit, "'"
    ))), reservedid), Token.VARID)
```

Figure 10: $varid \rightarrow (small\{small|large|digit|'\})_{\langle revservedid \rangle}$

5 Implementation

5.1 Lexing & Parsing

Looking at the lexical syntax within section 10.1 of the Haskell 2010 Language Report[Mar+10] there are many reused notational constructs, shown in Table 6. Due to this, I identified using a lexer combinator as the ideal choice for lexing.

A lexer combinator is a higher-order function that accepts one or more lexer, and returns a new lexer. When this lexer is run against an input, it returns an array containing all the possible matches. This quickly allows for complex lexers to be implemented, and for additional rules to be added. I first set about creating the lexers that represent the notational constructs given, as well as the maximal munch rule, shown in 7. This rule requires the longest input to be consumed at each step. This is especially apparent when matching identifiers, as it solves the problem of the identifier “letter” being lexed as the keyword “let” and the identifier “ter”. With these building blocks, all the other lexers could be quickly constructed, such as the lexer responsible for marking variable identifiers (used for type variables, function identifiers, and arguments). A benefit of using a lexer combinator, was that each parser could be individually tested before constructing the next one. For example, the “varid” parser depends on a number of smaller parsers. Once I knew the smaller parsers were working correctly, I could confidently rule them out as the causes for issues. This allowed for rapid prototyping, and lexing hence took a short time to implement.

Once the input had been lexed, it could then be converted into the internal representation, described in section 5.2. This would also use an LL parser but, due to the complexity of converting from tokens to thunks, was not implemented as a parser combinator. A top down parser was still used for simplicity.

The further goals of this project, data types and type classes, both rely on specific keywords (data, and class and instance respectively). In fact no keywords were initially required, as other keyword expressions (case-of, if-then-else, foreign, e.t.c) were not planned to be supported. This allowed the basics to be implemented, before more complicated structures were parsed.

As the tokens were in an ordered list, they could be shifted off of a queue, inspected, and then a decision made. If an invalid token was received, an error was thrown, which describes what token it encountered, what token it expected, and where in the user input it happened. This allows the user to modify their code quickly, and handle the errors they encounter one at a time.

Parsing of each construct was implemented inside a dedicated function. Some functions iterate other the tokens until a specific end token is found, such as a ⇒, or a close parenthesis, and others simply parse until the end of the line (new lines are the only whitespace preserved by the lexer). By using separate functions, each construct could be parsed and tested individually, and can use each other when required.

As they are parsed, functions, data types, and type classes are stored under the ”main” module. As the input is parsed from the top down, constructs must be defined before they are used. For example, when an identifier is seen within a function, if it exists as an already parsed function, it is recorded as a call to that function. Otherwise, it is recorded as an unbound identifier.

The full grammar of the language is described in Figure 25.

5.2 Internals

Before implementing the parser, I had to create the internal representation that the user input would parse into.

5.2.1 Types

Haskell does not have a dependant type system, which means that while thunks would depend on types, types would not depend on thunks. This

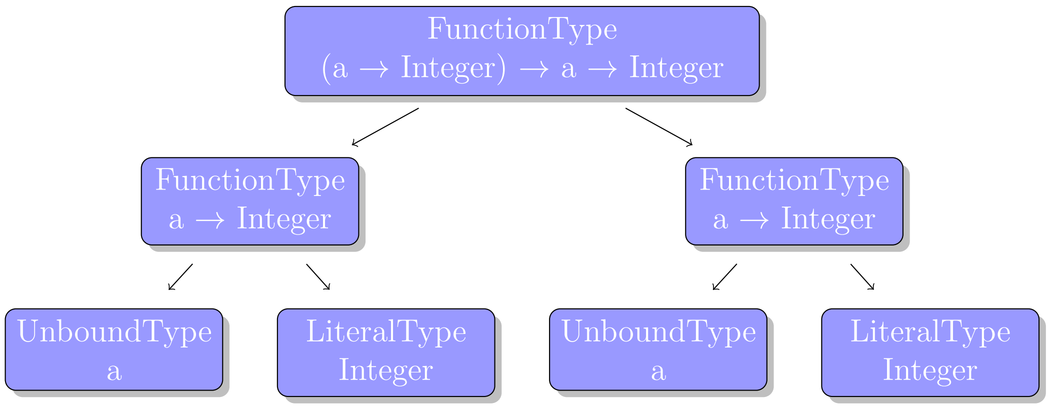


Figure 11: The internal representation of: (a → Integer) → a → Integer

meant I could implement and test them first before moving on to thunks. As planned, types were constructed as a tree, allowing types to be built using other types, as in Figure 11. Initially, only literal, unbound, and function types were implemented, with no notion of typeclasses, to create the first prototype. These were enough to create basic functions.

However, before they could work, a method of type inference was required. For example, if a `String` → `Integer` function is bound to the type in Figure 11, it is inferred that `a` is the type `String`, and the type `String` → `Integer` would be returned. This is performed by first collecting all of the type constraints, before unifying them using a variation of the algorithm presented in [MM82]. This reduces the constraints until either a solved set is created, or an error is thrown, indicating a badly typed expression.

The simplest matching was for LiteralTypes, which had to be identical (so an `Integer` could match an `Integer`, but not a `Bool`), and generated no constraints. FunctionTypes simply had to match their left and right children to the left and right child of the incoming type (which also had to be a FunctionType). It returned the union of the constraints of its children. UnboundTypes were by far the most complicated. Given two types with disjoint sets of unbound symbols, they could simply match together, with the constraint setting them equal. However, if the symbol sets were not disjoint, matching has additional constraints. This is solved by introducing a strict set, where matching can only occur against identical symbols within the set, but any symbol outside of the set. In Figure 12, “f” has the strict set “{a, b}”, so when an “a” tries to match a “b”, it fails. When the second

