SCHOOL OF COMPUTER SCIENCE

Implementing a Step by Step Evaluator for a Simple Functional Programming Language

Kiran Sturt

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering **worth 40CP**.

Wednesday 30th April, 2025

Abstract

Students often find functional programming languages more difficult to learn than imperative languages, and they may struggle to gain an intuitive understanding of how functional languages are evaluated. I have created a tool SFL Explorer, available at https://functional.kiransturt.co.uk, which aims to help build intuitive understanding of how functional programming languages work. The primary use case this project has been designed and tested for is for use as a demonstration tool in lectures, particularly in the University of Bristol's own combined imperative and functional programming unit COMS10016. My client, Samantha Frohlich, a lecturer on this unit plans to integrate this system into the unit in future.

The system includes my own functional programming language. SFL is a very minimal language, but it includes many standard functional programming features, including polymorphism, pattern matching and user definable algebraic data types. This language is type checked, using a modified version of Dunfield and Krishnaswami's bidirectional type checking algorithm [8], modified to include SFL's extended type system.

All functionality for the language was written in Rust, compiled to Web Assembly, and included into a React app that acts as the frontend. This functionality is therefore available entirely client side. The app is a Progressive Web App (PWA) and is therefore able to be installed and used offline.

As the system is designed to be a teaching tool, I have done user testing in the form of 3 focus groups at various points throughout the project who are at various stages in the process of learning functional languages. Their feedback ensured that the project stayed on track and remained as useful as possible to potential users of a wide variety of skill levels.

Dedication and Acknowledgements

My supervisors, Jess Foster and Sarah Connolly, have been unwaveringly helpful, supportive and kind throughout this project, as well as my university journey as a whole. I would like to thank them for all of their help, without which this would not have been possible.

I would like to thank Samantha Frohlich for being a really great client, whose enthusiasm for what I was creating really inspired me to do my best work. Likewise, I would also like to thank Dr. Steven Ramsey for being a fantastic lecturer in programming languages; this project would not have been anywhere near as good without the knowledge and inspiration I gained from his lectures.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Kiran Sturt, Wednesday 30th April, 2025

AI Declaration

I declare that any and all AI usage within the project has been recorded and noted within Appendix A or within the main body of the text itself. This includes (but is not limited to) usage of translators (even google translators), text generation methods, text summarisation methods, or image generation methods.

I understand that failing to divulge use of AI within my work counts as contract cheating and can result in a zero mark for the dissertation or even requiring me to withdraw from the University.

Kiran Sturt, Wednesday 30th April, 2025

Contents

List of Figures

Ethics Statement

This project is covered by the blanket ethics application 6683 as determined by my supervisor Jess Foster.

Supporting Technologies

- I used React to develop the website for this project.
- The bindings for the web assembly interface to the library for the language were generated by using macros from the wasm-pack rust crate.
- I used GitHub Copilot to help assist with generating unit tests.

Notation and Acronyms

SFL Simple Functional Language

FP Functional Programming

WASM Web ASseMbly

CLI Command Line Interface

MVP Minimum Viable Product

NPM Node Package Manager

AST Abstract Syntax Tree

Chapter 1

Introduction

In this dissertation I present SFL-explorer: a tool to demonstrate how functional programming languages are evaluated, allowing users to gain a valuable intuition of these languages. It is an open source web based tool, available for download and offline use as a PWA (Progressive Web App, see 2.4).

SFL-explorer takes the form of a functional language (Simple Functional Language (SFL)), packaged with two interfaces that allows users to observe the process of evaluation of a term as a series of step by step or multi-step reductions, and control the order that sub-terms are evaluated. These interfaces are a Command Line Interface (CLI) and a web application. The ultimate goal of this project was to make a tool that makes learning and teaching the basics of functional programming easier. There are two groups of people the project is designed to be of interest to:

- Those involved in learning functional languages. These could be students of a university course, or anyone interested in the topic.
- Those involved in teaching functional languages, as part of a university course or otherwise.

1.1 The Language

The language itself is not meant to be the main interest for the users of this system. It is designed to be fairly generic, with syntax and semantics similar to popular functional languages, so that users can take their understanding from using SFL-explorer and apply it to these languages. 1.1 is an example program in the language, to find the factorial of 2. The relevant prelude functions are included for clarity.

```
if :: Bool -> a -> a -> a
if cond then_branch else_branch = match cond {
  | true -> then_branch
  | false -> else_branch
}

fac :: Int -> Int
fac n = if (n <= 1) (1) (n * (fac (n - 1)))

main :: Int
main = fac 2
```

Figure 1.1: An example SFL program. Evaluation is shown 1.2

1.2 is a table showing the evaluation of this function in lazy mode by the system. The ‘Prompt‘ column shows what the user is presented with as a button to make progress. The first prompt entry at row 0 is empty, as it represents the starting program state. This table is generated dynamically as the user progresses through the given program.

The user is provided with messages telling them what the next step that they can make is. Additionally, there is a ‘free choice’ mode where users are presented with the options for progress, and they can choose which one is taken.

<table>
<tr><th>Step</th><th>Prompt</th><th>Main Expression Afterwards</th></tr>
<tr><td>0</td><td></td><td><code>fac 2</code></td></tr>
<tr><td>1</td><td>Apply function ‘fac’ to 2</td><td><code>if (2 <= 1) 1 (2 * (fac (2 - 1)))</code></td></tr>
<tr><td>2</td><td>Apply function if to (2 <= 1), 1 and (2 * (fac (2 - 1)))</td><td><pre>match (2 <= 1) {
  | true -> 1
  | false -> 2 * (fac (2 - 1))
}</pre></td></tr>
<tr><td>3</td><td>Apply inbuilt <= to ‘2’ and ‘1’</td><td><pre>match (false) {
  | true -> 1
  | false -> 2 * (fac (2 - 1))
}</pre></td></tr>
<tr><td>4</td><td>Match to pattern ‘false’</td><td><code>2 * (fac (2 - 1))</code></td></tr>
<tr><td>5</td><td>Apply function fac to (2 - 1)</td><td><pre>2 * (if ((2 - 1) <= 1) 1 ((2 - 1)
    * (fac ((2 - 1) - 1))))</pre></td></tr>
<tr><td>6</td><td>Apply function if to ((2 - 1) <= 1), 1 and ((2 - 1) * (fac ((2 - 1) - 1)))</td><td><pre>2 * match ((2 - 1) <= 1) {
  | true -> 1
  | false -> (2 - 1) * (fac ((2 - 1) - 1))
}</pre></td></tr>
<tr><td>7</td><td>Apply inbuilt - to 2 and 1</td><td><pre>2 * match (1 <= 1) {
  | true -> 1
  | false -> 1 * (fac (1 - 1))
}</pre></td></tr>
<tr><td>8</td><td>Apply inbuilt <= to 1 and 1</td><td><pre>2 * match (true) {
  | true -> 1
  | false -> 1 * (fac (1 - 1))
}</pre></td></tr>
<tr><td>9</td><td>Match to pattern true</td><td><code>2 * 1</code></td></tr>
<tr><td>10</td><td>Apply inbuilt * to 2 and 1</td><td><code>2</code></td></tr>
</table>

Figure 1.2: A table showing how the system leads a user through the step by step evaluation of the program shown in Figure 1.1

1.2 Agile Development Lifecycle

The project followed a development lifecycle inspired by Agile principles[3], structured into four iterative phases. Each phase built upon the last, integrating evaluation and feedback to continuously and rapidly refine the features and the UI/UX of the system.

Each phase was further subdivided into four phases:

- **Requirements Gathering**
- **Design and Research**
- **Implementation**
- **Evaluation**

This iterative methodology helped manage complexity and uncertainty. Getting frequent feedback from focus groups and other sources throughout the project ensured that the project stayed on course

1.2.1 User Testing

[TODO: finish] This desired outcome of this project is an effective learning/teaching tool for functional languages. As such, user testing is vital for ensuring that the system is usable and intuitive, and therefore effective. I conducted user testing throughout the

I tested my system in 3 separate ways. I held 3 focus groups with 12 people in total, all with varying levels of experience with functional programming.

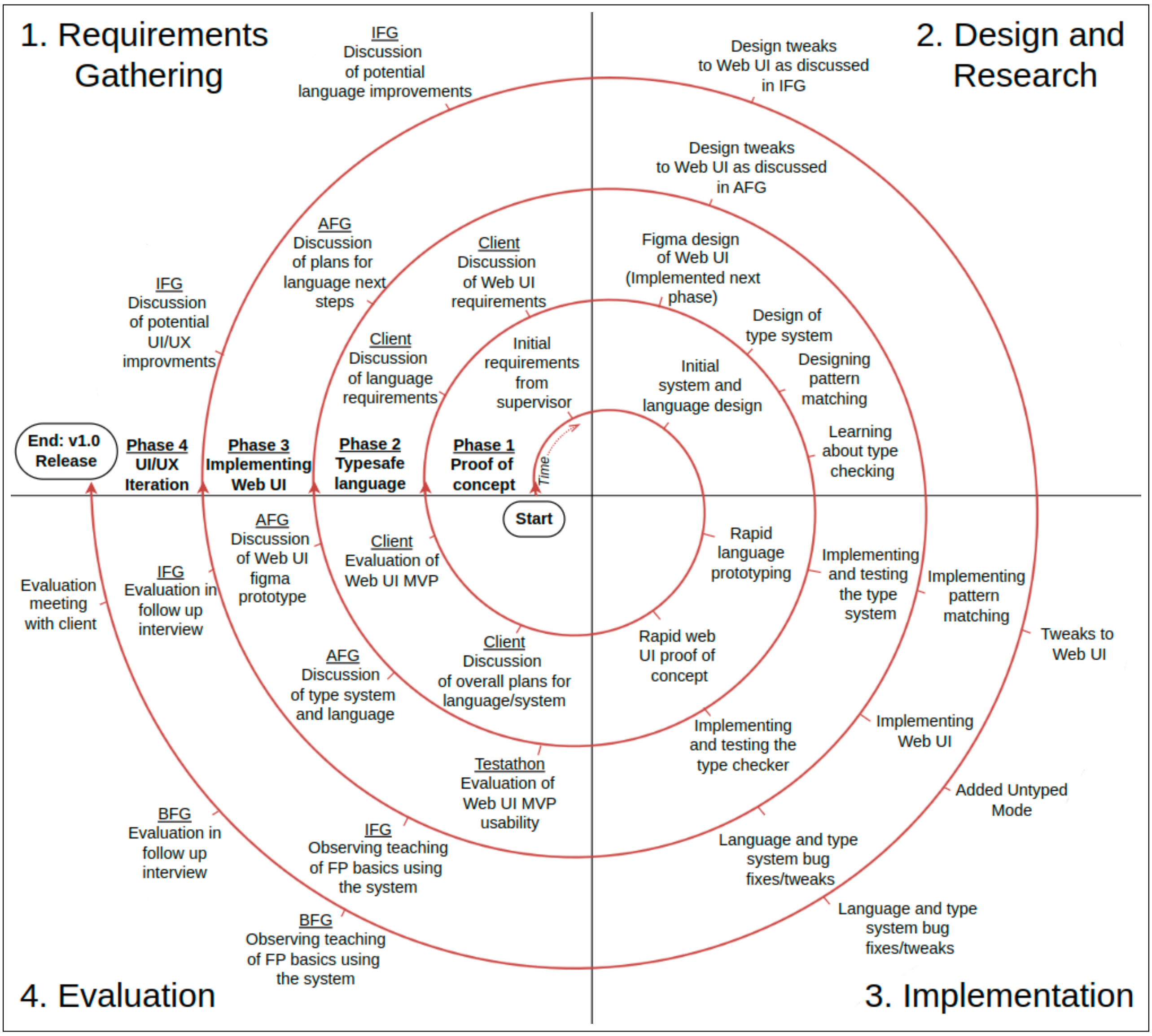


Figure 1.3: A spiral representation of the project lifecycle, showing the 4 iterations, and the work done in each part of each phase.

Chapter 2

Background

2.1 The Lambda Calculus

The lambda calculus (λ-calculus) was described by Alonzo Church in 1936 [5]. It is a universal model of computation, meaning we can implement any Turing machine using it, meaning we can compute any computable function using it [29]. The λ-calculus is important because it provides a foundation for functional programming languages. Understanding the λ-calculus is essential for understanding the principles behind these languages and how they evaluate expressions.

The set of all lambda terms is Λ. Lambda calculus is built from three syntax structures:

- Variables, selected from an infinite set of variables $V = \{x, y, z, ...\}$
- Abstractions, $\lambda x.M$ which are functions, where we 'bind' a variable x for use in the term M such that when we apply our function to a term N, all instances of x in M are substituted with N.
- Applications MN where we apply a term M to an argument N.

Below is a more formal definition of the λ-calculus [2].

$$\begin{array}{lcl} x \in V & \implies & x \in \Lambda \\ M, N \in \Lambda & \implies & (MN) \in \Lambda \\ M \in \Lambda, x \in V & \implies & (\lambda x.M) \in \Lambda \end{array}$$

We shall also use the following fairly standard conventions:

1. Application is left associative. The term $M_1\, M_2\, M_3$ means $(M_1\, M_2)\, M_3$ and not $M_1\, (M_2\, M_3)$
2. Nested abstractions can be grouped: the term $\lambda x\ y.M$ means $(\lambda x.(\lambda\ y.M))$.
3. Outermost parenthesis are omitted.
4. The body of an abstraction extends as far to the right as possible: the term $\lambda x.M\, N$ means $(\lambda x.(M\ N))$ and not $((\lambda x.M)\, N)$.

2.1.1 Free variables

> 'An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction on x' [25]

Free variables are a useful concept to express which variables are 'ready for substitution' in a term. Formally, the function $FV(M)$ is the set of free variables in the term M [2]:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(M\, N) & = & FV(M) \cup FV(N) \\ FV(\lambda x.M) & = & FV(M) - \{x\} \end{array}$$

A term is *closed* if it has no free variables, and *open* if it does.

2.1.2 Reduction

> ‘The sole means by which terms ‘compute’ is the application of functions to arguments (which themselves are functions). Each step in the computation consists of rewriting an application whose left-hand component is an abstraction, by substituting the right-hand component for the bound variable in the abstraction’s body’ [25]

The λ-calculus is evaluated by β-reduction. This is where an abstraction is applied to a value. The result of applying an abstraction to a term is the body of the abstraction, with the all free instances of the abstracted variable are substituted with the term the abstraction was applied to. Below is the definition of substitution within a term formally [2].

$$
\begin{array}{lcl}
x[x := N] & \equiv & N \\
y[x := N] \text{ where } y \neq x & \equiv & y \\
(M_1 M_2)[x := N] & \equiv & (M_1[x := N])(M_2[x := N]) \\
(\lambda y.M)[x := N] & \equiv & \lambda y.(M[x := N])
\end{array}
$$

The definition of β-reduction [2]:

$$
\begin{array}{lcl}
x & \to_\beta & x \\
\lambda x.M & \to_\beta & \lambda x.M \\
(\lambda x.M)N & \to_\beta & M[x := N]
\end{array}
$$

A term is said to be in **normal form** if it cannot be β-reduced. A term that can be beta reduced can be said to be a **redex**; a reducible expression. The resulting term from reducing a redex is often called a **contraction**.