```
foo :: (a -> b) -> b -> b
foo f b = f b
```

Figure 12: “foo” actually has the type “$\forall a.\forall b.(a \to b) \to b \to b$”, which fails as ‘b’ can only be bound to “a → b” when “a” and “b” are equal, not for all “a” and “b”.

argument’s type is changed to an “a”, it is able to successfully match an “a” to an “a” and succeeds. UnboundTypes also have one other opportunity to match. Whenever the incoming type is an UnboundType, matching is also performed in the other direction. This is required as constraining a variable within the incoming type may constrain other variables in both the incoming and receiving type.

Data types were relatively simple to add. Just like LiteralTypes (such as `Integer` and `Bool`[3], the names of the type must match exactly. You can’t substitute a `Maybe` for an `Either`, just like you can’t substitute a `String` for an `Integer`. The similarities meant that instead of creating a new class for these, LiteralType worked perfectly with no changes. Types which had arguments relied on a new class, ApplicationType, which had a left and right child. They matched identically to FunctionTypes, as their left and right children had to match, and the generated constraints were unioned.

The last feature to implement for types were type classes. Additional constraints to unbound types about what types could replace them. Due to the modular implementation, this only required additional logic to be added to the UnboundType class for checking if a binding was valid. Incoming types had to be at least as restrictive as the receiving type, e.g “Real c =¿ c” is more restrictive than “Num a =¿ a”, as the “Real” type class depends on the “Num” type class, whereas “b” (with no constraints) is less restrictive and cannot match. When a LiteralType is bound to an UnboundType, it now required the LiteralType to have implemented whichever constraints the UnboundType held.

[3]`Bool` is actually implemented as a data type, but as it has no variables behaves identically to a LiteralType.

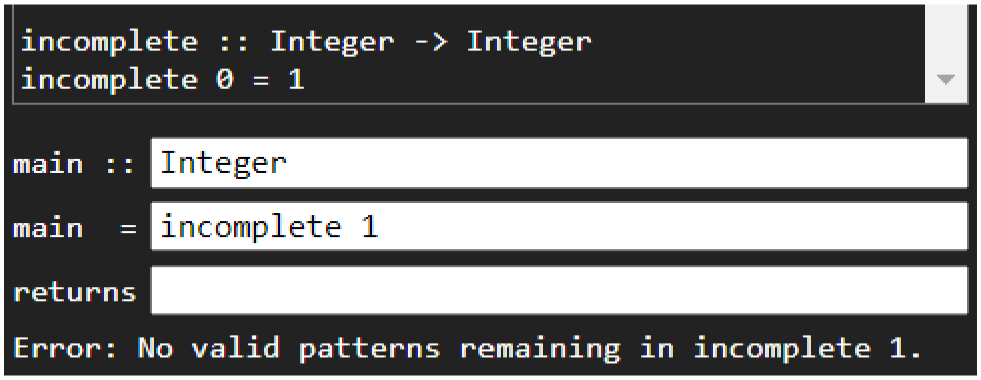
5.2.2 Thunks

Like types, thunks were implemented incrementally with a specific interface. To begin with, only 4 thunks were needed: literal, unbound, function, and application. These directly correspond to the types of the same name, with the ApplicationThunk corresponding to applying a type to a function type. Also like types, each child thunk of a thunk was to be a valid thunk by itself, allowing them to be passed around and substitued where required.

A LiteralThunk represents an actual value, such as the number 1, the float 1.0, or the character 'a', and, as its type is known at compile time, is annotated with its type. By annotating as many thunks as possible as early as possible, type inference and type checking takes less time, and provides stricter results. An UnboundThunk represents an unknown value, typically found in the arguments. It only needs to know which symbol it is represented by. It is initialised with an AllType, as it could have any value on its own. An ApplicationThunk represents an application of one thunk to another. It takes its two child thunks as arguments. When this thunk is asked for its type, it uses a JavaScript getter method to dynamically infer its type from its children. This also ensures that its left child is always a type of function. The ApplicationThunk is the first thunk that can "step", evaluating one step of computation. If the left child is an ApplicationThunk, or otherwise can step, then it must step before the right child can be applied. However the right child does not always need to be stepped to be applied, because of the call-by-need evaluation strategy.

A FunctionThunk represents an individual function. When creating a function, initially it just needs a name and a type. This allows it to be initialised, and then used in function definitions for recursion. A function with no cases is equivalent to $\perp$, and will throw an error if it is attempted to be called. Cases are added by defining a pattern and an implementation. The implementation is simply a thunk, but the pattern is a specific class, essentially a wrapper around a list of arguments.

When an ApplicationThunk applies a thunk to a function, it first evaluates the resultant type. During evaluation this is guaranteed to succeed, as the types of functions and expressions are checked after parsing. Then each pattern is checked for a match, which is described in Section 5.2.3. If the match fails, then the pattern and relevant implementation are discarded, as there's no way for the function to evaluate that branch. If the match succeeds the pattern discards the matched argument, and either sets a new case for



```
incomplete :: Integer -> Integer
incomplete 0 = 1

main :: Integer
main  = incomplete 1
returns
Error: No valid patterns remaining in incomplete 1.
```

Figure 13: Incomplete functions will error if no cases match.

the resultant FunctionThunk, or (if the pattern matched completely) returns the relevant implementation. If, after matching, no patterns are left in the resultant FunctionThunk, then it means all matches failed (and the pattern matching was incomplete), which is equivalent to ⊥ and a helpful error is thrown, shown in Figure 13. Like in Haskell, a partially matched function is a function of its own, and can be parsed around like any other thunk.

Not all computation is easy to define using Haskell alone. Int addition could be defined with lines upon lines of pattern matching, but this is impractical in the best cases, and impossible (e.g for types with infinite constructors such as Integer) in the worst cases. In these cases Haskell defers to C but, to keep it available to use on the web, we will defer to JavaScript. These functions, called JSThunks, are restricted to using a subset of types. JavaScript has no notion of the type "Either a b", IO types, and many more, but it does contain numbers, characters, strings, and arrays, which have corresponding types in Haskell. To implement these, they require a name, a type, and a JavaScript function. Again, as Haskell allows all functions to be curried, and can be partially applied. This is solved for JSThunks by passing in a function that returns a function.

```
((a, b) => a+b)(1, 2)
(a => b => a+b)(1)(2)
```

Listing 3: A curried addition function in JavaScript.

Whenever a JSThunk's function returned a new function, it was wrapped in a new JSThunk with the appropriate name and type. If the value being returned was to be a Literal, it was wrapped in a LiteralThunk, and given the

```
fmap :: (a -> b) -> m a -> m b
fmap f ? = ?
```

Figure 14: fmap

type inferred from the function and argument's types, allowing basic polymorphism. Lastly, (once data types had been implemented), specific data types were supported, such as converting JavaScript's primitive true and false, to Haskell's True and False constructors, as well as converting strings to and from character lists to JavaScript strings. To simplify JSThunks, they do not perform any pattern matching. This could be emulated using if-else statements in their JavaScript functions, as could any arbitrary function. This was specifically avoided to keep computation within the interpreted Haskell environment as much as possible, to ensure as much could be visualised as possible. This is because JavaScript execution, such as adding or multiplying 2 numbers isn't visualised step by step. If, for example, the nth Fibonacci number was implemented as a JSThunk, instead of a regular FunctionThunk, the user would not be able to see each step be visualised. It would jump from the function call to the final answer, instead of showing the recursive calculation.

With data types, ConstructorThunks were introduced. These behave similarly to LiteralThunks, but their type is defined within the data type declaration. Essentially they are a LiteralThunk which has a FunctionType.

With type classes, their methods can behave completely differently depending on which instance method is required. Which method is required is determined by the type of its arguments and return value. To fix this, FunctionReferenceThunk was introduced. When a class method is parsed, the number of arguments that must be applied to it to determine which instance to use. When the FunctionReferenceThunk has enough arguments applied, it reduces to the specific form of the method. For example, the fmap function in Figure 14, which is dependant on the “m” type variable to decide which “Moand” instance is used. When the List Monad is used it behaves like the standard “map” function, but when the Maybe Monad is used it propagates “Nothing”, and only applies the function within the “Just” case.

5.2.3 Arguments

As described above, FunctionThunks use Patterns, which wrap a list of Arguments. The initial arguments, either literals or unbound, would match either literals, or anything respectively. When matching a LiteralThunk to a LiteralArgument, their types and values must be the same, or else they fail. If the thunk passed to a LiteralArgument is not fully evaluated, it must be. This is because an unevaluated thunk, despite having a fixed type inferred from its children, does not have a fixed value. As in Figure 15, no assumption can be made about what "$(1 + 1)$" might return, other than it will return an "Integer". When matching an UnboundArgument, only the type was required to match, and the symbol used for the argument is replaced within the corresponding implementation. This type of argument doesn't require a thunk to be evaluated, allowing for the call-by-need evaluation strategy. In the second case of Figure 15, as "a" is not required to be a specific value, "$(1 + 1)$" does not need to be evaluated. A special argument, the WildcardArgument, written in Haskell as an underscore is the same as an UnboundArgument, except no replacements occur when a match succeeds. There can also be multiple wildcards within a single pattern, unlike the symbols of unbound arguments.