2.1.3 Reduction, Evaluation Strategies and Values

We often have a term where we have multiple options for β-reduction. In this section, we will briefly discuss three different evaluation strategies that inform which option is selected when evaluation: call-by-value (a.k.a. strict), call-by-name and call-by-need (a.k.a. lazy). When a term is fully reduced under a given evaluation strategy, we say it is a value. I have used the same examples as Pierce [25]. For instance, the closed term

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

has three redexes (id is shorthand for $\lambda x.\, x$):

$$
\begin{array}{c}
\underline{id\,(id\,(\lambda z.\, id\, z))} \\
id\,\underline{(id\,(\lambda z.\, id\, z))} \\
id\,(id\,(\lambda z.\, \underline{id\, z}))
\end{array}
$$

Most languages use the **call-by-value** evaluation strategy, where ‘only the outermost redexes are reduced and a redex is reduced only when its right-hand side has already been reduced to a value … where the only values are expressions’[25]. This reduction strategy is also known as ‘strict’. Our chosen redex here would be number 2. We would reduce by the following sequence:

$$
\begin{array}{rl}
id\,\underline{(id\,(\lambda z.\, idz))} & \to \\
(\underline{id\,(\lambda z.\, idz)}) & \to \\
(\lambda z.\, idz) &
\end{array}
$$

The **call-by-name** strategy is less restrictive. ‘The leftmost outermost redex is always reduced first …and allows no reductions inside abstractions’ [25]. Only abstractions are valid values. Our reduction sequence would look like this:

$$
\begin{array}{rl}
\underline{id\,(id\,(\lambda z.\, idz))} & \to \\
(\underline{id\,(\lambda z.\, idz)}) & \to \\
(\lambda z.\, idz) &
\end{array}
$$

Call-by-need is similar to call by name but with sharing. This means we ‘overwrite all occurrences of an argument with the value the first time it is evaluated’ [25]. Only abstractions are valid values once again. This is an optimization of call-by-name that is known as lazy evaluation. It is used by many functional languages including Haskell, and is of particular importance to SFL explorer. In this case, our reduction order would be the same as **call-by-name**.

2.1.4 Types

If we were to extend our λ-calculus with a new sort of term, an integer literal (something commonly done, especially when building up to discussing practical functional languages)

$$\ldots, -2, -1, 0, 1, 2, \ldots$$

we could say that these values are members of a set of values Int.

In this extended value of the lambda calculus, the set of valid values V becomes:

$$V ::= \lambda x.M \mid \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$$

It would be useful for us to be able to assert that a term eventually evaluates to one of these Ints. The λ-calculus terms that evaluate to a value in the set of Ints can all be said to have 'type' Int. More generally:

> 'Saying that 'a term t has type T' (or 't belongs to T' or 't is an element of T') means that t 'obviously' evaluates to a value of the appropriate form - where by 'obviously' we mean that we can see this statically, without doing any evaluation of t' [25]

For instance, the term $(\lambda x.\, 1)\, 2$ has type Int, as it evaluates to a value in the set of valid Int values.

Functions

We want to be able to express the types of functions. The term $\lambda x.x$ can be said to have type $T \to T$, as it takes in a term of type T and returns the same term, which still has type T.

A more complex term $\lambda x\ y.x$ can be said to have type $T \to (U \to T)$; If we give it a term M of type T, it would return the function $\lambda y.M$ which takes whatever is given to it (represented by U) and returns M which has type T.

Judgements

[TODO: To make typechecking bit easier]

By convention, $\to$ is right associative so way may omit the right most parenthesis.

2.1.5 Typechecking: Well typed programs do not go wrong

The λ-calculus that we have discussed so far is untyped. This means that any λ term can be applied to any other term. However, this may result in things that can no longer be reduced, but are not valid values. The evaluation of an λ-calculus expression is said to have 'gone wrong' if it gets to a normal form that is not a valid value.

Let us consider the expression and its reduction

$$(\lambda x.x\ x)\ 1 \to_\beta 1\ 1$$

The reduction is not a valid value, however it cannot be further reduced. We have *gone wrong*.

We will now attempt to derive the type of the parameter x, in order to show that it is untypeable. As x here is applied to itself, it must be some kind of function that takes a term x of with type T and then applies it to itself. This means that $T = T \to U$ which is absurd. This means that this is 'untypeable'. Indeed, it is clearly never possible to type an expression where a term is applied to itself. If this was a real programming language, when it got to the normal form 1 1, it would be some form of runtime error. We can see that only allowing typeable terms would have prevented us from creating a term that does not evaluate in this case.

In general, **well typed programs do not go wrong** [19]. Therefore, if we are able to exclude all terms in our functional language that are untypeable, we will be able to guarantee that it does not go wrong, and thus prevent these runtime errors. The system that looks at a program to decide whether it is well typed is called the **typechecker**. Types can often also be inferred without specific assignments, which is called **type inference**.

2.2 Haskell: A Functional Programming Language

Functional programming languages are programming languages where 'computation is carried out entirely through the evaluation of expressions' [10]. Functional programming languages are based on the lambda calculus. In this section, we will only discuss one example: Haskell.

Haskell is a very prominent functional programming language that is widely taught. It is a programming language specifically designed to be suitable for teaching [12]. This dissertation involves the development of a programming language with some similar features to Haskell, so the corresponding Haskell features and ideas will be introduced here.

2.2.1 Declarations

Haskell, along with most other languages, provides the facility to name functions and values for reference elsewhere in the program. These can be typed, but the types can almost always be inferred.

Some examples of these declarations, all typed for clarity, are below. For instance, the top level declaration

```
x :: Int
x = 5
```

means that x is equal to 5. We can also name lambda functions:

```
add :: Int -> Int -> Int
add = \x y -> x + y
```

This can be shortened to:

```
add :: Int -> Int -> Int
add x y = x + y
```

2.2.2 Polymorphic functions

In Haskell, functions can be written that operate on values of various types. A simple argument is

```
id :: a -> a
id x = x
```

which simply returns its argument. 'a' in the type signature represents any type, and can be substituted for any type. The two 'a's must be the same however, mandating that the argument and the return value must be the same type.

2.2.3 User Defined Algebraic Data Types

Many languages, including Haskell, have Algebraic Data Types allowing us to 'Compose' other data types. The set of all values of an algebraic data type is isomorphic to an expression involving the sets of values of their constituent types combined using 'set algebra' operations. Haskell allows for 'union' and 'product' types.

Haskell allows users to define their own algebraic data types using the `data` keyword. For instance, booleans can be defined as

```
data Bool = True | False
```

This data definition creates a type *Bool* with two data constructors, `True` and `False`. These data constructors are zero-ary. We can also have data constructors that have arguments.

An example of a union type in Haskell is the tagged union

```
data Shape = Circle Int | Rectangle Int Int
```

which is isomorphic to the type $Int \cup (Int \times Int)$.

An example of a product type is the tuple $(Int, Bool)$: the set of all possible values of this type is isomorphic to the Cartesian product of the set of all values of Int and the set of all values of $Bool$. Most languages have product types, which often take the form of structs or tuples.

2.2.4 Polymorphic Types and Kinds

Haskell includes polymorphic types. These are 'types that are universally quantified in some way over all types' [11].

One example is the type constructor $Maybe$, written as $Maybe\ a$. This is first-order polymorphism, as opposed to higher-order polymorphism where a type can be an 'abstraction over type constructors'. [30]. Here, $Maybe$ is not a type in itself, but it represents a constructor that takes a type, and returns a concrete type.

The 'Type of a Type' is its *kind* [25]. For example, the type constructor $Maybe$ has the kind $* \to *$. This notation looks similar to how functions over values are defined, indicating that it behaves like a function, but at the type level rather than the value level. If we were to apply the constructor $Maybe$ to the concrete type Int, the resulting type would be the concrete type $Maybe\ Int$.

'Either' is a type constructor with the kind $* \to * \to *$, meaning it takes two concrete types and returns a concrete type.

2.2.5 Pattern Matching

Haskell allows us to do pattern matching, allowing conditional execution based on whether a term matches a given form. The following function would have different results depending on whether the input value was 0 or another number

```
isZero :: Int -> Bool
isZero 0 = true
isZero _ = false
```

The underscore '_' represents a wildcard pattern that matches anything. In this case, it matches any *Int* that is not 0. We can match more complicated expressions, and assign variables throughout the pattern.

```
data SomeValues a = One a | Two a a | Three a a a | Four a a a a

valuesToList :: SomeValues a -> [a]
valuesToList (One x) = [x]
valuesToList (Two x1 x2) = [x1, x2]
valuesToList (Three x1 x2 x3) = [x1, x2, x3]
valuesToList (Four x1 x2 x3 x4) = [x1, x2, x3, x4]
```

In the first match case of `valuesToList`, we assign the variable x when matching the pattern. In the second, we assign `x1` and `x2` etc.

2.3 Rust

This project is written in rust. [Sam: why? motivate] Some of the decisions made, particularly in the implementation of the AST, require an understanding of Rust, especially the memory management model.

'Ownership' is an important concept. The rules of ownership [16]:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

If a value is owned in one scope, but another scope needs to read/write it, we may use a reference to the value. The rules of references [16]:

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

These rules ensure that immutable references are to things that don't change, and all references are always to things that exist.

2.4 Frontend Technologies

- Vite
- React
- NPM
- PWAs

2.5 Web Assembly

This project runs entirely within the browser, despite being written in Rust. This is due to the fact that it compiles to web assembly. Automated tools exist for the generation of JavaScript bindings around Rust functions/types, but this process places certain restrictions around their arguments and return type, or attributes. We will discuss this here to allow us to refer to these restrictions, and also to explain the process of compiling and using Rust code in a modern web browser.

Web Assembly 2.0 is a 32 bit target [27]. This means we only have 4 GB of addressable memory. The Rust compiler is based on LLVM, which provides a web-assembly compilation target. The Rust compiler has a toolchain around this compilation target [REFHERE: rust WASM toolchain], that allows for easy compilation to web-assembly. However, this only creates a binary blob, which requires more work to make interoperable with our JavaScript build system (Vite). We must do two things to achieve interoperability:

- Incorporate it into our build so it can be served with it.
- Load the WASM package in a way that allows for us to call the functions.

Producing an NPM package with some JavaScript functions that call the WebAssembly functions would achieve both of these goals. However, if we wish to use TypeScript, we must create a separate type definition file that contains the types of all of the JavaScript wrapper functions around the WASM functions. This would be difficult to maintain manually as we would have to update it every time we made a change to the public interface of our rust library.

Fortunately, the rust crate `wasm-bindgen` provides macros that generate a whole NPM package, including TS bindings, automatically. This package can then be added as a dependency to an NPM app that provides a website, and the functions within it can [TODO: WASM-bindgen vs WASM pack?]

wasm-pack

2.6 Existing systems

Below are the most relevant existing systems for SFL explorer.

2.6.1 Duet, and Duet Delta

Duet is:

> 'A tiny language, a subset of Haskell (with type classes) aimed at aiding teachers teach Haskell' [6]

```
data List a = Nil | Cons a (List a)
foldr = \f z l ->
  case l of
    Nil -> z
    Cons x xs -> f x (foldr f z xs)
foldl = \f z l ->
  case l of
    Nil -> z
    Cons x xs -> foldl f (f z x) xs
list = (Cons True (Cons False Nil))

main = foldr _f _nil list
```

Figure 2.1: An example Duet program provided in the repository. _f and _nil are not defined, but the underscore indicates that this is fine and they should just be left unchanged

When running Duet [6] on the program shown 2.1, we get a large block of text as output that shows the reduction or this program. This all happens at once, and we do not get the chance to pick reduction order. It is also quite hard to tell what is going on, as it is dumped as one block of text. The author, Chris Done, also hosts a website where one can try it out without installing it c [7]. The website does not provide much in the way of a UI, it is a text box input and a text box output for running Duet programs.

The main strong point of this project is the language. It is a solid subset of Haskell that includes many similar features to SFL. For this project, I did not draw direct inspiration from Duet or Duet Delta. This was because even though I liked the subset of Haskell selected for Duet, I wanted to attempt to design a clearer language, and potentially break away from being a Haskell subset. Furthermore, Duet and Duet Delta focus on the language, not the UX/UI, whereas I wanted SFL explorer to be strong in both regards.

```
foldr _f _nil list
(\f z l ->
    case l of
        Nil -> z
        Cons x xs -> f x (foldr f z xs))
    _f
    _nil
    list
(\z l ->
    case l of
        Nil -> z
        Cons x xs -> _f x (foldr _f z xs))
    _nil
    list

...

_f True
  (_f False
     ((\l ->
         case l of
           Nil -> _nil
           Cons x xs -> _f x (foldr _f _nil xs))
        Nil))
_f True
  (_f False
     (case Nil of
        Nil -> _nil
        Cons x xs -> _f x (foldr _f _nil xs)))
_f True (_f False _nil)
```

Figure 2.2: The output of evaluating the program shown in Figure 2.1. The beginning and end are shown, with the middle removed.

2.6.2 λ-Lessons

λ-Lessons is a website created by Jan Paul Posma and Steve Krouse at YC Hacks '14 [14]. It is a very effective demonstration of 'map', 'foldr' and 'foldl:

> 'A short, interactive lesson that teaches core functional programming concepts. It was designed to transform the way you think about performing operations on lists of things, by showing you how functions are executed' [14]

It is unfortunate I only found out about it at the end of phase 4 of the project, otherwise my project could have drawn inspiration in terms of UI for free choice evaluation. Indeed, I only discovered this project through correspondence with Chris Done, the author of Duet 2.6.1. Despite the fact that my project did not take inspiration from this system, it would be remiss not to mention it.

My particular favorite features of λ-Lessons's UI are:

- It allows the user to select in the code what part they want to reduce next by clicking on it.
- It also shows informations about functions when you hover over them.
- The language looks like Haskell. However, I had many problems with the functionaltiy of the language that will be discussed later.

The UX/UI is the definite strong point of λ-Lessons(see 2.3). However, there are a few things that I would identify as weaknesses that make it not useful for what SFL explorer is useful for.

- The language is not typechecked. There are type assignments, but upon testing and inspection of the source code [15], type assignments are ignored.
- The language does not allow for user defined algebraic data types, but it has `List` built in.

```
(map addOne [1,2,3,4,5]) (edit) (clear)
((addOne 1) : (map addOne [2,3,4,5]))
((addOne 1) : ((addOne 2) : (map addOne [3,4,5])))
((1 + 1) : ((addOne 2) : (map addOne [3,4,5])))
(2 : ((addOne 2) : (map addOne [3,4,5])))
(2 : ((2 + 1) : (map addOne [3,4,5])))
(2 : ((2 + 1) : ((addOne 3) : (map addOne [4,5]))))
(2 : (3 : ((addOne 3) : (map addOne [4,5]))))
(2 : (3 : ((3 + 1) : (map addOne [4,5]))))
(2 : (3 : ((3 + 1) : ((addOne 4) : (map addOne [5])))))
(2 : (3 : ((3 + 1) : ((addOne 4) : ((addOne 5) : (map addOne []))))))
(2 : (3 : ((3 + 1) : ((addOne 4) : ((addOne 5) : [])))))
(2 : (3 : ((3 + 1) : ((addOne 4) : ((5 + 1) : [])))))
(2 : (3 : ((3 + 1) : ((4 + 1) : ((5 + 1) : [])))))
(2 : (3 : ((3 + 1) : ((4 + 1) : (6 : [])))))
(2 : (3 : ((3 + 1) : (5 : (6 : [])))))
(2 : (3 : ((3 + 1) : (5 : [6]))))
(2 : (3 : ((3 + 1) : [5,6])))
(2 : (3 : (4 : [5,6])))
(2 : (3 : [4,5,6]))
(2 : [3,4,5,6])
[2,3,4,5,6]
```

Figure 2.3: Evaluation of '`map addOne [1, 2, 3, 4, 5]`' with λ-Lessons

```
(map (plus 1) [1,2,3,4,5]) (edit) (clear)
(( 1) : (map (plus 1) [2,3,4,5]))
(( 1) : (( 2) : (map (plus 1) [3,4,5])))
(( 1) : (( 2) : (( 3) : (map (plus 1) [4,5]))))
(( 1) : (( 2) : (( 3) : (( 4) : (map (plus 1) [5])))))
(( 1) : (( 2) : (( 3) : (( 4) : (( 5) : (map (plus 1) []))))))
(( 1) : (( 2) : (( 3) : (( 4) : (( 5) : [])))))
(( 1) : (( 2) : (( 3) : (( 4) : [( 5)]))))
(( 1) : (( 2) : (( 3) : [( 4),( 5)])))
(( 1) : (( 2) : [( 3),( 4),( 5)]))
(( 1) : [( 2),( 3),( 4),( 5)])
[( 1),( 2),( 3),( 4),( 5)]
```

Figure 2.4: Evaluation of '`map (plus 1) [1, 2, 3, 4, 5]`' with λ-Lessons. It gets confused by currying and partial application

- As can be seen in 2.3, it seems to imply that `((x : y : []))` reduces to `[x, y]` which is misleading.
- The language does not support lambda functions
- The language does not support currying (2.4)
- The program states are not saved between refreshes.