```haskell
foo :: Integer -> Integer
foo 0 = 1
foo a = a
foo _ = 0

main = foo (1+1)
```

Figure 15: $(1+1)$ must be evaluated before the pattern can test each match.

To go through step by step, in Figure 15, "$(1 + 1)$" is compared to "0". As the argument is an unevaluated thunk, instead of continuing to match against the other cases, the argument is evaluated, leaving the expression as "foo 2". If matching was allowed to continue, a unevaluated thunk that represented the value "0" could erroneously match later cases. In the next step, "2" is first compared to "0". While their types match, their values do not, so the "foo $0 = 1$" case is discarded. Then "2" is compared to "a". As "a" is unbound, it accepts all arguments, and the match returns the constraint "$a = 2$". This constraint is applied to the implementation,

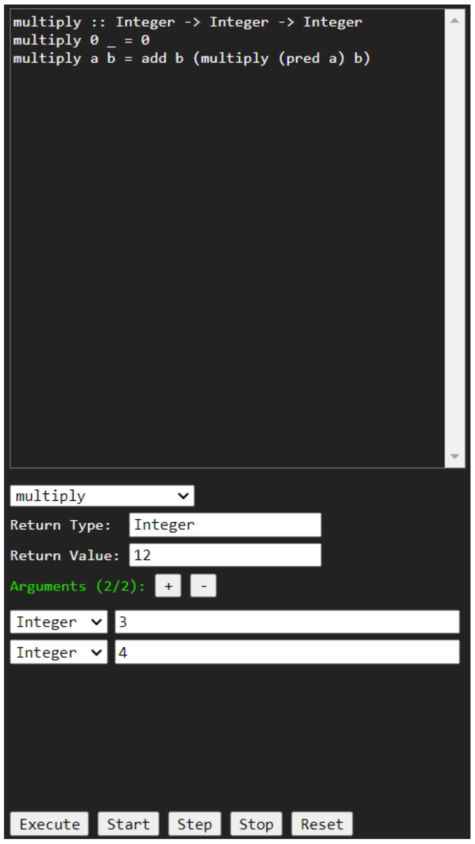
```haskell
data Foo a b c = Unit
               | Fixed Integer
               | Var a b c
               | Reverse c b a
               | Omit a c

example :: Foo Integer String Bool -> Foo a b c
example (Var u v w)      = Unit
example (Reverse x y z) = Fixed 1
```

Figure 16: While all of "Foo"'s constructors return a value of "Foo a b c", the order of their arguments (if any) can differ from the order in the declaration.

setting the UnboundThunk "a", to the LiteralThunk "2". As "a" was the final argument in "foo", the case has been fully matched, and so the second case is returned. While "(1+1)" could also match on the third case, as it is a wildcard, only the first successful match is returned. This allows some cases to be more restrictive (such as the "0" case as shown) and take precedence over less restrictive cases.

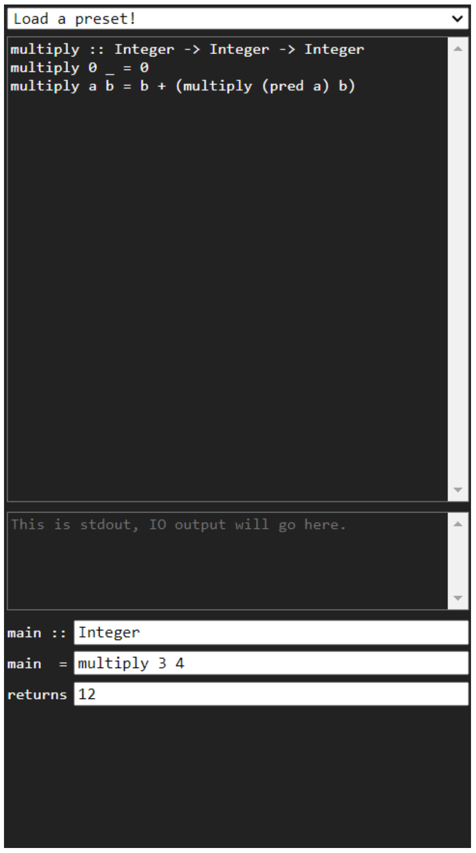
Introducing data types to arguments came with some challenges. While the matching of ConstructorArguments and ApplicationArguments was simple, with ConstructorArguments having to match exactly, and ApplicationArguments having to match their children. Annotating the types of arguments was initially challenging. As a data type's constructors can be of many different forms inside the same data type. The type that the constructor returns must first be fetched, and then matched against the expected type of the argument, and potentially restricting some types. In Figure 16, while the first case's symbol match the order of "Foo", the second cases does not. Assuming the constructor returns type checks so returns the correct type, the type and order of arguments must be deduced through the constructor.

When implemented, typeclasses slotted in seamlessly, as they were implemented completely separately of arguments and thunks. When matching types, because the methods defined in the Type interface were used, once changes to those methods were successfully implemented, users of those methods were able to work with typeclasses with no additional changes.



```
multiply :: Integer -> Integer -> Integer
multiply 0 _ = 0
multiply a b = add b (multiply (pred a) b)
```

Figure 17: The first iteration of the UI.



```
multiply :: Integer -> Integer -> Integer
multiply 0 _ = 0
multiply a b = b + (multiply (pred a) b)
```

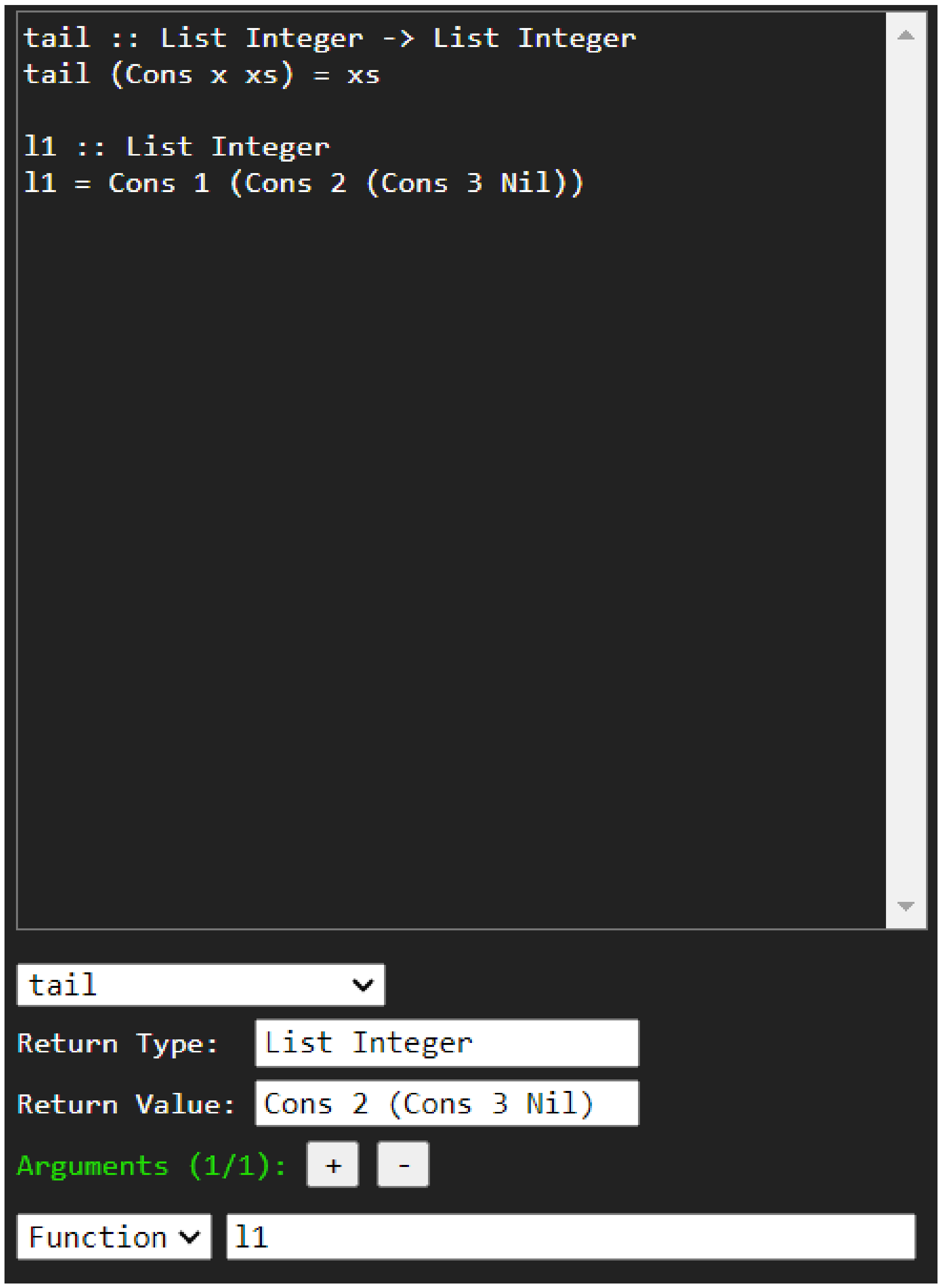
Figure 18: The second iteration of the UI.

5.3 Visualising

5.3.1 UI & User Input

The UI, and how the user input a program underwent through two major iterations during development.

Initially, as shown in Figure 17, the user would input a program within the topmost box. Using the input boxes below, the user could then select a function to evaluate and determine the number of arguments to pass into it (using the + and - buttons). The text would turn green when enough arguments are being passed into the chosen function. To choose arguments, the type could be selected from a drop down list, either "Integer", "Float",