In summary, λ-Lessons is designed for a different purpose than SFL Explorer. It describes itself as a 'document' [14] rather than a tool, which is an apt description as it is a document meant to teach one specific thing rather than to be an all around teaching tool for functional languages. It does not provide much of a capability to experiment yourself as the language is not very extensive. However, the ability to reduce an expression by clicking on the bit of the expression you want to reduce is very good, and inspiration should definitely be drawn for future work for SFL Explorer.

2.7 COMS10016: Imperative and Functional Programming at the University of Bristol

In the first year of most computer science programs at the University of Bristol, students take the module COMS10016, a combined imperative and functional programming module. This is many students first encounter with both of these types of programming. In the functional part of this unit, students are taught Haskell. The unit material is presented to students through a very effective lecture series, supplemented by weekly worksheets that students have the opportunity to work through in labs attended by the lecturers, as well as some teaching assistants. Two of the lecturers in this unit are Jess Foster and Samantha Frohlich.

> ‘The aim [of the functional portion of the unit] is to introduce types and functions. Important principles include datatypes, evaluation order, higher-order functions, and purity’ [21]

I acted as a teaching assistant in the labs for two academic years. My role was to answer students questions about functional languages or the worksheets they were given. The inspiration for this project came from my experience struggling to explain key functional programming concepts.

Chapter 3

Phase 1 — Proof of Concept

The goal of Phase 1, which spanned approximately the first month of my project, was to arrive at a proof of concept. This phase started at the beginning of the project with a discussion my supervisor, and the identification of my client. I proceeded by analysing the project requirements using autoethnographic methods, designing the system, designing SFL and the Explorer, and then implementing the proof of concept. I then evaluated the merits and drawbacks of the proof of concept by speaking to my client.

3.1 Requirements Analysis

3.1.1 Autoethnography

> 'Autoethnography is an ethnographic method in which the fieldworker's experience is investigated together with the experience of other observed social actors. [26]'

In this phase, I took an autoethnographic approach to requirements analysis and to design. As the 'fieldworker', I drew on my own experience being involved in teaching Haskell for the last two academic years. This experience was very valuable to this project, and it allowed me take the initial brief from my supervisor and effectively design a solution, and then quickly implement a proof of concept of this solution.

3.1.2 The Brief

This project was proposed by my supervisor, Jess Foster. In our initial meeting, we discussed how she wanted a tool that would help build intuition for how functional languages are evaluated, that she could use to supplement her explanation of otherwise difficult to intuit functional language concepts. We also discussed the benefits of the tool being accessible to students to use themselves during labs or at home. Jess helped me to identify an appropriate client: Samantha Frohlich. Jess and Samantha are both lecturers on COMS10016. It was necessary to identify a client other than Jess, as her existing role as my supervisor/primary marker could limit guidance she would be able to give me if she were also my client.

Following this meeting, I broke down this brief into smaller parts. Taking an autoethnographic approach, I used my own experience teaching functional languages to consider solutions and come up with requirements for each part.

Building Intuition

This is the key to an effective solution. Most students of the first year Functional Programming (FP) unit do not have any experience with functional programming.

In my experience teaching FP, a very effective way to build intuition for functional programming languages is to demonstrate evaluation step by step. I frequently wrote out evaluations on paper for student during the COMS10016 labs. I would also ask students to complete sections themselves. Others have also found that encouraging stepwise evaluation on paper is an effective way to get 'a feeling for what a program does'. [4] [TODO: theres more things to cite here]

Thus, a tool to perform these step by step evaluations in an interactive manner would be very valuable. The tool should have an interface that allows progress to be made step by step, showing the history of past steps as well as giving information about the step about to be taken. This would allow students to understand and interact with a stepwise evaluation, without anyone having to undertake the long process of writing it out, and without risk of incorrectness. Furthermore, the effects of changing the input program could be seen quickly, providing instant feedback.

Use as a Lecture Tool

The tool should be suitable for use in lectures. It should provide an interface that facilitates quality explanation of functional programming languages. The interface must be understandable, for both FP 'experts' (lecturers, advanced users) as well as people who have never seen a functional language before. The tool must also be portable, and not require a complex installation process.

Use as a Self Teaching Tool

The tool should be 'self-explanatory' enough for people to use it on their own without expert help. It should be fairly intuitive, and should have all the information required to use it presented to users. The tool must also be portable, and not require a complex installation process. The less complex this tool is to use, the more people will use it.

Demonstrating FP languages

The tool must contain, at its core, a functional language in order to demonstrate how they work. A language that is similar to Haskell would make for easier evaluation of the project, as this would match the language taught in COMS10016, and therefore more people at the University of Bristol will be able to engage with the language. The most Haskell-like programming language that exists (as far as I am aware) is Haskell.

Haskell could be included in the system/required to be installed on the host machine, however creating a demonstration tool around Haskell would be difficult due to the sheer size of the language, the number of features, and the complexity of the type system. It would be better to create Haskell-like language with a strictly limited size and maximum clarity, and include this in the system. The programming language should be designed with simplicity and clarity at its heart.

3.1.3 SFL Explorer

The requirements extracted from autoethnographic methods, as well as from my initial supervisor meeting, came together to form the idea for SFL Explorer.

The system should be a website to maximize portability. The system should include a functional programming language, as well as some sort of UI that allows a functional program in the language to be entered, and evaluated step by step in a visual manner.

The Simple Functional Language

The language was given a name reflecting its core design principle: Simplicity. More precisely, the programming language should be designed with the following design principles in mind:

1. **It should be simple and easy to understand**. This requires that the language should not have features that users might find difficult to understand why they work. This means that the language should have very few inbuilt functions, all of which should be easy to understand why they work.

2. **It should be similar to existing functional languages**. This would allow users to be able to transfer their intuition to other languages. It should be similar in syntax (it should have similar tokens and structures), as well as semantics (it should work similarly).

3. **It should be powerful enough to explain key concepts.**

The features that should be selected for the language are the features that maximize these goals for the minimum implementation complexity. Out of our design goals, 2 and 3 have the potential to be in conflict, as more expressive power often requires more complex syntax. We must ensure a sensible compromise between all of our goals, while accounting for implementation complexity. When adding features for the language, we must prioritize the features that allow explanation of the 'core' features of functional languages, and de-prioritise features that are not so 'core' to the understanding of functional languages.

The Explorer

The website (the explorer) should include a code editor for people to enter programs. Including the functionality for the language inside the website rather than requiring complex client/server communication would simplify the system, as well as improving responsiveness.

[TODO: Finish this bit, summarize requirements]

3.2 Language Design

In this section, I will discuss the design of SFL with respect to the requirements. This is iteration 1 of the design, and it was the proof of concept.

3.2.1 Definitions

Definition 1 (Lowercase and Uppercase ID syntax as regular expressions)
(Lowercase Identifier): $id ::= [a..z][a..zA..Z0..9\_]*$
(Operator): $op ::= + \mid - \mid \times \mid / \mid > \mid \geq \mid < \mid \leq \mid == \mid != $
(Uppercase Identifier): $Id ::= [A..Z][a..zA..Z0..9\_]*$

[Sam: im so sorry but this looks ugly. I think it just needs more space, it is very crampt, this is a minor thing and can wait till the very end]

3.2.2 Basic Syntax

Lambda calculus is the basis of modern functional programming languages. As discussed in the background, lambda calculus consists of 3 structures: identifiers, application, and abstraction. [Sam: sadly this effectly says in prose the same as the BG, the only thing the BG adds is the formal definition, which not everyone will be able to read] One common extra structure that functional languages implement is an assignment. This is where we label an identifier with a certain meaning, such that all references to the assignment henceforth are identical to a reference to the meaning assigned. For instance:

```
f = (\x.x)
main = f y
```

Is identical to [Sam: dont do this, cos it gives latex the chance to mess up your formatting. Always give your listings a name and use the name in a full sentence e.g. ”For instance, listing 1 and listing 2 are semantically equivalent”]

```
main = (\x.x) y
```

Note the use of "\" instead of λ as it is the closest character available on most keyboards. A program is then defined as a set of assignments, and we pick one specific label name to mark the ‘entry-point’ expression in the program. Haskell, as well as many other languages, uses ‘main’ to represent a programs’ entry point, so we may use main.

Most programming languages, including functional ones, at least support integers. Booleans are also often supported to represent the results of integer comparison. Without literal values, programs would have to use complicated encodings (such as church numerals) to represent these values, making programs look more complicated. We must also add a way to represent values, such as integers and booleans, to our language. These two features massively shorten and simplify programming in this language.

[Sam: this is not referenced anywhere. Above you say we need x and then just dump x here. Instead motivate, introduce and explain. What additions have been made to execute x. Explain your definitions like you would explain your code]

Definition 2 (The basic syntax of SFL)
(Expression) $E, F ::= [-][0,1,..] \mid E \; op \; F \mid true \mid false \mid id \mid \backslash id.E \mid E \; F$
(Assignment) $A ::= id = E$
(Module) $M ::= A \; M \mid End$

3.3 Implementation

3.3.1 The Abstract Syntax Tree

The tree structure of SFL requires the following different types of tree nodes:

- Identifier
- Literal
- Application
- Abstraction

```rust
struct AST {
    vec: Vec<ASTNode>,
    root: usize,
}

enum ASTNodeType {
    Identifier,
    Literal,
    Application,
    Assignment,
    Abstraction,
    Module,
}

struct ASTNodeSyntaxInfo {...}

struct ASTNode {
    t: ASTNodeType,
    token: Option<Token>,
    children: Vec<usize>,
    line: usize,
    col: usize,
    type_assignment: Option<Type>,
    additional_syntax_information: ASTNodeSyntaxInfo
}

```

Figure 3.1: The Rust code listing for the definition of the AST, with lifetime specifiers, accessibility modifiers, and the syntax information (see 3.3.2) removed for conciseness.

- Assignment
- Module

[Sam: bullets look silly, if you have spare time draw a pictures, otherwise just say that it is a tree structure, listing the nodes adds nothing]

Initially, the approach taken when implementing this tree structure was to have each node 'owning' its child nodes (see 2.3). However, it will be frequently necessary to be able to find nodes based on certain conditions (for example, the condition that this node is a valid redex) and then provide a value that represents the location of this node within the tree. Even if each of the tree nodes had a unique ID, locating a node from this value representing its location will require some sort of tree search.

Rather than this solution, which would have a non-constant node lookup time, a secondary structure can be used to store the tree nodes with constant time lookup, and then each node can store a value enabling constant time lookup of its children within this structure. In the implementation, these types were labelled as Abstract Syntax Tree (AST) and ASTNode, where AST was an array of ASTNodes, and each ASTNode stored their children's indices in this array. The position in the array of an ASTNode will be referred to as its index.

See 3.1 for the code listing for the AST definition. In this implementation, `Vec` was used for the array, as it is growable, resizeable, and facilitates constant-time lookup of its elements. The AST stores and owns all of the nodes, as well as storing the index of the root node rather than requiring it to be at a specific index.

The node indices in the `children` vector represent different things depending on what kind of node it is.

- If it is an abstraction, the first node represents the variable (or pair of variables) abstracted over, and the second node represents the expression.
- If it is an application, the first node is the function, and the second is the argument.
- If it is a module, then each of the children is an assignment.
- If it is an assignment, then the first child is the variable being assigned to, and the second is the expression.

`Literal` and `Identifier` nodes store the tokens that defined them, so the strings can be accessed. `Identifier` nodes used as abstraction arguments. These types can either be specified in the source program, or inferred later. Nodes also store their positions (line and column) in the source program, which can be used for error messages.

In order to effectively explain the structure of a parsed program going forwards, the following structure will be used to give a written representation of an AST:

- Nodes are represented as one line each, where, with the name of the node type, followed by its value for `Literals` and `Identifiers`.
- The children of a node are all of the nodes with an indentation level one deeper than the node in question listed directly below it, until a shallower or equal depth node is listed.

For instance,

```
main = (\x.1) 2
```

would be represented as:

```
Module:
  Assignment:
    Identifier: main
    Application:
      Abstraction:
        Identifier: x
        Literal: 1
      Literal: 2
```

With the Benefit of Hindsight

[Sam: very nice evaluation, love this section] This project was my first major project using Rust. Below is a discussion of some Rust features which were not fully taken advantage of in this definition of syntax trees, followed by a discussion of the combination of these features that would have been more optimal.

Tagged Unions An alternative implementation could have involved `ASTNodeType` being a tagged union, with different node types being associated with different children and data items. For instance, application could be represented by `Application(f: usize, x: usize)`, and identifiers could be `Identifier(String)`. This would be more space efficient, as each node requires different data. It would also more elegantly represent the fact that each type of node is a different thing, and de-obfuscate the meaning of each of the different fields of a node.

References This definition of the AST [Sam: i dont understand the rest of the sentence] and the nodes has a parent object owning all of the nodes. As previously discussed, this was done to enable constant-time lookup of nodes from their indices. However, all things in a program already have such a reference enabling constant time lookup: a pointer, represented in rust by a reference. This was not used, as there were concerns about ensuring validity of each reference, and avoiding use-after-free bugs. These concerns were unfounded, as one of Rust's major features is that it provides safety guarantees ensuring that these problems are never encountered [16]. An object can only store a reference to another object if it can be guaranteed that it exists, and it will continue to exist for at least as long as the object storing the reference will. This is achieved via lifetime checking, using either inferred or explicitly stated specifiers of how long the two objects will exist relative to each other.

A Better Implementation [Sam: numbers alone look odd, especially at the beginning of a sentence. An easy solution is to always give references titles e.g. Figure bla or Table bla]3.2 shows an implementation that uses tagged unions to store information that is different for different node types, and pointers to the nodes directly rather than list indices. This avoids the possibility of referencing nodes that don't exist. It is also easier to understand what is common between nodes (syntax info) and what is uncommon. It is also more space efficient as it only stores the information that each type requires. The size of the improved implementation is 88 bytes, and the size of the original implementation is 128 bytes. The improved implementation is subjectively more elegant and readable. Objectively, it also takes up less space. It also forces memory safety, without the need for carefully implemented getter and setter functions.