```
tail :: List Integer -> List Integer
tail (Cons x xs) = xs

l1 :: List Integer
l1 = Cons 1 (Cons 2 (Cons 3 Nil))
```

tail

Return Type: List Integer

Return Value: Cons 2 (Cons 3 Nil)

Arguments (1/1): + -

Function l1

Figure 19: An inelegant fix to bad UI.

"String", or "Function". "Function" allowed the user to set the argument as a function they had defined in the code input box. The values, such as a number, a string, or the name of a function, could be input in the box to the right of the type. The buttons at the bottom controlled the evaluation of the expression. "Evaluate" would fully evaluate the expression, putting the return type and value into the relevant boxes once finished. "Start" and "Stop" would enable and disable stepping through the expression five times a second. "Step" would step once through the evaluation and, if the expression had finished evaluating, fill the return type and value boxes. "Reset" would reset the expression back to the beginning, so the user could step through it again.

While functional, this iteration had numerous drawbacks. The main two were how slow it was to setup evaluation, and how data types could not be easily integrated. Any function could be run, but defining the arguments one by one, and manually setting their type was both tedious and error prone. The user could input a string, but set its type as an "Integer" which could cause unintended issues[4]. As this version of the UI was made before data types had been implemented, when they were added an elegant solution for the user to user data types in their arguments was not possible. The easiest way was for the user to define a new function, which took no arguments, and returned the specific data type they wanted, such as in Figure 19.

The second iteration of the UI, shown in 18, was the result of fixing the problems of the first iteration, new features, and especially user feedback. The main change was switching from setting individual arguments, to a "main" function. This was implemented by simply parsing the expression within the main function input box. This could automatically work out the types of the arguments, as well as parse more complicated expressions such as those used to make data types. This box was strictly limited to parsing function implementations (as it was the implementation of "main", and can even be used recursively), so when the parser was expanded, the capabilities of "main" also expanded. Haskell usually restricts this function to the "IO ()" type, but, as it was the only entry point, it was relaxed to allow any return type. This UI also has the additional feature of live evaluation. Whenever the expression does not depend on user input, the expression will be evaluated to attempt to find the return type and value. If these are found

[4]Though because of JavaScript's automatic type conversion usually didn't cause any JavaScript errors.

within a couple seconds, they are displayed to the user. These values help the user quickly debug programs, as if the return type or value doesn't match what they thought it would, they can edit their program, and see the new changes live.

When monadic IO was implemented, a new read-only box was added for IO output to display. IO input was naturally implemented through JavaScript's modals, as they can easily request a line of input from the user when run.

5.3.2 Expressions

Visualising expressions was the core method for the user to learn and understand functional concepts. Figure 1 shows an expression mid evaluation, as well as showing some type information and sub-expressions.

Instead of multiple control buttons, as in the first iteration of the UI in Figure 17, the right hand side can instead be clicked on to evaluate a single step. The user can click on the "T" button to open up the type information box, and close it with the "X" button. This box shows the type information of the corresponding expression. This is useful as every sub-expression is also a valid expression with it's own type, so the user can use this information to see why types are the way they are. For example, in Figure 1, the type of the full expression is "Either Integer Bool", which is made by combining the type of the two sub-expressions within the "ite" function, which have the "Either Integer a" and "Either a Bool" type respectively.

To open up a sub-expression, the user can click on part of an expression. In Figure 20, the "((:) 1)" expression has been clicked, opening it up below. In this case, it can't be evaluated further, but if it can, when the below box is clicked, it is evaluated separately from the main expression. This allows the user to experiment with parts of expressions, without them affecting the main expression. Once fully evaluated, it can be clicked again to be removed. Thunks which are the same object, such as "(:) 1" in both the main and sub-expressions, are highlighted the same colour when hovered, to show that they are linked. 3 layers of applications above of the hovered-over thunk are also highlighted, to show where it exists in relation to the rest of the expression.

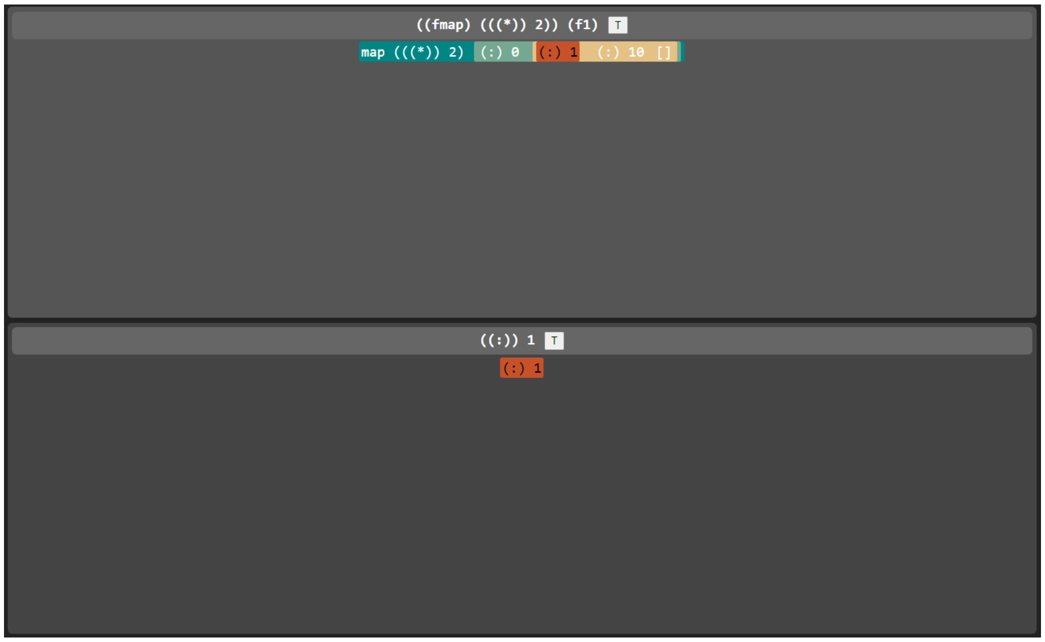


Figure 20: When hovering over an application, it, and its parents, are highlighted, including within sub-expressions.

6 Testing

Due to constructing each feature incrementally, progress on each feature could be gated by successful tests. In this case, implementing basic functionality was done, and corrected until tests passed, before implementing both data types, and then type classes. This ensured additional features didn't break the existing ones, creating a strong foundation to build from.

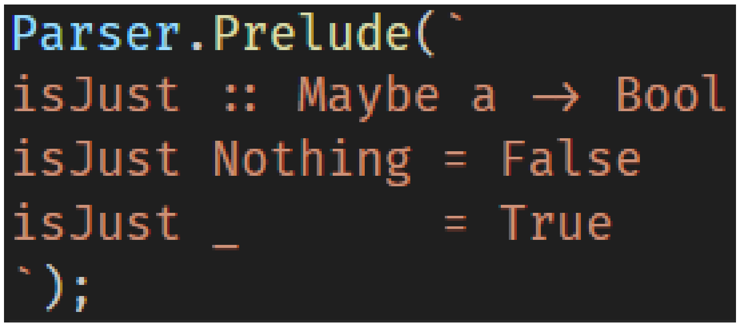
Once type classes had been introduced, a large number of Haskell's standard library was implemented to facilitate integration testing. Implementing these covered both an extensive range of test cases (as they covered all of the implemented features of Haskell), and gave a subset of the Haskell standard library for the user to use while making their own functions. They are run when the page is initially loaded, outputting positive and negative results into the console, which allowed for errors to be quickly pinpointed. These tests covered lexing, parsing, and type checking of functions, data types, and type classes, as if any failed to parse and threw an error, it could be inspected and the code fixed. In most error cases they failed due to being incorrectly defined, a problem with the function itself and not the tool, however they were important to catching a variety of edge cases, such as type class constraints within a class method seen only in the "RealFrac" class, shown in Figure 21. Initially class methods did attempt to parse class constraints, and so when "RealFrac" was added it causes a parse error. These are contained within `src/interpreter/Prelude`.

The structure of various standard library functions for both types and thunks was checked manually, to ensure they had parsed correctly. With the earlier unit tests passing, no issues were found when analysing the parser output, an example is shown in Figure 22.

The visualiser output also had to be checked manually. For this a number

```haskell
class (Real a, Fractional a) => RealFrac a where
    properFraction :: (Integral b) => a -> (b,a)
    truncate :: (Integral b) => a -> b
    round :: (Integral b) => a -> b
    ceiling :: (Integral b) => a -> b
    floor :: (Integral b) => a -> b
```

Figure 21: Of all the type classes, "RealFrac" is the only one that contains method that have a constraint, "Integral b".