Despite this, the decision was made not to update the implementation for several reasons. The AST is so central to the implementation, that it would take a long time to switch properly. Memory and speed are not major constraints for this project, but implementation time is. Furthermore, as long as all indices used are either produced by a helper function, or the AST root, there should not be a problem with memory safety.

```
struct AST<'a> {
    vec: Vec<ASTNode<'a>>,
    root: &'a ASTNode<'a>,
}

enum ASTNodeType<'a> {
    Identifier{name: String},
    Literal{value: String, _type: PrimitiveType},
    Assignment{to: String, expr: &'a ASTNode<'a>, type_assign: Type},
    Abstraction{var: String, expr: &'a ASTNode<'a>, type_assign: Type},
    Module{assigns: Vec<&'a ASTNode<'a>>},
}

struct ASTNodeSyntaxInfo { ... }

struct ASTNode<'a> {
    t: ASTNodeType<'a>,
    info: ASTNodeSyntaxInfo
}

```

Figure 3.2: An alternative implementation with a few advantages over the actual implementation.

3.3.2 Methods on the AST

Below are a selection of the more important or interesting methods implemented on the AST and its nodes. [Sam: i think I want to see the code as well]

Adding new nodes We will frequently want to add new nodes to the tree. Where they are inserted is not important, so the tree will add them to the end, and return their index. These methods are needed extensively for the parser.

Getting children of nodes As the interpretation of the `children` array for each node changes depending on what type of node it is, a series of getters are implemented, such as '`get_func`' to get the function of an application. These methods are needed extensively for the type checker, and the redex finding system.

Substitute variable Substitutes all instances of a variable in an expression with a given expression. This is needed for applying abstractions. For instance, the process of reducing `(\x.x) 1`, is:

- Get the name of the variable abstracted over: `x`.
- Replace all instances of x in the abstraction expression with the right hand side of the application: `1`.
- Replace all references to the abstraction with references to the abstractions expression.

Note that this orphans the node for the abstraction, and the node for the abstraction variable `x`. This is hard to rectify as deleting any nodes will shift the whole list, which would invalidate indies of nodes, which will break many of the references. This is rectified by cloning the AST, as described below.

Clone The AST, or just a subsection of the AST from a given node, can be cloned by starting from the desired new root, and cloning each nodes children recursively. The new indices of each node may not be the same, as they may be moved in the list, but they will all be in the same place relative to each other. This also removes orphaned nodes, as they will never be cloned as they have no parents.

To String Programs can also be effectively transformed back into strings. This requires a few other pieces of information to be associated with some tree nodes, to make the output program as similar to the input program as possible. The more similar the output is to the input, the easier it is to understand. Some examples include:

- Whether the application was generated by using the right associative `$` operator in order to avoid parenthesis, for instance `id $ 1 + 1`.

- Whether the assignment, where the expression is an abstraction, was generated using the syntax `x = \a.e` or the syntax `x a = e`.

We must also take into account whether a binary infix operator was used to generate a function call, and if so we must place it in the middle of its arguments.

3.3.3 The Parser

The parser needs to consume a program, and return the following things:

- The AST.
- The 'Label Table': The types of all labels defined, including both those defined explicitly (assignments) or implicitly (constructors). This is implemented as a struct '`LabelTable`' which is a wrapper around a `HashMap<String, Type>` with some useful methods.
- The 'Type Table': All type constructors and concrete types defined, stored with their arities. This is implemented as: `HashMap<String, usize>`.

For instance, from the program: [Sam: examples are good!]

```
data List a = Cons a (List a) | Nil
double x = x * 2
main :: List Int
main = Cons (double 1) (Cons (double 2) Nil)
```

We should extract the following data:

- The AST:

  ```
  Module:
    Assignent
      Identifier: double
      Abstraction
        Identifer: x
        App
          App
            Identifier: +
            Identifier: x
          Literal: 2
  ```

- All the known type assignments (excluding inbuilts)
  - Cons: $\forall a.a \Rightarrow List\ a \Rightarrow List\ a$
  - Nil: $\forall a.List\ a$
  - main: $\forall a.List\ a$
- The names of all known types (excluding inbuilts)
  - List, with an arity of 1

The parser will also store a set of all bound variables at each location. This will allow us to disqualify some invalid programs while generating the tree, rather than having to traverse it after generation to catch these issues. For instance, we must the following assignments:

- `x = (\x. e)` where e is a valid expression, as x is ambiguous during the expression e. This would be disqualified when attempting to parse the abstraction as x is already bound.
- `x = y` where y is undefined.

Lexical Analysis

Lexical analysis is the process splitting a program into its constituent tokens (Lexemes). For instance, the program `main = (\x.x) 1` is the following stream of tokens:

$$[Id : main, Assignment, LeftParen, Backslash, Id : x, Dot, Id : x, RightParen, Literal : 1]$$

See H for the code listing of the definition of the tokens output by the lexical analysis.

The lexer loads the entire string into memory at once. This is not typical, as this can lead to problems with large files. The approach discussed in [1] relies on a system of two buffers only holding individual pages of the file from disk. However, this system will not be loading files from disk; the program string is already in memory as it comes from the UI. Therefore, there would be no benefit to a more traditional lexer optimised to reduce memory usage.

The lexer provides a `next_token` function that returns the next token, and advances the pointer to the start of the token after. The lexer keeps track of line and column information, which is stored in the token to then be stored in the AST.

Expression Parsing

Expressions are parsed using recursive descent parsing. Some of the techniques used for this part of the parser were inspired by the discussion of top down parsing in [1].

At the top level, the expression parsing method is `parse_expression`. A variable `left` stores what is currently the index of the expression. It is called `left` as if we encounter a token that denotes that `left` is applied to whatever comes next, it becomes the left hand side of the application. `left` is originally set to be the output of parsing a primary (see 3.3.3), and then progresses differently based on the next token. Below are some of the ways that `parse_expression` could proceed.

- If the next token is an open bracket, we consume the token and then parse an expression. We then expect a closing bracket. We set `left` to the application of `left` to the expression
- If the next token is a dollar sign, we consume the token and then parse an expression. We do not expect a closing bracket, and we error if we receive one. We set `left` to the application of `left` to the expression.
- If the next token is a token denoting the start of a primary expression structure:
  - A backslash, indicating the start of a lambda
  - An identifier, indicating a variable.
  - A literal

  We parse a primary, and set `left` to the application of `left` to our primary.
- If the next token is:
  - A closing bracket
  - EOF
  - A newline
  - A double colon, indicating a type assignment follows

  We return `left`.

Primary Parsing A primary is a less complex structure than an expression. In this system, a primary is any expression structure other than applications. The primaries are:

- Literals
- Identifiers
- Lambdas
- Expressions in brackets

Each of these have their own specific parsing algorithms, which may include calling `parse_expression`.

Literal and Identifier Parsing Literals and identifiers are turned trivially into their respective AST Nodes. For instance, the token:

```
Token {
    line: 0,
    col: 0,
    tt: TokenType::IntLiteral
    value: "2"
}
```

Is turned into this ASTNode:

```
ASTNode {
    t: ASTNodeType::Literal,
    token: Some({the token}),
    children: [],
    line: 0,
    col: 0,
    type_assignment: Option<Primitve::Int>,
    additional_syntax_information: ...
}
```

To parse an identifier, we must also check that the identifier is bound at this location.

Parsing Abstractions Abstractions (in the simple case) are parsed by:

- Consuming a lambda (represented by '\' for ease of typing on standard keyboards)
- Parsing a variable. This variable must be added to our set of 'bound' variables.
- Consuming the dot separator'.'
- Parsing an expression
- Constructing an abstraction node from the variable and the expression

However, the definition of abstractions has a few complicating elements of syntax sugar.

Abstractions May be Assignments The assignment `f x = x` is implicitly `f = \x. x`. This is solved by parsing an argument to `parse_abstraction` representing whether this is an assignment. If it is an assignment, we do not parse the lambda, and expect the assignment operator '`=`' as our separator rather than the dot. As previously mentioned in 3.3.2, in order to output the string in a format that is as close as possible to the input, we set a flag in the `ASTSyntaxInfo`: `assign_abst_syntax` to all abstraction nodes defined like this.

Abstractions May Have Multiple Variables The abstraction `\x y. x` is syntax sugar for `\x. (\y. x)`. Additionally, with the assignment syntax, `f x y = x` is syntax sugar for `f = \x. (\y. x)`. This can be accounted for by continually parsing variables until we encounter '`.`' or the assignment operator '`=`', and then producing a series of abstractions over these variables in order.

3.3.4 Making Progress

Functional programs progress via reduction. λ-calculus expressions can reduce when we have an abstraction applied to a term. This is much the same in SFL, the difference being we can also apply labels representing abstractions to terms. In the following listing, expressions labelled as '`main1`' and '`main2`' both reduce to `5`.

```
f x = 5

main1 = f 10
main2 = (\x. 5) 10
```

Rather than simply reducing '`f 10`' to 5 in one step, it would be useful to show the substitution to help users to understand why '`f 10`' becomes '`5`' here. This would involve substituting the variable `f` for the function `(x. 5)`. We have two types of progress: reduction and substitution. In this section, we shall broaden the term 'redex' to include both.

Representing and Applying Redex-Contraction Pairs We can define a structure representing progress for our AST: a pair where the first element is the node in the tree to be replaced, and the second element is an entirely new tree to replace it with.

```
type RCPair = (usize, AST)
```

To do the substitution, we combine the two ASTs by concatenating their two lists of nodes, and replace all references to the original node to the root of the replacement AST.

Finding Redex-Contraction Pairs in a Term

Is the Term T a Redex? The core of the redex finding algorithm is being able to identify whether a term T with AST index P is a redex, and get its contraction. This is achieved by proceeding case wise on the shape of T.

- $T = (\backslash x.M)\,N$: We construct an `RCPair` where the left size is P, and the right side is the body of the abstraction M with all instances of x replaced with N.
- $T = f$: We construct an `RCPair` where the left size is P and the right-hand side is the value associated with the identifier `f` cloned into a new AST.
- Any other shape, T is not a redex.

Getting a List of Redexes in the Term T We first check if T is a valid redex, and if so add it to our list. If T is an application we extend this list with the lists of `RCPairs` in the function and argument of the application.

3.3.5 Web UI MVP

As part of this section, I also developed the MVP for the Web UI. 3.3 shows the web UI after this stage of the project.

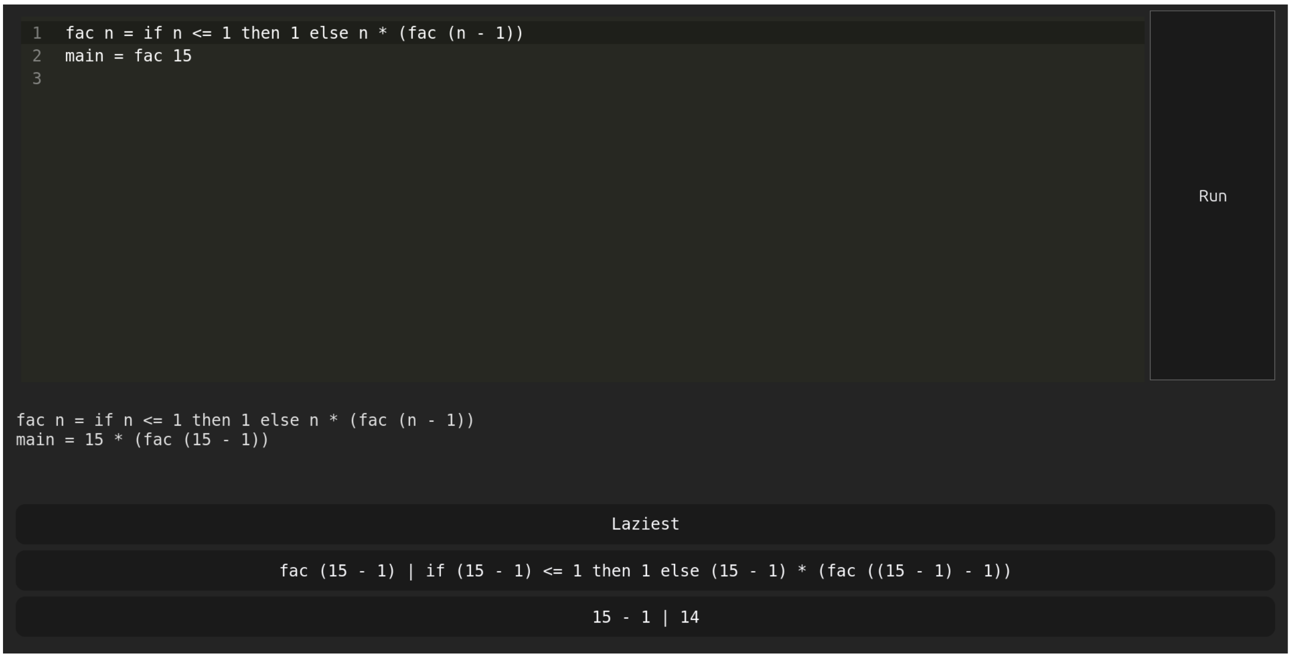


Figure 3.3: The Web UI MVP, as presented to my client at the end of phase 1.

Until this point, development was done in one rust package. This package would be compiled to a binary and run natively, with a basic CLI. This needed to be changed so that it can compile to Web ASseMbly (WASM) and run in the web browser. As I wanted to keep the CLI for debugging, as well as for use later, I did not want to change the whole project to a project with a WASM interface. A solution to keeping both interfaces was to separate the functionality that would be common to the CLI as well as the WASM library into a separate library, and then have the two interfaces as separate packages that depended on this one. The structure of the project became the following 4 packages.

- **libsfl**: All of the language functionality, as this is common to both interfaces
- **sflcli**: All of the original CLI functionality without the language functionality.
- **libsfl_wasm**: a package set up for use with wasm-pack (2.5). It provides a wrapper around **libsfl**, with wrapper functions returning data structures supported by wasm-bindgen**??**. wasm-pack would compile this to an Node Package Manager (NPM) package containing:
  - The WASM binary blob of the compiled rust code
  - A JavaScript file that would load the blob into the browsers' memory, and provides methods that can call the appropriate the methods in the blob
  - A TypeScript file providing the types of all of the packages exported functions.
- The Vite+React frontend (see 2.4) that requires the package that results from compiling **libsfl_wasm**

The WASM library provided functions that could be called from the JavaScript module. Rather than passing the AST around between the WASM library and the JavaScript module, the AST was stored in

[TODO: original approach was to use a set memory region for the AST as i did not wanna pass it to javascript]

[TODO: More stuff about WASM bindings??]

3.4 Proof of Concept Client Meeting: Evaluation and Next Steps

At the end of the phase, I presented the proof of concept project to my client, who was very positive about the project and its potential. The discussion was informal, a friendly conversation rather than a structured interview, to allow the direction of questioning to change depending on the clients answers. The meeting started with me giving my client a demo of the proof of concept using by using the system to evaluate the following program:

```
fac n = if n <= 1 then 1 else n * (fac (n - 1))
main = fac 5
```

Below is a summary of my clients thoughts about various aspects of the proof of concept system and potential future iterations

3.4.1 Usefulness as a Teaching Tool

Below are some notes on what the client thought about the effectiveness of the project as a teaching tool, and how it could be improved in future iterations.

- The project is already very useful as a teaching tool to demonstrate:
  - Evaluation order, and the importance of laziness
  - Currying
  - Recursion
  - The λ-calculus
- On top of this my client wanted to be able to use the system to demonstrate:
  - High Priority
    - List and common list functions such as 'map' and 'fold'. These do not have to by polymorphic, they could be just defined over Ints or some other type. These also do not need to be user-definable, they can be built in.
    - Pattern Matching
  - Lower Priority
    - User definable data types, preferably polymorphic. This would mean we could define '*List*' as part of the language which would be good for clarity.