```
Parser.Prelude(`
isJust :: Maybe a → Bool
isJust Nothing = False
isJust _       = True
`);
```

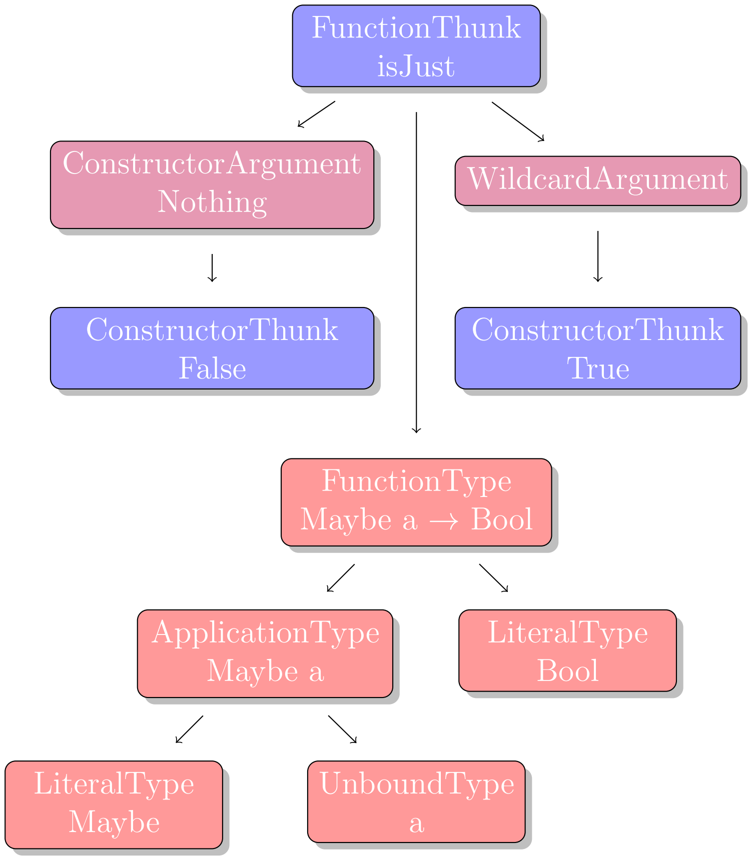
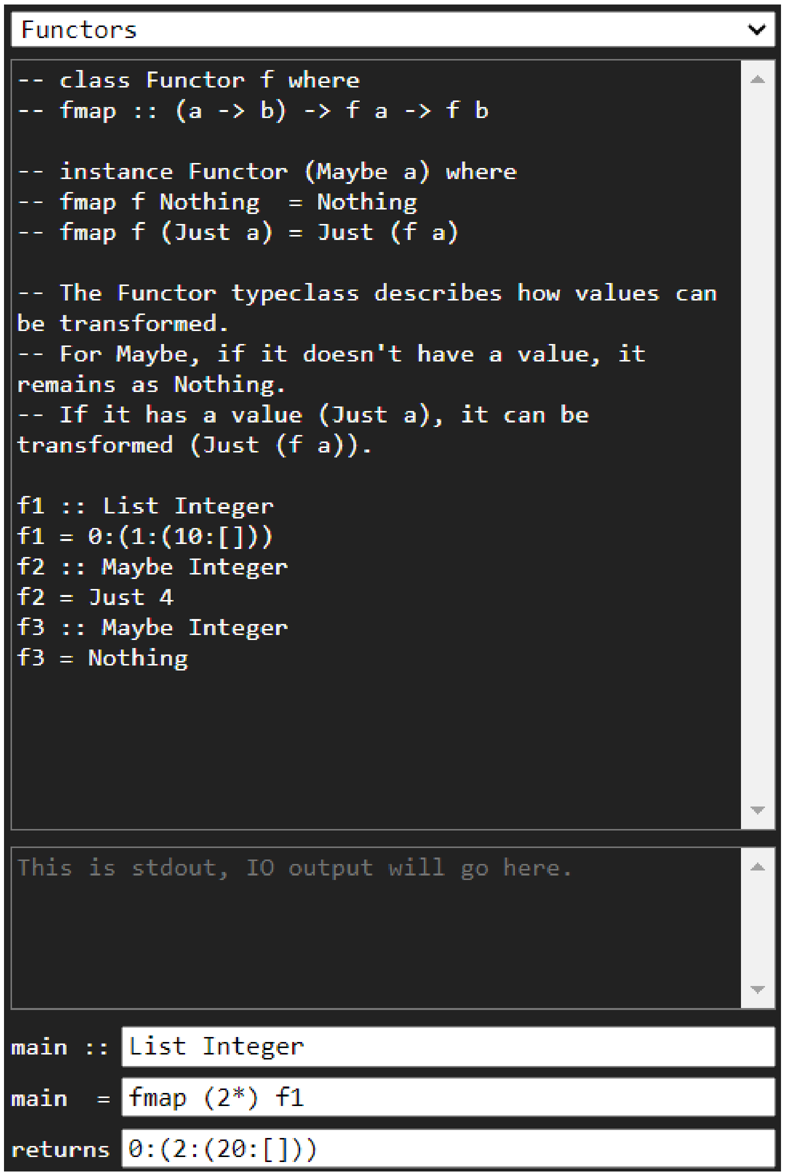


Figure 22: The structure of the `isJust` function. It has 2 patterns with corresponding implementations, and a type.

of preset functions were made to quickly load between, such as the "Functor" preset shown in Figure 23, and then stepped through to check each step to make sure it was correct. This also checked the type information that displayed was correct, and the "main" function displayed the correct final result when evaluated live, such as in Figure 24.



```
-- class Functor f where
-- fmap :: (a -> b) -> f a -> f b

-- instance Functor (Maybe a) where
-- fmap f Nothing  = Nothing
-- fmap f (Just a) = Just (f a)

-- The Functor typeclass describes how values can
be transformed.
-- For Maybe, if it doesn't have a value, it
remains as Nothing.
-- If it has a value (Just a), it can be
transformed (Just (f a)).

f1 :: List Integer
f1 = 0:(1:(10:[]))
f2 :: Maybe Integer
f2 = Just 4
f3 :: Maybe Integer
f3 = Nothing
```

```
This is stdout, IO output will go here.
```

```
main    :: List Integer
main    =  fmap (2*) f1
returns 0:(2:(20:[]))
```

Figure 23: A preset showcasing the Functor type class, and a few examples.

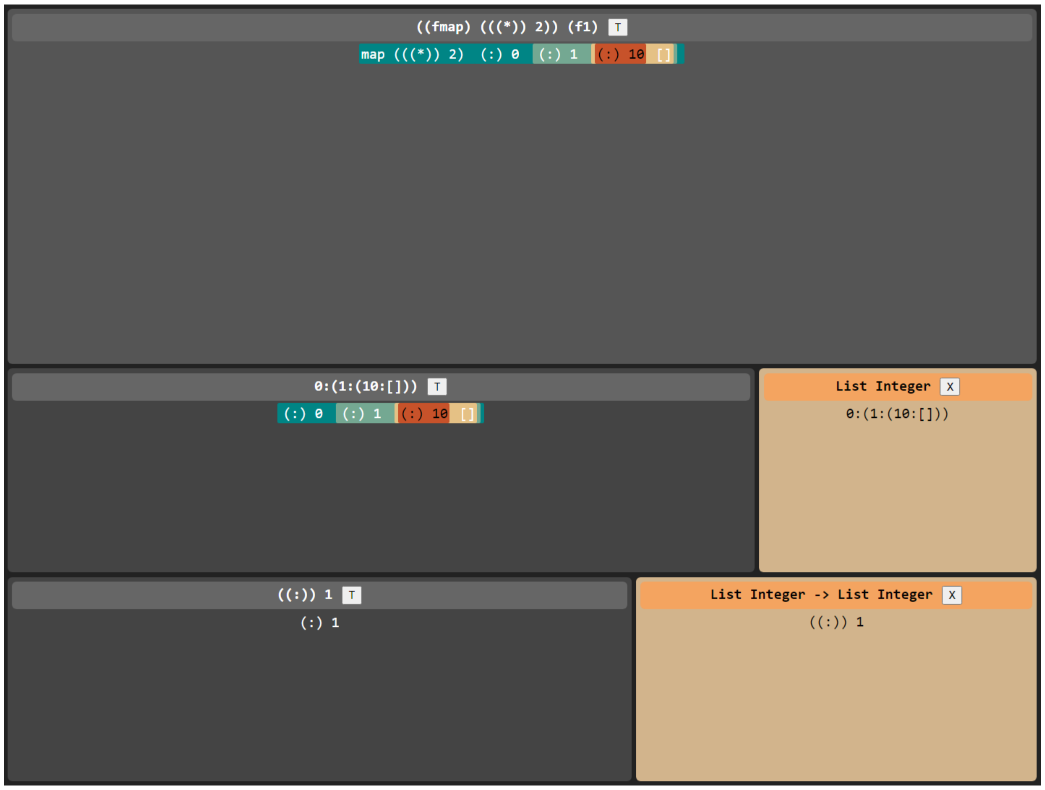


Figure 24: A successful visualisation of the “fmap (2*) (0:(1:(10:[])))” expression mid evaluation.

7 Evaluation

7.1 Requirements

From the testing in Section 6, it's clear that the functional requirements listed in Section 3.1 have been met as required. Although not every feature of Haskell has been implemented, the planned core features required for use: functions, data types, and type classes, as well as IO, have been implemented. The user is able to quickly respond to both syntax and type errors, with the live evaluation features, and can step through expressions, and see the type of expressions using the visualiser.

The following discussions on each non-functional requirement are in the same order specified in Section 3.2.

1. Inputs that use the entire input field, about 500 characters, take a few seconds to fully parse. This is mitigated by only parsing once the user stops inputting for a couple seconds, and is a similar approach used by common live code analysers.

2. As the visualisation only updates when stepped, and is not continually refreshed, there is no noticeable delay between states. This is also improved by splitting of sub-expressions from their parents, as only one expression updates at a time.

3. As shown from a short user evaluation survey in Table 4, on a scale from 1-5 with 1 being the worst, 5 being the best, on average user's rated the visualiser a 3.6 out of 5 for ease of use, and a 3.3 out of 5 for how well it helped with understanding of Haskell and functional programming.

7.2 Further Improvements

7.2.1 The Haskell Language

There are many ways this visualiser could still be improved. Countless features from the Haskell 2010 Language Report [Mar+10] were not implemented to save time and create a working prototype quickly. This includes many of the keywords, such as fixity declarations, as well as extending to GHC Haskell, which has a much larger array of changes and additions from new type classes, to language extensions. There are also a few changes from the Haskel 2010 Language Report, such as lists being called "List a" instead of "[a]". The Standard Library has also not been fully implemented. Missing many functions that handle tuple types, as well as ones that use the "seq" function, that forcefully evaluates a value. Extending to GHC Haskell would also require extending the standard library to the considerably larger GHC standard library.

7.2.2 Performance

While the performance is at an acceptable level, it could be improved. If the user wanted to evaluate a large program, it will take several seconds to parse. During this time, the visualiser temporarily freezes. This also occurs when the visualiser is opened, as it loads each function. This is partially mitigated by splitting parsing on double new lines, to break a large input into many smaller inputs, but the delay is still noticeable. As mentioned, visualisation of expressions has no issues with performance due to only being updated when required, rather than updating all of the expressions every time one expression changes.

7.2.3 Visualisation

While the visualisation works well to show expressions and the thunks within, it is not perfect. Oftentimes there are more or less brackets than expected which can clutter the display or confuse the user respectively. On the other hand, the ability for the user to evaluate part of an expression, without affecting the entire thing, was a well liked feature of this visualiser that is not present in others.

8 Conclusion

This project aimed to help address the problem of learning functional programming, and has successfully done so by creating a visualiser for Haskell that allows a user to see a program evaluate step by step, as well as see the type information of expressions within that program. While not covering all the features of modern GHC Haskell, I feel the project has been a success as I was able to implement the major features I wanted, namely functions, data types, and type classes, so they can evaluate and be visualised in a useful way to the user.

Source Code

The source code, and how to run this project can be found at:
https://git.cs.bham.ac.uk/projects-2023-24/mxh162

A Bibliography

[MM82] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), pp. 258–282. ISSN: 0164-0925. DOI: 10.1145/357162.357169. URL: https://doi.org/10.1145/357162.357169.

[Joh84] Thomas Johnsson. “Efficient compilation of lazy evaluation”. In: *SIGPLAN Not.* 19.6 (June 1984), pp. 58–69. ISSN: 0362-1340. DOI: 10.1145/502949.502880. URL: https://doi.org/10.1145/502949.502880.

[Hug89] J. Hughes. “Why Functional Programming Matters”. In: *The Computer Journal* 32.2 (Jan. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf. URL: https://doi.org/10.1093/comjnl/32.2.98.

[Jon92] Simon L. Peyton Jones. “Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine”. In: *Journal of Functional Programming* 2.2 (1992), pp. 127–202. DOI: 10.1017/S0956796800000319.

[SGC07] Don Syme, Adam Granicz, and Antonio Cisternino. “Lexing and Parsing”. In: *Expert F#*. Berkeley, CA: Apress, 2007, pp. 461–489. ISBN: 978-1-4302-0285-1. DOI: 10.1007/978-1-4302-0285-1_16. URL: https://doi.org/10.1007/978-1-4302-0285-1_16.

[Swa08] Michael Swaine. “It’s time to get good at functional programming”. In: *Dr. Dobbs Journal (December 2008), http://www.drdobbs.com* (2008).

[Mar+10] Simon Marlow et al. “Haskell 2010 language report”. In: (2010).

[HHW15] Zhenjiang Hu, John Hughes, and Meng Wang. "How functional programming mattered". In: *National Science Review* 2.3 (July 2015), pp. 349–370. ISSN: 2095-5138. DOI: `10.1093/nsr/nwv042`. eprint: `https://academic.oup.com/nsr/article-pdf/2/3/349/31566307/nwv042.pdf`. URL: `https://doi.org/10.1093/nsr/nwv042`.

[SA18] Jeremy Singer and Blair Archibald. "Functional baby talk: analysis of code fragments from novice haskell programmers". In: *arXiv preprint arXiv:1805.05126* (2018).

[UoW23] URL: `https://warwick.ac.uk/fac/sci/dcs/teaching/modules/cs141/`.

[UoK23] URL: `https://www.kent.ac.uk/courses/modules/module/COMP5450`.

[UoB23] URL: `https://www.cs.bham.ac.uk/internal/modules/2023/06-34253/`.

[Cou] URL: `https://www.coursera.org/courses?query=functional%20programming`.

[Algo] URL: `https://algorithm-visualizer.org/`.

[CSC] URL: `https://cscircles.cemc.uwaterloo.ca/visualize`.

[CBN] URL: `https://well-typed.com/blog/2017/09/visualize-cbn/`.

[SPARTAN] URL: `https://tnttodda.github.io/Spartan-Visualiser/`.

[ANTLR] URL: `https://www.npmjs.com/package/antlr4`.

[OhmJS] URL: `https://ohmjs.org/`.

[cytoscape.js] URL: `https://js.cytoscape.org/`.