3.4.2 The Existing Language

Positives:

- The language looked similar to Haskell. Particularly, the `if _ then _ else` syntax, and the function assignment shorthand syntax (`fac n = ...` rather than `fac = \n. ...`, even though these are identical)
- The language is minimal and clear
- The factorial function was quite elegant, and it would be understandable to people who did not know Haskell.

Negatives: My client had no specific complaints about the language as it currently stands, however we agreed was lacking many important features. The most difficult things to teach are concepts involving more complex data types.

Requested Features: Below are the specific features my client asked for in order for the system to be able to demonstrate the things she wants to use the system to demonstrate:

- Recursive Types
- Polymorphism
- Type Aliases
- Typechecking
- User Definable Data Types

3.4.3 The Existing UI/UX

Positives:

- The editor, as it feels like a very popular editor: VSCode

Negatives

- It is unstable. This is bad in a teaching tool, as it would waste a lot of time if it constantly broke in the lecture.
- 'laziest' as an option is confusing, as it was unclear if it was referring to one of the other on screen options, or if it was referencing a 'hidden' option
- The vertical bar separating redex from contraction on the progress buttons was not obvious enough. On top of this, the bar was not centred, so it was hard to look through all the redexes at once as they were not aligned with each other.

Requested Features

- Syntax highlighting, to make the language easier to read
- A history of what the expression has been is vital to demonstrate step by step evaluation. I identified this as an important feature at the beginning of the phase (see 3.1.2), but I had not finished it by the client meeting. It was implemented in the next phase (see 4.3.8).
- Sample Programs

3.4.4 Conclusion

At the end of this phase, and going into phase 2, I had a strong proof of concept system and an idea for how the system will look. The meeting with my client yielded many ideas, all of which I sucessfully implemented throughout this project.

Chapter 4

Phase 2 — Types and Pattern Matching

In this phase, I moved away from the autoethnographic (3.1.1) approach, where most of my requirements came from within, to an externally motivated client-led approach.

At the end of this phase, I held a focus group (4.5) to help me evaluate the progress of the project. Because this was the plan from the beginning of the phase,

4.1 Requirements Analysis

The requirements for this phase were motivated by my client meeting (??). I wanted to tackle the most technical aspects in this phase to give me the maximum time to complete them, as this was still early in the project lifecycle. The most difficult features out of the client's requests were the ones to do with extending the language, so these were the main focus for this phase.

The client's central idea for what they wanted to use the tool was to demonstrate methods on lists, such as 'map' and 'foldr/l'. This requires lists to be built into the language. Lists in functional programming languages are commonly defined recursively, using `Cons x xs` to represent constructing a list from an element `x` and the rest of the list `xs`. `Nil` represents an empty list. This recursive construction of lists comes from Lisp [18]. Similarly, in Haskell, lists are defined as `data [a] = [] | a : [a]`

[TODO: finish yapping about lists and talk about why that means we need ADTs]

This definition of lists is as example of a polymorphic data type. It also implicitly defines two polymorphic constructors, '[]' also known as Nil which has type $\forall a.[a]$, and ':' also known as Cons which has a type $\forall a.a \to [a] \to [a]$.

4.2 Design

4.2.1 Language Changes

The focus of this project phase is mainly to upgrade the language SFL. We have already identified what features we would like to add the language. This section will go into detail about the design for the extension for the language enabling these new features.

Type System

If we are to effectively represent the type of expression containing integers and booleans, we must have types *Int* and *Bool*. We also want our type system to be able to express functions, as our language support functions. We also want polymorphism in our type system, as rewriting functions many times for different data types makes programs more verbose.

Allowing for algebraic user defined data types similarly to Haskell would make the language much more expressive and much more powerful, as well as bringing it closer to Haskell. Supporting tagged unions and tuples in the SFL type system would massively increase the ease of writing complex programs. It would also allow for complex data structures such as trees and lists.

Type names, as well as constructor names, start with uppercase letters in Haskell. This allows them to be easily differentiated from type variables, as well as regular variables.

$$\text{Types} \quad A, B, C \quad ::= \quad \text{Int} \mid \text{Bool} \mid \alpha \mid \forall\alpha.\ A \mid A \rightarrow B \mid (A, B) \mid \text{Name}[A_1, \ldots, A_n]$$

Figure 4.1: The SFL type system

First-order polymorphic type constructors would be useful to have in SFL, with one example of their utility being defining the polymorphic function '`length :: List a -> Int`' which should work regardless of what type the list is over.

Figure 4.6 shows the type system in SFL. Note that the definition of type constructors here is more permissive than is correct, as it does not enforce that we apply our type c

User Definable Algebraic Data Types

In Haskell, we can create algebraic types using the `data` keyword (see 2.2.3). Replicating this syntax for SFL's user defined data types is desirable because it would allow people already familiar with Haskell to use the system, as well as viva versa.

As an example, the SFL (and Haskell) data declaration:

```
data Either a b = Left a | Right b
```

creates a tagged union type called *Either* with two constituent type parameters a and b. In our type system (4.2.1) this would be represented as $Either[a, b]$. The Name `Either` uniquely identifies this type, this must be enforced by the parser. It also creates two data constructors: `Left` which has the type $\forall a\ b.a \rightarrow Either[a, b]$, and `Right` which has the type $\forall a\ b.b \rightarrow Either[a, b]$.

Type aliases allow us to make code more readable and expressive. For instance, if we were to define playing cards like this:

```
data Suit = Hearts | Clubs | Spades | Diamonds
data Rank = Num Int | Jack | Queen | King | Ace
type Card = (Suit, Rank)
```

having the type alias `Card` for `(Suit, Rank)` allows us to very easily, and more readably, create functions on Cards, as well as values with that type.

To summarize, we will implement type aliases and algebraic data types to work similarly to Haskell with similar syntax.

Match

[Sam: one sentence reminder to reader of what patternmatching is: "Pattern matching is an elegant form of branching used in many functional languages." (this just sets the scene better)] See 2.2.5 for more information about Haskell pattern matching. A basic example of pattern matching in Haskell: [Sam: the preceeding sentence sounds very weird because it has no subject. You either want to say "fac is a basic example" or "here is a basic example"]

```
fac :: Int -> Int
fac 0 = 1
fac n = n * factorial (n - 1)
```

Here, the definition of the 'fac' function is different depending on if is it applied to 0 or to any other *Int*. If it is applied to an *Int* other than 0, n is substituted for this value in the expression.

Pattern matching at the top level like this would be difficult to implement, as it would require significantly changing how abstractions are represented. It would be easier to create a new syntax structure: a match expression. This could look like:

```
fac :: Int -> Int
fac n = match n {
    | 0 -> 1
    | _ -> n * (fac (n - 1))
}
```

This syntax was fairly arbitrary, as syntax is quite easy to change. However, this syntax proved to be fairly popular with all three focus groups, so it did not change between this stage and the end of the project. [Sam: the last code sandwich is your best yet! More lik this please! It is clear and to the point]

[Sam: no need to repeat yourself. Just say something like this does the same as the Haskell fac, just with different syntax. Here you are not explaining the code, but the difference between this code and the previous version]The 'fac' function takes an *Int* n, and proceeds differently with different values of n. If the value is 0, the value of the whole expression becomes 0, otherwise it becomes `n * (fac (n - 1))`. We can use literals in our pattern to differentiate between different values of literals. Inspired by Haskell, we can use a variable (which is a lowercase identifier) to match anything, a 'wildcard' pattern. All instances of the variable in the pattern's corresponding expression with the term that the variable matches. '_' is a special case wildcard, where no variable is bound, but it still matches anything,

We should also be able to match more complex structures including Algebraic Data Types. For instance, we can write the following function to figure out whether a list has length 2 or grater

```
lenIsAtLeastTwo :: List a -> Bool
lenIsAtLeastTwo list = match list {
    | Cons _ (Cons _ _) -> true
    | _ -> false
}
```

In this example, it is important that we evaluate the term 'list' enough to *know for sure* that it does not match the first pattern before moving on to the second, as the second pattern is irrefutable.

4.2.2 Next UI Iteration

At this phase of the project, the current version of the web UI is a proof of concept. See 3.3 for the current state. The UI requires a total redesign [Sam: end of sentence? why does it need a redesign?]

I completely redesigned the UI based on the clients' feedback, as well as based on other requiremenets identified during the autoethnographic phase of the project. See 4.2 and 4.3 for screenshots of the new design. [Sam: talk about these designs! This section is very thin, you just say heres what im aiming for, totally chatted to focus group, but the reader will be interested in why you went this way and what the focus group said. This is more reason I suggested at the top to tackle each change one by one cos you're always leaving the reader hanging and pointing else where in the document, but it is not dramatic suspense, I'm just opening more tabs in my brain than I can cope with]These designs were done using Figma.

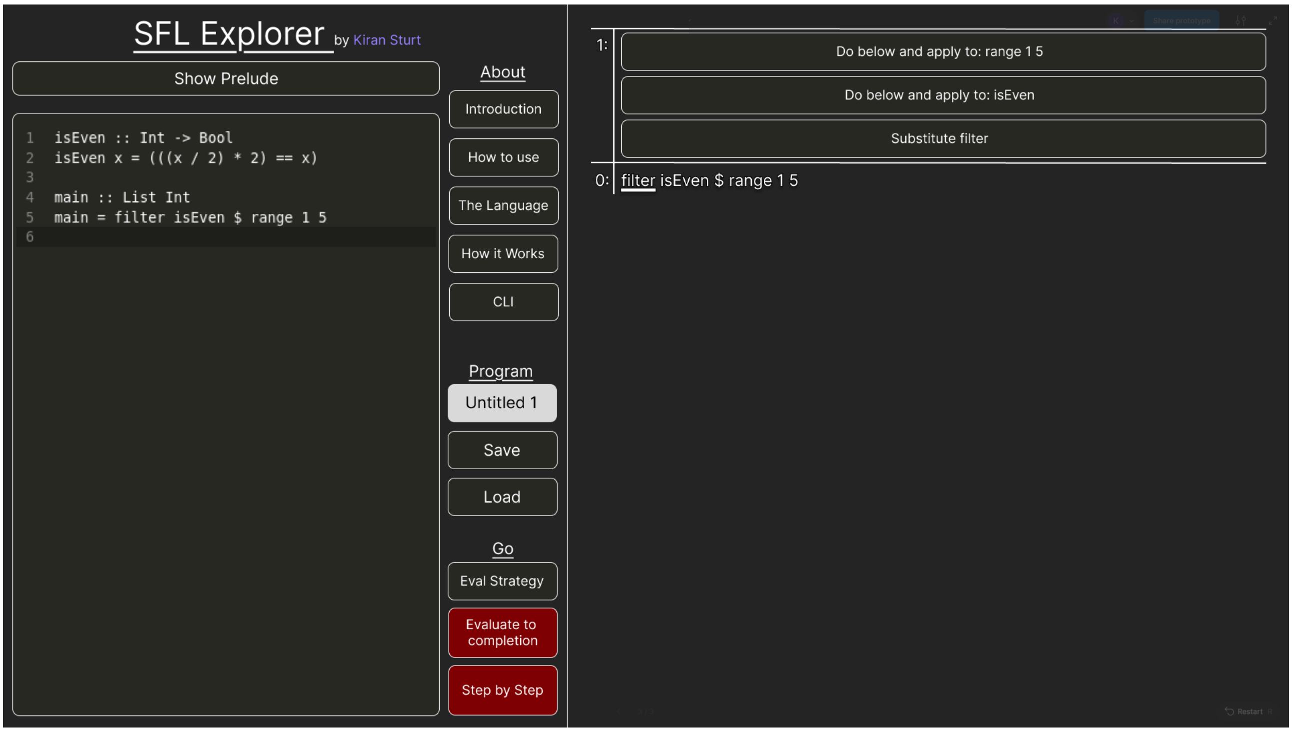


Figure 4.2: Screenshot 1 of the Figma design of the web UI[Sam: formatting bug bear: put full stops at the end of captions]

This design was meant to be a work in progress, but it looks quite similar to the final release of the product (Screenshots I.5, I.6, I.7 and I.8). Before implementing this design, I discussed this design with the Advanced Focus Group (see 4.5) and they were much more positive about this UI than the existing one

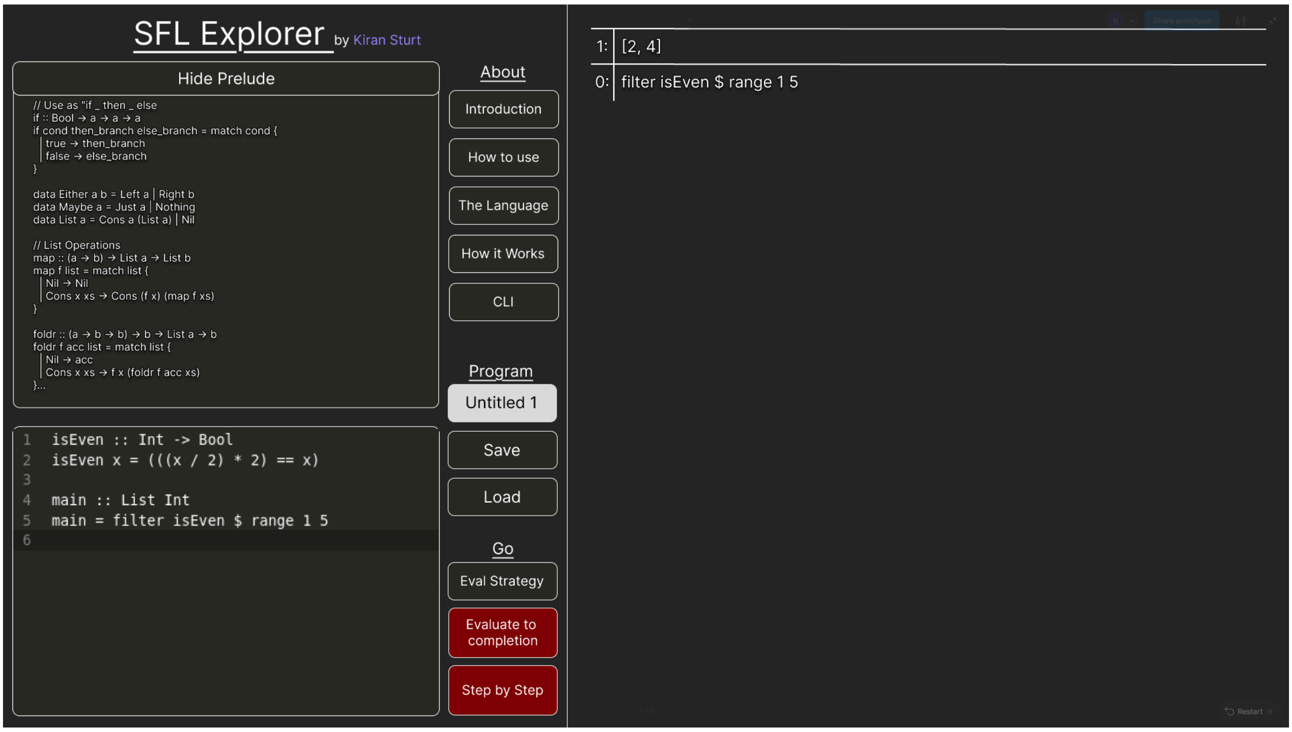


Figure 4.3: Screenshot 2 of the Figma design of the web UI. This version shows the prelude extended [Sam: here is motivation as to why, it looks like somethign following extended has been cut off]

[TODO: Discuss revert]

[TODO: Design principle: simplicity, speed, minimalism, feeling like vscode.]