B Survey Results

Question	Responses
Are/have you taken the 2^{nd} year Functional Programming module?	11 - Yes, 3 - Currently taking it.
For Haskell, have you used any tools/visualisers for testing and debugging?	10 - No, 3 - GHCi, 1 - VSCode
Which tools/visualisers have you used for testing and debugging other languages?	5 - VSCode Plugins, 4 - IDE Debuggers, 2 - GDB, 3 - No
How did you find the transition from object-oriented imperative languages to a functional language? (1-5, hard-easy)	1 2 2 3 3 3 3 4 4 4 4 4 5 5
How well would you say you understood the following concepts? (0-1-5, I don't know what that is-not well-very well)	
Higher Order Functions	2 3 4 4 4 4 4 4 5 5 5 5 5 5
Anonymous Functions	3 3 3 4 4 4 4 5 5 5 5 5 5 5
Monads	0 0 1 2 2 2 3 3 3 3 3 4 4 4
Lazy Evaluation	0 0 0 0 3 3 3 4 4 4 4 5 5 5
Partial Application	0 1 2 2 3 3 4 4 4 4 5 5 5 5
Pure Functions	0 0 1 1 2 3 3 4 4 4 5 5 5 5

Table 1: Questions and responses as a preliminary survey.

It was sometimes unclear what exactly designated say a certain class of functions (eg for me, monads) - I could use them, but never fully understand WHY they worked or were a certain way. I tried a lot to mostly just practise lots of different applications of different concepts to get an intuitive feel for them.
Monads were slightly tricky. Visualising them as containers/wrappers around some primitive type, and the bind operator as a transform from one wrapper to another, helped a lot
I mainly found a disconnect between how the concepts were explained and how I would actually apply them within programs
Mainly the lack of examples that we could apply the concepts to. With lazy evaluation and monads, I found that it wasn't until I went away and did my own learning in my free time (with textbooks and videos) that they made total sense and I could use them.
I understand monads at a surface level but not much more. This may be because we weren't required to learn them that deeply but I still didn't find them intuitive
Why are functions pure and impure? I never really figured it out. I remember memorising some functions that "lifted" functions to return a value, but that has left my brain now.

Table 2: Additional comments to the "How well would you say you understood the following concepts?" question.

Yes, mostly because I found them fun
Very likely because it is expressive for some tasks, esp those requiring correctness and notational accuracy with the underlying math
Not very likely because it has bad tooling (stack ¡¡¡¡)
I would be fairly likely as I find recursion intuitive and enjoy the elegance of functional programs
Extremely likely!
I just enjoy using functional languages so I'll use them for fun for personal projects, but also I want to go into research and lecturing in a Theory of Computation department, so they can prove very useful
Fairly likely, since I've worked with them in industry
I'm much more comfortable in oop languages so not likely, but if a use case comes up that is perfect for functional I would.
Unlikely. Most jobs don't require skills related to functional programming
Pretty likely, it seems like an interesting concept that I enjoy programming in the style of.
Python is my main programming language and inherits many aspects of functional programming that I use frequently: list comprehensions, currying & partial functions, Anonymous/lambda functions, higher order functions.
Less likely as it's not that interesting to me
Unlikely - however functional paradigms are something I use in imperative software as a result of having studied purely functional languages.
Honestly, never again

Table 3: Answers to the "How likely are you to use a functional language in the future?" question.

C User Evaluation Results

Question	Responses
How easy was the visualiser to use? (1-5, hard-easy)	1 2 3 3 3 4 4 4 4 5 5 5
How well did the visualiser help with your understanding of Haskell and functional programming? (1-5, not well-very well)	2 2 2 3 3 3 3 4 4 4 5 5

Table 4: Questions and responses given to users after using the visualiser.

D Tables

Type	Use
typeidentifier	A type identifier.
TypeConstructor	A type constructor.
$type_1 \to type_2$	Function type from $type_1$ to $type_2$.
$type_1\ type_2$	Type application, applying $type_2$ to $type_1$.
[*type*]	A list of values with the type *type*.
($type_1$,$type_2$)	The tuple type between $type_1$ of $type_2$.
($type_1$,$type_2$,$type_3$)	The tuple type of $type_1$, $type_2$, and $type_3$.
...	...

Table 5: Various built in type syntax.

Lexeical Notation	Rule
[*pattern*]	optional
{*pattern*}	zero or more repetitions
(*pattern*)	grouping
$pat_1 \mid pat_2$	choice
$pat_{\langle pat' \rangle}$	difference — elements generated by *at* except those generated by *pat'*
`fibonacci`	terminal syntax in typewriter font

Table 6: The notation defined in 10.1 of the Haskell 2010 Language Report.

Symbol	→	Expression
program	→	{*typeclass* | *instance* | *datatype* | *funcDecl* | *funcImpl*}
typeclass	→	`class` [(*context*) `=>`] *conid* *varid* `where` *lf* {*funcDecl*}
instance	→	`instance` [(*context*) `=>`] *conid* `where` *lf* {*funcImpl*}
datatype	→	`data` *conid* {*varid*} `=` {*varid* | *conid* | `(`*type*`)`}
type	→	*btype* [`->` *type*]
btype	→	*atype* [*btype*]
atype	→	`()` | *varid* | *conid* | `(`*type*`)`
context	→	*constraint* [`,` *context*]
constraint	→	*conid* *varid*
funcDecl	→	*var* `::` [(*context*) `=>`] *type*
funcImpl	→	*var* *pattern* `=` *expression*
pattern	→	*argument* [*pattern*]
argument	→	*literal* | *varid* | *conid* | `[]` | `(`*argument*`:`*pattern*`)` | `(`*pattern*`)`
expression	→	*aexpression* [*expression*]
aexpression	→	*literal* | `[]` | `()` | *var* | *conid* | *symbol* | `(`*expression*`:`*expression*`)` | `(`*expression*`)`
var	→	*varid* | `(`*symbol*`)`
varid	→	*small* {*small* | *large* | *digit* | `'`}
conid	→	*large* {*small* | *large* | *digit* | `'`}
literal	→	*int* | *float* | *hex* | *oct* | *char* | *string*
int	→	*digit* {*digit*}
float	→	(*int*`.`*int* [*exponent*]) | *int* *exponent*
exponent	→	(`e` | `E`) [`+` | `-`] *int*
hex	→	`0` (`x` | `X`) *hexit* {*hexit*}
oct	→	`0` (`o` | `O`) *octit* {*octit*}
symbol	→	`!` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `^` | `|` | `-` | `~` | `:`
small	→	a | b | ... | z
large	→	A | B | ... | Z
digit	→	0 | 1 | ... | 9
octit	→	0 | 1 | ... | 7
hexit	→	digit | a | ... | f | A | ... | F
lf	→	\`n`

Figure 25: The grammar of the Haskell-like language.

Lexer Function	Use
all	Matches the grouping rule.
any	Matches the choice rule.
opt	Matches the optional rule.
many	Matches the zero or more rule.
diff	Matches the difference rule.
str	Matches a string exactly.
reg	Matches a regex exactly.
tok	Combines and labels all tokens into one token.
max	Returns the sequence according to the maximal munch rule.

Table 7: The basic building blocks of my lexer combinator.