4.3 Implementation

4.3.1 Types

Rust allows us to represent our types (see 4.6 for the definition of the type system) quite easily using Enums. Rust's Enums are an example of algebraic data types, and are therefore very useful for defining our own algebraic data type system. See 4.4 for the listing.

We must use `Box<Type>`, which represents a pointer to a heap allocated object, otherwise it would be impossible to calculate the size of `Type`, as it could be infinitely large with it containing another `Type` recursively. `Box<Type>` however has known size: the size of a pointer in the target architecture.

We also define Existential, as an implementation detail needed for the type checker.

Aliases are defined here with their name, and the type they are an alias for. Aliases could simply be implemented by replacing all occurrences of the string on the left with the string on the right, but defining them here allows us to use their names to generate type errors making them easier to understand.

Methods on Types

Below are a selection of the more important or interesting methods implemented on Types. [Sam: didn't we have similar before?]

Substitution of type variables [Sam: it is annoying that para has the same formatting as subsubsection, if you have spare time, id change that as it makes reading harder]We may wish to set a type variable to another type. For instance, if given the type expression $T\ U$ where T and U are types, and we know that one of the constructors of T is of generic type $\forall a.a \to T\ a$, the type of the constructor for this type should be $U \to T\ U$. We have 'instantiated' the type variable a to be U by substituting a with U throughout the expression, and removing the $\forall a$. This is required for the type checker.

To String We will frequently wish to display types as strings for debugging purposes.

```rust
pub enum Primitive {
    Int64,
    Bool,
}

pub enum Type {
    Unit,
    Primitive(Primitive),
    Function(Box<Type>, Box<Type>),
    TypeVariable(String),
    Forall(String, Box<Type>),
    Product(Box<Type>, Box<Type>),
    Union(String, Vec<Type>),

    Alias(String, Box<Type>),
    Existential(usize),
}
```

Figure 4.4: The Rust code listing for the definition of types. 'Existential' and 'Alias' are separated as they are more of an implementation detail than a part of the type system

4.3.2 Type Assignment and Definition Parsing

We must also be able to parse type assignments (`x :: Int`) and type definitions (`type a`, or `data a`). Parsing both of these things require the ability to parse type expressions.

Parsing Type Expressions

Type expressions can be parsed using recursive descent parsing. Our connective for our parsing is arrow: '`->`', and our primaries are anything that does not contain arrows. The structure of the type expression parser is similar to that of the expression parser. We first parse a primary, and then we terminate or parse our connective. The algorithm for parsing a primary is as follows:

Parsing Type Expression Primaries

```
function parse_type_expression_primary(bound_type_variables, type_table) {
    next_token = consume()
    match (next_token) {
        case lowercase_id {
            name = next_token.value
            if bound_type_variables exists {
                if !bound_type_variables.contains(name) {Error}
            }
            return TypeVariable(name)
        }
        case uppercase_id {
            name = next_token.value
            if type_table.contains(name) {
                return type_table.get(name)
            } else {Error}
        }
        case left_paren {
            return parse_type_expression(bound_type_variables, type_table)
        }
        otherwise Error
    }
}
```

Parsing Type Alias Definitions

We may wish to add another name that a type can be known by. For instance, we may wish to define '`String`' as a list of characters, so that we may reference it easier. A type alias declaration consists of:

- The '`type`' keyword
- The name of the alias
- The assignment operator (=)
- The type expression

The function that produces type aliases returns a '`Type::Alias(String, Box<Type>)`'. The reason we do not want to simply rename all references to the alias name to the type in question is that this would make type errors more obscure, and harder for users to understand the error with reference to the original program.

Parsing Data Declaration

We want to be able to define and parse `data` declarations 4.2.1. A `data` deceleration consists of:

- The '`data`' keyword
- The name of the type (uppercase ID)
- A list of type variables
- The assignment operator (=)
- A set of constructors separated by |. Constructor definitions consist of the following.
  - The name of the constructor
  - Zero or more type expressions, representing what types the constructor can be applied to. These type expressions must only contain type variables in the set before the assignment operator, but can be any other concrete type expression.

An example definition is: '`data Either a b = Left a | Right b`'. The information that should be extracted from here is:

- '`Either`' is a type constructor with a kind of $* \to * \to *$. As we have no higher kinded types, this can simply be stored as a number representing the arity of the type constructor. In this case, the arity is 2.
- The constructors and their types are:
  - '`Left`': $\forall a\ b.a \to Either\ a\ b$
  - '`Nil`': $\forall a\ b.b \to Either\ a\ b$

We must store the type name and its arity in the 'Type Table', and all constructors in the 'Label Table'.

In order to parse the type constructor definition, we continually expect a lowercase identifier token until we reach the assignment operator '`=`'. The lowercase identifiers declared are passed to the functions that parse constructors so that we can enforce that all of the type variables used in the constructor parsing are 'in scope'. We also do this to make sure that the variables are correct, and in the correct order in the constructor definitions.

Parsing constructors is not complex, as we have already implemented the mechanism for parsing type expressions. We simply keep parsing expressions until either the constructor separator (|) or a newline is reached. When parsing the type expression, we expect only valid concrete types in the Type Table, or valid in scope type variables from the type constructor definition. We then produce a type from the types of the arguments in order, so '`ConstructorName expr1 expr2 expr3`' results in the type '$expr1 \to expr2 \to expr3 \to TypeName$', with all of the free type variables lifted to the start of the type into a series of nested '`Type::Forall`'s.

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against input type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes output type A, with output context Δ

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C \dashv \Delta}$ Under input context Γ, applying a function of type A to e synthesizes type C, with output context Δ

Figure 4.5: Checking and Synthesis judgements the typechecking algorithm attempts to derive. These are from Dunfield and Krishnaswami [8], and are not modified in my extension of their algorithm.

4.3.3 The Type Checker

The type checker uses an algorithm which is a modified version of the algorithm proposed by Dunfield and Krishnaswami [8].

> 'Bidirectional typechecking, in which terms either synthesize a type or are checked against a known type, has become popular for its scalability … its error reporting, and its relative ease of implementation' [8]

It was the 'relative ease of implementation' that attracted me to bidirectional type checking. I modified their algorithm to add my extra types (the inbuilt types *Int*, *Bool*, as well as the $\bigcup$ and $\times$ types) and my extra expression syntax structures (literals, match, pairs). This does not include assignment and modules as these are not part of expression syntax.

$$\begin{array}{llll}
\text{Types} & A, B, C & ::= & \text{Int} \mid \text{Bool} \mid \alpha \mid \hat{\alpha} \mid \forall \alpha.\, A \mid A \to B \mid (A, B) \mid \text{Name}[A_1, \dots, A_n] \\
\text{Monotypes} & \tau, \sigma & ::= & \text{Int} \mid \text{Bool} \mid \alpha \mid \hat{\alpha} \mid \tau \to \sigma \mid (\tau, \sigma) \mid \text{Name}[\tau_1, \dots, \tau_n] \\
\text{Contexts} & \Gamma, \Delta, \Theta & ::= & \cdot \mid \Gamma, \alpha \mid \Gamma, x : A \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}}
\end{array}$$

Figure 4.6: Syntax of types, monotypes, and contexts as seen by the typechecker. The definition of types differ slightly from the definition offered in figure 4.1, as we include existential type variables ($\hat{\alpha}$) that can not actually be created by users. They are an implementation detail required for the type checking algorithm.

$$\begin{array}{ll}
[\Gamma]\text{Int} & = \text{Int} \\
[\Gamma]\text{Bool} & = \text{Bool} \\
[\Gamma]\alpha & = \alpha \\
[\Gamma[\hat{\alpha} = \tau]]\hat{\alpha} & = [\Gamma[\hat{\alpha} = \tau]]\tau \\
[\Gamma[\hat{\alpha}]]\hat{\alpha} & = \hat{\alpha} \\
[\Gamma](A \to B) & = ([\Gamma]A) \to ([\Gamma]B) \\
[\Gamma](\forall \alpha.\, A) & = \forall \alpha.\, [\Gamma]A \\
[\Gamma](A, B) & = ([\Gamma]A, [\Gamma]B) \\
[\Gamma]\text{Name}[A_1, \dots, A_n] & = \text{Name}[[\Gamma]A_1, \dots, [\Gamma]A_n]
\end{array}$$

Figure 4.7: Applying a context, as a substitution, to a type.

The typechecking algorithm (Figure E.1) is presented as a series of *inference rules*, which describe how to derive conclusions (written below the horizontal line) from a set of premises (written above it). These rules allow us to construct *derivation trees* by starting from the conclusion we wish to prove and recursively deriving each premise, continuing until we reach axioms—rules with no premises. The conclusions derived in this way are called *typing judgements*. A typical typing judgement takes the form $\Gamma \vdash x : A$, meaning 'under context Γ, it is derivable that x has type A'. However, in our bidirectional type system, we have three different types of typing judgements we attempt to derive, and in the process of deriving these judgements we also update our context with additional information gleaned during the derivation process 4.5.

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \textsf{Sub}$$

Figure 4.8: One of the inferrence rules in the algorithm, listed here as an example.

Figure 4.8 shows one of the most important rules in the algorithm as an example. This rule Sub allows us to derive that $\Gamma \vdash e \Leftarrow B \dashv \Delta$ (Under imput context Γ, e checks against type A with output context Δ)

Hole notation Below is a paragraph from the original paper [8] explaining 'hole notation' in contexts, which is used throughout the algorithm descriptions in E.1, E.2 and E.3.

> 'Since we will manipulate contexts not only by appending declarations ...but by inserting and replacing declarations in the middle, a notation for contexts with a hole is useful:
>
> $$\text{`}\Gamma = \Gamma_0[\Theta] \quad \text{means } \Gamma \text{ has the form } (\Gamma_L, \Theta, \Gamma_R)$$
>
> For example, if $\Gamma = \Gamma_0[\hat{\beta}] = (\hat{\alpha}, \hat{\beta}, x : \hat{\beta})$, then $\Gamma_0[\hat{\beta} = \hat{\alpha}] = (\hat{\alpha}, \hat{\beta} = \hat{\alpha}, x : \hat{\beta})$
> ...
> Occasionally, we also need contexts with *two* ordered holes:
>
> $$\Gamma = \Gamma_0[\Theta_1][\Theta_2] \quad \text{means } \Gamma \text{ has the form } (\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R)\text{'}$$ [8]

Algorithmic Type Checking and Synthesis

Figure E.1 shows the main algorithm for checking and synthesizing the types of various expression structures. Some of the inferrence rules listed The rule I added are:

- Checking and synthesizing rules for Ints and $Bool$s: An int literal synthesizes the type Int, and checks against the type Int etc.
- Checking and synthesis rules for pairs
  - A pair (e_1, e_2) checks against type (A, B) if e_1 checks against A and e_2 checks against B.
  - If e_1 synthesizes type A and e_2 synthesizes type B then (e_1, e_2) synthesises (A, B)
- The rule for synthesizing the type of a match expression. We synthesize the type of the expression being matched to a type A ($\Gamma \vdash e \Rightarrow A \dashv \Delta$). We then check all the types of the patterns against type A. To get the output type, we add an existential type variable $\hat{\alpha}$ to our context, and then check the type of all the output expressions against the type $\hat{\alpha}$. We synthesize the type $\hat{\alpha}$. The context is passed through all of these operations, and the final context contains all of the things 'learned' from typechecking the match expression.

Note that the checking rules IntLit$\Leftarrow$, BoolLit$\Leftarrow$, Pair$\Leftarrow$ are not actually necessary, as they could be caught by the Sub rule. They are included as they remove the unnecessary steps that using the Sub rule in this manner creates, speeding up/simplifying the algorithm.

Algorithmic Subtyping

Figure E.2 shows the algorithm for how we verify that a type is a subtype of another type. For instance, our typechecking rule Sub synthesizes the type, and the uses the algorithmic subtyping rules to check that the synthesized type is a subtype of the expected type. I have added rules for:

- Int <: Int and $Bool$ <: $Bool$ trivially.
- (A_1, B_1) <: (A_2, B_2) if A_1 <: A_2 and B_1 <: B_2.
- The union with name N_1 and arguments $A_1 \ldots A_n$ is a subtype of a union with name N_2 and arguments $B_1 \ldots B_n$ is a subtype, if the names are the same as the names uniquely identify these types, as well as all the constituent types being subtypes of each other. The context is passed through all of these operations, and the final context contains all of the things 'learned' from typechecking the match expression.

Context Instantiation

Figure E.3 shows the special subtyping rules for existential type variables that also instantiates the variable within the context. The subtyping rule <:InstantiateL allows us to derive that the existential type variable $\hat{\alpha}$ is a subtype of the type A. It requires that we instantiate $\hat{\alpha}$ to the value of A in our context. <:InstantiateR does the opposite. Both add to the context all the information we gain about $\hat{\alpha}$ by saying that it is a subtype of A, or that A is a subtype of it.

For instance, the rule InstLArr instantiates $\hat{\alpha}$ such that $\hat{\alpha}$ <: $A_1 \to A_2$ by adding to our context $\hat{\alpha}_1, \hat{\alpha}_2$, and setting $\hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2$ in the context. We then instantiate $\hat{\alpha}_1 :\leq A_1$ and $\hat{\alpha}_2 \leq: A_2$.

$$T_{Nil} = List\ Int \qquad\qquad T_{Cons} = Int \to List\ Int \to List\ Int$$

$$\Gamma = T_{Cons}, T_{Nil} \qquad\qquad \Gamma = \Gamma_0 = \Gamma_1$$

$$\dfrac{\dfrac{}{\Gamma_0 \vdash \mathrm{Int} <: \mathrm{Int} \dashv \Gamma_1}\ {<:}\mathsf{Int}\ [11]}{\Gamma_0 \vdash List[Int] <: List[Int] \dashv \Gamma_1}\ {<:}\bigcup\ [10]$$

$$\dfrac{\dfrac{\dfrac{(Nil : T_{Nil}) \in \Gamma}{\Gamma \vdash Nil \Rightarrow T_{Nil} \dashv \Gamma}\ \mathsf{Var}\ [9] \qquad [10]\Gamma \vdash List[Int] <: List[Int] \dashv \Gamma}{\Gamma \vdash Nil \Leftarrow List\ Int \dashv \Gamma}\ \mathsf{Sub}\ [8]}{\Gamma \vdash List\ Int \to List\ Int \bullet Nil \Rrightarrow List\ Int \dashv \Gamma}\ {\to}\mathsf{App}\ [7]$$

$$\dfrac{\dfrac{(Cons : T_{Cons}) \in \Gamma}{\Gamma \vdash Cons \Rightarrow T_{Cons} \dashv \Gamma}\ \mathsf{Var}\ [4] \qquad \dfrac{\dfrac{}{\Gamma \vdash 1 \Leftarrow \mathrm{Int} \dashv \Gamma}\ \mathsf{IntLit}{\Leftarrow}\ [6]}{\Gamma \vdash T_{Cons} \bullet 1 \Rrightarrow List\ Int \to List\ Int \dashv \Gamma}\ {\to}\mathsf{App}\ [5]}{\Gamma \vdash Cons\, 1 \Rightarrow List\ Int \to List\ Int \dashv \Gamma}\ {\to}\mathsf{E}\ [3]$$

$$\dfrac{[3]\Gamma \vdash Cons\, 1 \Rightarrow List\ Int \to List\ Int \dashv \Gamma \qquad [7]\Gamma \vdash List\ Int \to List\ Int \bullet Nil \Rrightarrow List\ Int \dashv \Gamma}{\Gamma \vdash Cons\, 1\, Nil \Rightarrow List\ Int \dashv \Gamma}\ {\to}\mathsf{E}\ [2]$$

$$\dfrac{[2]\Gamma \vdash Cons\, 1\, Nil \Rightarrow List\ Int \dashv \Gamma \qquad [10]\Gamma \vdash List[Int] <: List[Int] \dashv \Gamma}{\Gamma \vdash Cons\ 1\ Nil \Leftarrow List\ Int \dashv \Gamma}\ \mathsf{Sub}\ [1]$$

Figure 4.9: An example derivation showing how the type derivation $. \vdash Cons\ 1\ Nil \Leftarrow List\ Int \dashv .$

Examples Derivations

Below are two example derivations that cover most of the key aspects of the algorithm. The first example aims to give a simpler demonstration and explaination of how the typechecking and synthesis inference rules work, and use some of my rules. The second aims to show a more complex derivation heavily relying on the context instantiation rules. A supplamentory derivation is included in the appendices (D), which shows more of my rules in action.

Typechecking an Expression Involving Lists We shall attempt to use the algorithm to check the type of Cons 1 x against *List Int*. In this derivation, we defined `data List = Cons Int (List Int) | Nil` to avoid the complexities of polymorphism, which are better demonstrated by the next example 4.3.3. The reason the context is never changed is as we do not have any abstractions or foralls, so no variables or type variables are introduced. Type checking/synthesis is done recursively from bottom up, so read [1] upwards.

1. To check *Cons* 1 *Nil* against *List Int* [2], we first synthesise the expression type, and check the synthesised type is as subtype of *List Int* [10].
2. To synthesise the type of the expression *Cons* 1 *Nil* (implicitly ((*Cons* 1) *Nil*)) we synthesise the type of the left hand side of the application *Cons* 1 to be $List\ Int \to List\ Int$ [7] and then we synthesise the type of *Nil* under the application of that type, which gives us *List Int*.
3. To synthesise the type of *Cons* 1, we synthesise the type of the left hand side of the application *Cons* to be $T_{Cons} : Int \to List\ Int \to List\ Int$ [4], and then synthesise the type of 1 under the application of that type, which gives us $List\ Int \to List\ Int$ [5].
4. We synthesise the type of *Cons* to be T_{Cons} from the context.
5. We synthesise the type of 1 under the application of T_{Cons}, by first checking the type of 1 against the type *Int* which is the left hand side of the applied type[6]. This allows us to synthesise the right hand side of the applied type: $List\ Int \to List\ Int$.
6. 1 checks against the type *Int*, as it is an Int literal.

$$T_{Just} = \forall \alpha.\alpha \to Maybe\,\alpha \qquad\qquad T_{Nothing} = \forall \alpha.Maybe\,\alpha$$

$$\Gamma_0 = Just : T_{Just},\ Nothing : T_{Nothing} \qquad \Gamma_1 = \Gamma_0, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \qquad \Gamma_2 = \Gamma_1, \hat{\gamma}$$

$$\Gamma_3 = \Gamma_1, \hat{\gamma} = \hat{\alpha} \qquad\qquad \Gamma_4 = \Gamma_0, \hat{\alpha}, \hat{\beta} = Maybe\,\hat{\alpha}, x : \hat{\alpha}, \hat{\gamma} = \hat{\alpha}$$

$$\dfrac{\Gamma_3 \vdash Mabye\,\hat{\alpha}}{\{\Gamma_0, \hat{\alpha}\}, \hat{\beta}, \{x : \hat{\alpha}, \hat{\gamma} = \hat{\alpha}\} \vdash Mabye\,\hat{\alpha} \mathrel{\leqq:} \hat{\beta} \dashv \{\Gamma_0, \hat{\alpha}\}, \hat{\beta} = Mabye\,\hat{\alpha}, \{x : \hat{\alpha}, \hat{\gamma} = \hat{\alpha}\}}\ \textsf{InstRSolve}\ [12]$$

$$\dfrac{\hat{\beta} \notin \mathrm{FV}(Mabye\,\hat{\alpha}) \qquad \Gamma_3 \vdash Mabye\,\hat{\alpha} \mathrel{\leqq:} \hat{\beta} \dashv \Gamma_4[12]}{\Gamma_3 \vdash Mabye\,\hat{\alpha} \mathrel{<:} \hat{\beta} \dashv \Gamma_4}\ \mathrel{<:}\textsf{InstantiateR}\ [11]$$

$$\dfrac{\hat{\alpha} \notin \mathrm{FV}(\hat{\gamma}) \qquad \dfrac{}{\Gamma_2[\hat{\alpha}][\hat{\gamma}] \vdash \hat{\alpha} \mathrel{:\leqq} \hat{\gamma} \dashv \Gamma_2[\hat{\alpha}][\hat{\gamma} = \hat{\alpha}]}\ \textsf{InstLReach}\ [10]}{\Gamma_2 \vdash \hat{\alpha} \mathrel{<:} \hat{\gamma} \dashv \Gamma_3}\ \mathrel{<:}\textsf{InstantiateL}\ [9]$$

$$\dfrac{\dfrac{\dfrac{(x : \hat{\alpha}) \in \Gamma_2}{\Gamma_2 \vdash x \Rightarrow \hat{\alpha} \dashv \Gamma_2}\ \textsf{Var}\ [8] \qquad \Gamma_2 \vdash [\Gamma_2]\hat{\alpha} \mathrel{<:} [\Gamma_2]\hat{\gamma} \dashv \Gamma_3\ \ [9]}{\Gamma_2 \vdash x \Leftarrow \hat{\gamma} \dashv \Gamma_3}\ \textsf{Sub}\ [7]}{\Gamma_2 \vdash \hat{\gamma} \to Mabye\,\hat{\gamma} \bullet x \Rrightarrow Mabye\,\hat{\gamma} \dashv \Gamma_3}\ \to\!\textsf{App}\ [6]$$

$$\dfrac{\dfrac{(Just : T_{Just}) \in \Gamma_1}{\Gamma_1 \vdash Just \Rightarrow T_{Just} \dashv \Gamma_1}\ \textsf{Var}\ [4] \qquad \dfrac{\Gamma_1, \hat{\gamma} \vdash [\hat{\gamma}/\alpha](\alpha \to Mabye\,\alpha) \bullet x \Rrightarrow Mabye\,\hat{\gamma} \dashv \Gamma_3[6]}{\Gamma_1 \vdash \forall\alpha.\ \alpha \to Mabye\,\alpha \bullet x \Rrightarrow Mabye\,\hat{\gamma} \dashv \Gamma_3}\ \forall\textsf{App}\ [5]}{\Gamma_1 \vdash Just\ x \Rightarrow Mabye\,\hat{\gamma} \dashv \Gamma_3}\ \to\!\textsf{E}\ [3]$$

$$\dfrac{\dfrac{\Gamma_1 \vdash Just\ x \Rightarrow Mabye\,\hat{\gamma} \dashv \Gamma_3[3] \qquad \Gamma_3 \vdash [\Gamma_3]Mabye\,\hat{\gamma} \mathrel{<:} [\Gamma_3]\hat{\beta} \dashv \Gamma_4[11]}{\Gamma_0, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash Just\ x \Leftarrow \hat{\beta} \dashv \{\Gamma_0, \hat{\alpha}, \hat{\beta} = Mabye\,\hat{\alpha}\}, x : \hat{\alpha}, \{\hat{\gamma} = \hat{\alpha}\}}\ \textsf{Sub}\ [2]}{\Gamma_0 \vdash \lambda x.\ Just\ x \Rightarrow \hat{\alpha} \to \hat{\beta} \dashv \Gamma_0, \hat{\alpha}, \hat{\beta} = Mabye\,\hat{\alpha}}\ \to\!\textsf{I}\!\Rightarrow\ [1]$$

Figure 4.10: An example derivation showing how the type of $\lambda x.\ Just\ x$ can be synthesized

7. To synthesise the type of Nil under the application of $List\ Int \to List\ Int$, we check Nil against the type $List\ Int$ which is the left hand side of the applied type[8]. This allows us to synthesise the right hand side of the applied type: $List\ Int$
8. To check Nil against the type $List\ Int$, we first synthesise the type of Nil resulting in $T_{Nil} : List\ Int$[9]. We then check that this is a subtype of $List\ Int$[10].
9. We synthesise the type of Nil to be T_{Nil} from the context.
10. We apply the $<:\bigcup$ rule to check that $List\ Int$ is a subtype (non strict) of $List\ Int$. The first check is that the names are the same, as the names uniquely identify these types. We then iterate over the list of the arguments to the type constructor. The name of the context increments to reflect this iteration, but the context is unchanged during this check. There is only one type in the list

Synthesizing Example SFL explorer requires type assignments by default, but the below example does not have a type assignment as this allows us to hit some of the rules that we did not hit in the 'pairs' example. We have a function `f x = Just x` (desugared: `f = (x. Just x)`) that we want to synthesize the type of. `Just` is a constructor defined in the type declaration `data Maybe a = Just a | Nothing`, and it has a type $\forall a.a \to Maybe\,a$.

1. We begin synthesis with the $\to$I$\Rightarrow$ rule. We are trying to synthesize a type $A \to B$ as the abstraction $\lambda x.\ Just\ x$ must have a function type. We add to our context Γ_0 two existential type variables $\hat{\alpha}, \hat{\beta}$. We also add to our context the type assignment $x : \hat{\alpha}$, and then we check the inner body of the abstraction $(Just\ x)$ against $\hat{\beta}$ [1]. Check [1] yields the context Γ_4, and we discard everything after $x : \hat{\alpha}$ to remove data we no longer need. We synthesize $\lambda x.\ Just\ x \Rightarrow \hat{\alpha} \to \hat{\beta}$. Substituting the information from our

context Γ_4 gives us $\hat{\alpha} \to \mathit{Mabye}\ \hat{\alpha}$. This is the final result of inference, but an existential type variables are simply a statement that 'there exists a type', so they can be replaced with '$\forall$'s, giving our final result as $\forall a.a \to \mathit{Maybe}\, a$.

2. To check $\mathit{Just}\ x$ against $\hat{\beta}$, we synthesize $\mathit{Just}\ x \Rightarrow \mathit{Mabye}\ \hat{\gamma}$ [3]. After applying Γ_3 as a substitution to $\mathit{Mabye}\ \hat{\gamma}$ yielding $\mathit{Mabye}\ \hat{\alpha}$, we check this type against $\hat{\beta}$ [11], yielding Γ_4.

3. To synthesize the type of $\mathit{Just}\ x$, we synthesize $\mathit{Just} \Rightarrow T_{Just}$ [4], and synthesize the type of the application of type T_{Just} to x.

4. From the context: $\mathit{Just} \Rightarrow T_{Just}$.

5. To synthesize the type of applying $\forall \alpha.\alpha \to \mathit{Maybe}\, \alpha$ to x, we unwrap the forall by substituting all of the αs for a fresh existential type variable $\hat{\gamma}$. We then synthesisze the type of the application of $\hat{\gamma} \to \mathit{Maybe}\, \hat{\gamma}$ to x.

6. To synthesisze the type of the application of $\hat{\gamma} \to \mathit{Maybe}\, \hat{\gamma}$ to x, we check x against the first part of the arrow type $\hat{\gamma}$ [7]. We then yield the second part of the arrow type $\mathit{Maybe}\, \hat{\gamma}$.

7. To check x against $\hat{\gamma}$, we synthesize $x \Rightarrow \hat{\alpha}$ [8], and then check that $\hat{\alpha} <: \hat{\gamma}$ [9].

8. From the context: $x \Rightarrow \hat{\alpha}$.

9. We use the <:InstantiateL rule to derive $\hat{\alpha} <: \hat{\gamma}$. We could have equally used <:InstantiateR as both rules are for deriving a subtyping judgement where one side is an existential, but here both are existentials. To do this, we check that $\hat{\gamma}$ has no free instances of $\hat{\alpha}$, and we instantiate $\hat{\alpha}$ such that $\hat{\alpha} <: \hat{\gamma}$ [10].

10. We instantiate $\hat{\alpha}$ in our context Γ_2 to be a subtype of $\hat{\gamma}$ by setting $\hat{\gamma}$ to $\hat{\alpha}$ in our context, yielding Γ_3.

11. To check that $\mathit{Maybe}\, a <: \hat{\beta}$, we check that $\mathit{Maybe}\, a$ has no free instances of $\hat{\beta}$, and we instantiate $\hat{\beta}$ such that $\mathit{Maybe}\, a <: \hat{\beta}$ [12].

12. To instantiate $\hat{\beta}$ such that $\mathit{Maybe}\, a <: \hat{\beta}$, we use the rule InstRSolve to 'solve' type $\hat{\beta}$ to be $\mathit{Maybe}\, a$. This only requires that $\mathit{Maybe}\, a$ is well formed under the context, which means that the context 'knows about' all of the existential type variables.

4.3.4 Pairs

To support pairs, we must first add 'Pair' as an option to our enum '`ASTNodeType`' 3.1. A 'Pair' node has `ASTNodeType` of 'Pair', and two children.

Parsing Pairs

Parsing pairs is trivial:

- Expect an open parenthesis `(`.
- Parse an expression.
- Expect a comma.
- Parse a second expression.
- Expect a closing parenthesis `)`.

We then produce a '`Pair`' AST node with two children: the two expressions.

Updating the Redex Finding System With Pairs

Pairs themselves can never be redexes, so getting a list of all redex-contraction pairs in a 'Pair' is trivial: we concatenate the list of `RCPairs` in the left and right expression.

4.3.5 Pattern Matching

We add 'Match' as an option to our enum '`ASTNodeType`' 3.1. A match node has many children. The first node represents the expression being matched over, then each case followed by its corresponding definition.

Parsing Match Statements

An example of using a match statement follows:

```
lengthIsAtLeast2 list = match list {
  | Cons x (Cons y xs) -> true
  | _ => false
}
```

The algorithm used for parsing match statements is:

1. Consume the 'match' keyword.
2. Parse the expression matched over.
3. Consume an open brace.
4. While the next token isn't a close brace:
   (a) Consume bar '|'.
   (b) Parse a pattern (4.3.5).
   (c) Consume a right arrow.
   (d) Parse an expression.
5. Consume a close brace.

This creates a 'match' node, where the `children` vector is set appropriately with the pattern and expressions.

Patterns A pattern must not contain anything that can be reduced. It would be nonsensical to have a situation where we had a pattern not in normal form such as `1 + 1` and the expression to be matched was `2`.

To parse a pattern, we may use the same techniques as parsing an expression, with a few differences:

- Disallowing abstractions and match statements
- Identifiers must be either
  - Unbound lowercase variables
  - Underscore (_) representing a wildcard pattern
  - A bound uppercase variable (a constructor)

Updating the Redex Finding System With Pattern Matching

As discussed in the design (4.2.1), patterns are checked in order from first to last. Not only do we need to check that it does not currently match before moving on to checking the next pattern, we must check that it *can not* match the expression i.e. we must refute the pattern. In the below example, `repeat 1` must be evaluated enough to know whether it matches the first pattern before we move on to matching the second.

```
repeat :: a -> List a
repeat n = Cons n $ repeat n

main :: Bool
main = match repeat 1 {
    | Cons _ (Cons _ _) -> true
    | _ -> false
}
```

When matching an expression against a pattern, we have three possible results:

- Success: Matching was successful, and we have a list of what to bind
- Refute: We can not match this pattern, and evaluating the expression further would definitely not result in being able to match
- Unknown: It does not match, but we cannot refute

- `f 1 2 3` → `(\x y z. x) 1 2 3`
- `(\x y z. 1) 1 2 3` → `(\y z. 1) 2 3`
- `(\y z. 1) 2 3` → `(\z. 1) 3`
- `(\z. 1) 3` → `1`

The algorithm for finding the next evaluation step for a match expression is to sequentially attempt to match each pattern. If the result of matching an expression is a refutation, we check the next one. If the result is not yet known, we do not look at any further patterns, and we evaluate the expression further instead.

Below is a short summary of how pattern matching is done for different structures. This summary does not show all cases, but instead aims to give a general idea about how the algorithm works. To match an expression E against a pattern P and get all variables that the pattern would bind, we proceed case wise on the structure of P.

- An Identifier F.3.
  - If the identifier is a lowercase variable, we succeed matching and returning the binding.
  - If it is an underscore, matching succeeds, but we do not bind anything.
  - If it is an uppercase identifier (and therefore a constructor), we attempt to refute the pattern by showing that our expression will never evaluate to this constructor. Otherwise, our result is 'unknown'.
- An application: F.5. We proceed case wise on the structure of E:
  - Literal, Pair, Abstraction or Uppercase Identifier: Refute as these will not evaluate further.
  - Another application: Match the functions and match the arguments. If either Refutes then Refute, and if either is Unknown then return Unknown. Otherwise, Succeed, and concatenate the two lists of bindings
- A literal: trivial, the algorithm is just a string match. It is listed inline in F.2 for completeness's sake.

The algorithm for matching an expression against a single pattern, and getting either 'Success, Refute, or Unknown' is listed in pseudocode: F.2. The algorithm for getting a redex-contraction pair from a match statement is also listed F.1. If we succeed in pattern matching, the result should be the appropriate case with all the bindings substituted.

4.3.6 Multi Step Reduction and Lazy Mode

As it currently stands, the system gives users the option to select any possible redex to make progress. However, whilst demonstrating this project to my client, I found that these steps were often too small, and sometimes one larger step would have been easier. I found that having to apply a nested abstraction to concurrent terms particularly tedious. The expression `f 1 2 3` where `f = (xy z. x)` would have reduction sequence: Many users of the system, particularly more advanced users, would not need to see each nested step of this application. These can all be grouped into one step, where we substitute label f and perform all three reductions simultaneously.

[TODO: Algorithm if time, and reformat reduction]

Users may not always want to choose reduction order themselves. The 'Lazy' strategy 2.1.3 is the one employed by Haskell and other functional languages, so it should be the one employed by SFL. Conveniently, because of the way we are currently generating redexes, the list of redexes we generate already has the laziest option as the first element. This is because when generating redexes in an application, we calculate the redexes in the function before the redexes in the argument, which leads to a 'leftmost first' list of redexes.

4.3.7 The Prelude, and 'if e then a else b'

Most programming languages come with functionality packaged that is included by default, and is written in the language. In Haskell, this is referred to as the Prelude. There is also the standard library which is more extensive and is not imported by default.

As our language does not need extensive extra functionality, we do not need a whole standard library. However, a basic prelude with common functionality would be useful. C shows the SFL prelude. I included 'if' in the prelude to show that it is based on a match statement, rather than being a mysterious inbuilt:

```
if :: Bool -> a -> a -> a
if cond then_branch else_branch = match cond {
    | true -> then_branch
    | false -> else_branch
}
```

In order to make the language more like Haskell, I also added syntax sugar that allowed you to use it using the ‘`if e then a else b`’ syntax. The parser would ignore the ‘`then`’ and the ‘`else`’ keywords, and it would be equivalent to ‘`(((if e)  a)  b)`’ internally. However, this was unpopular with the advanced focus group, who said that this was confusing (see 4.5.2).

The prelude is listed in the appendix: C.

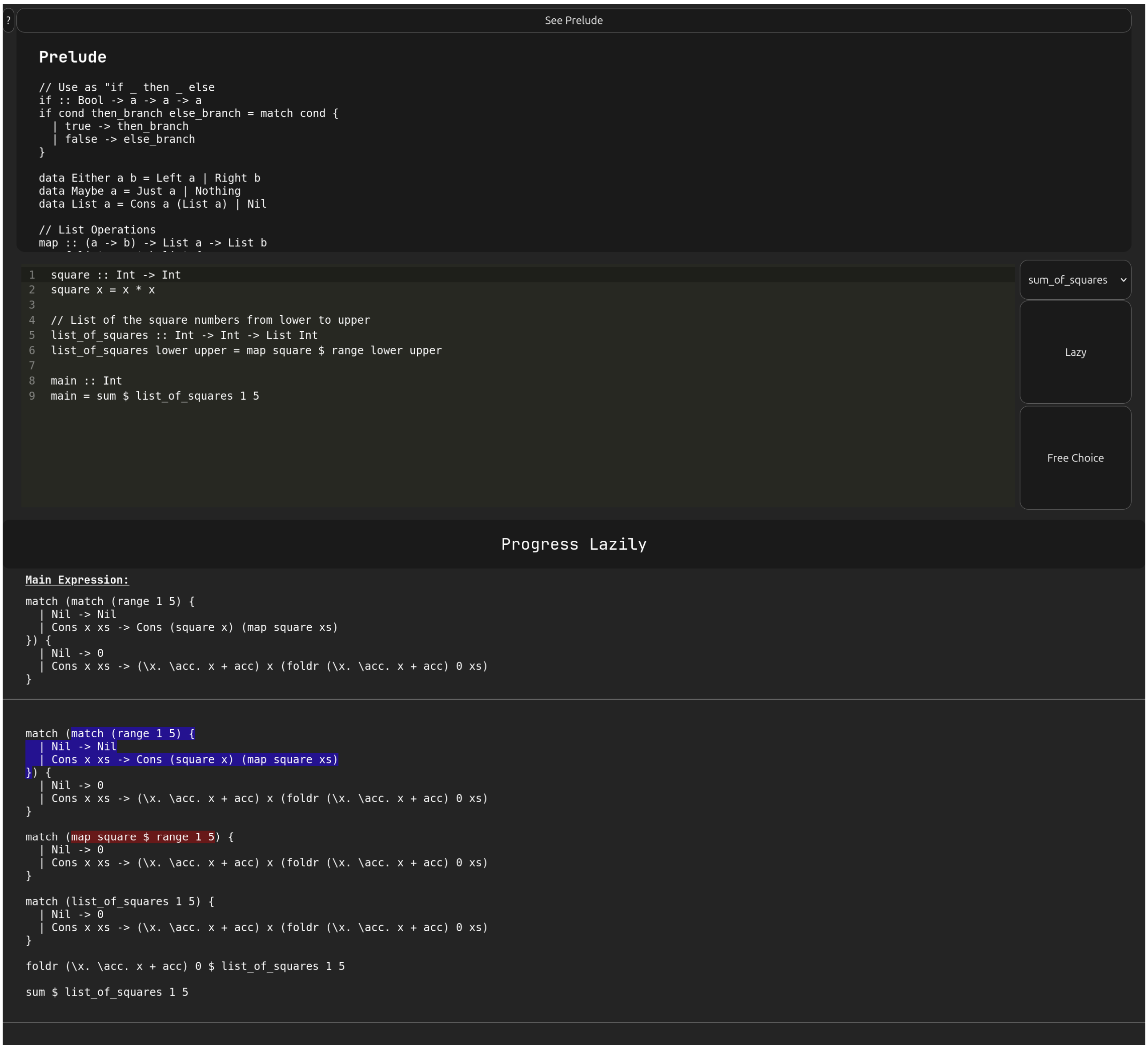


Figure 4.11: The product at the end of phase two during lazy evaluation of the ‘sum of squares’ sample program, with the prelude dropdown extended

4.3.8 Changes to the Proof of Concept UI

In this phase, I made some changes to the proof of concept web UI. See 4.11 and I.1 for the desktop UIs, and I.3 for the mobile UI.

Lazy Mode Added a separate ‘Lazy mode’ which would only offer one button labelled ‘Progress Lazily’. The original functionality was included in ‘Free Choice’ mode

History The history of the main expression is listed. The top two rows shows the most recent change, in blue is the result of the most recent change, in red is what it used to be. This does not work using the diff algorithm discussed in Phase 3 (5.2.1), it instead gets the string before and the string after, and locates them in the second most recent and most recent program state. This is not fully accurate, as a string match results in false positives. If we reduced `1 + 1` to `2` in the expression `(x. x + (1 + 1)) (1 + 1)`, it would highlight both `(1 + 1)`s even though the one in the abstraction has not been reduced.

Other

- The prelude was offered as a dropdown.
- Some example programs can be loaded from a dropdown.
- The program is saved in the browsers 'localStorage' as it is edited
- A help menu was offered when the page was loaded, or when the '?' button in the top left corner was pressed: I.4

4.4 Testathon

The testathon was a valuable opportunity to test my system at the midpoint of the project. During the testathon, I encouraged people to test the system on my laptop, as well as providing a QR code for them to be able to access it on their phone. I initially wanted to adopt a 'think aloud' method for usability testing, which is 'a method for studying mental process in which participants are asked to make spoken comment as they work on a task'[17]

The plan was to implement this, and passively watch them interact with the system and not give them any extra instruction. However, I found that people required significant instruction. I attempted to delegate any instruction to the 'help menu', but this did not solve the problem for the following reasons: people do not naturally want to read instructions, and my instructions were insufficient for people asked to interact with the system without any guidance to be able to effectively use it. Many people couldn't find the instructions, or were confused by the notation.

Data Gathering

After I explained the system and participants engaged with the system, participants were asked to fill out a survey. There were 15 participants, who were a mixture of undergraduate and postgraduate computer scientists, all of whom had taken the first year FP unit.

Participants were given some free-form questions:

- 'What do you like about the interface'
- 'Do you have any things I can improve about the interface'
- 'What do you like about the language'
- 'Do you have any language features you think would make it better. I am intending to add pattern matching, so not that'
- 'What do you like about the type system'
- 'Anything I can improve about the type system'

The answers to these questions were mostly complimentary, but some useful information was extracted:

- Participants appreciated the decluttered and simple UI.
- They noticed that certain UI elements overflowed their boundaries, and that the UI had visual glitches on Safari.
- They found the help menu too long and wordy, and not clear or to the point enough.
- They liked the language, the type system and the inference.

Key Takeaways

The findings from the testathon informed my future testing strategy:

- The 'Think aloud' method of watching people interact with this version of the system and asking them to narrate what they are doing is ineffective, as the UI is not 'self-explanatory' enough for people to be able to use it without help

- People do not want to read things.

Certain visual glitches were also identified and fixed in phase 2.

4.5 The Advanced Focus Group: Evaluation and Next Steps

The aims of this phase were to develop the language as well as some other more technical features of this project. To discuss the language, I held a focus group with students who were very knowledgable and interested in functional programming languages.

This was my first of three focus groups, the most advanced of the three. As the UI/UX was not polished at this stage, I wanted to find people who would be able to discuss the parts that I had already implemented to a reasonable level of completion: the language. However, I also wanted to discuss future steps for the system as a whole. Because of this, I wanted to find people who had learned functional languages as a part of a university course fairly recently and within memory, so they would have an insight into what is required for the system to be useful for use in this setting.

The transcript from this focus group is included in the additional submitted materials (see B)

4.5.1 Selection and Format

For this focus group, I recruited four students in their fourth year of studies here at the University of Bristol. They had all taken the first-year FP unit 3 years prior, and they had all taken units specializing in programming language theory since, including:

- The second year Programming Languages and Computation unit COMS20007, where they learnt to (among other things) 'Understand the interplay between the design and implementation of programming languages' [22]

- The third year optional Types and Lambda Calculus unit COMS30040 where they learnt (among other things): [23]
  - 'Type systems: types, judgements and rules'
  - 'Syntax and semantics of an untyped lambda calculus'

- The fourth year optional Advanced Topics in Programming Languages, where the unit outcomes were that they should be able to (among other things): [20]
  - 'Specify the dynamics of program evaluation for a variety of programming constructs'
  - 'Specify static typing rules for a variety of programming constructs'

These people I selected for this focus group were the closest to 'subject experts' that I could find while still being students. This focus group started with me briefly presenting SFL explorer. 4.11 shows how the system looked at this stage of the project. We also discussed the next UI iteration (see 4.2.2).

4.5.2 Outcomes

Below is the summary of outcomes from the discussion with this focus groups.

The Language

Positives:

- They liked the explicit match statements, and did not want me to change to more Haskell-like pattern match syntax:

  'Stick with the match expressions because it's very clear that matching has happened when you have the word match there' [Participant 2, 24:11]

- They liked that `Cons` was a prefix constructor rather than infix:

  'I think it's good that Cons is a prefix, like a normal constructor, and not a colon or something like that' [Participant 4, 17:41].

- Similarly, they liked the limited set of operators, and the fact that you cannot define your own:

  'I think if I was learning functional programming for the first time, I would really hope there aren't custom operators' [Participant 3, 15:56]

  'If I'm trying to learn functional programming, I don't think it helps me to be able to define, like, things that have different precedents. I think that distracts from learning how programs are reduced' [Participant 4, 16:42]

Negatives/Potential Improvements: Sentiment about the language was good, the only language specific issue was that they were confused about if-then-else syntax. They said it could be confusing to have the parser act differently for one specific function type.

'The issue I was having is just the fact that there is a function in the prelude which has the same name as some syntactic sugar that is a parser construct' [Participant 2, 42.55]

The Existing User Interface, and the System as a Whole

Positives They really appreciated its utility for what it was designed for. Most of what we discussed was potential improvements rather than positives of the proof of concept system, however they seemed engaged and excited despite not being explicitly positive about it, beyond this one comment:

'I think this is very good … I wish I'd had this in the functional labs' [Participant 3, 1:00:09]

Negatives/Potential Improvements:

- 'Syntax highlighting would obviously help' [21:45]

- They were confused as to why, in free choice mode, some reduction options included each other:

  'The first one is a chain of reductions that contains the second one. I think it's fine to display that as long as you make it visually distinct that these two are related in that way and the other reductions are just independent' [12:57]

  This could be implemented by having a dropdown where the highest level one shows all the ones below it.

  'You could put a number next to the reduction and say, you know, this is four steps. And then …make a drop-down. Yeah. So if someone wants to see what steps are going on inside there, then they could see' [08:57]

- They wanted an indication of which direction evaluation was going:

  'Because the reduction steps generate bottom-up, it might be good to have some sort of indication about the direction things are going in' [28:12]. This was already in the new UI, which had not seen by this point in the transcript.

- They wanted to be able to hover over an option and have it highlight what would change:

  'I think one thing that is not immediately clear is how the different reductions you see are related to the main program. If there was some way that like if you hovered over one, you could highlight the portion of the program that it corresponds to' [10:28]

The Next Iteration UI Design

Positives:

- The new UI included indication of which direction evaluation was going, see above.

- They liked the ability to go revert:

  'Something I had not thought of, very good' [54:59]

- They liked the horizontal split:

  'It's easier to have everything on screen and it's more akin to what people may have experienced' 'Its like compiler explorer'. [52:38] 'I think immediately not having to scroll is a massive plus' [52:58]

Negatives/Potential Improvements: No improvements were discussed for the next UI specifically, but most of the potential improvements for the current UI apply.

4.6 Phase 2 Conclusion

Phase 2 resulted in a good programming language which has syntax, semantics and a type system that are fit for purpose. The language was very popular with the Testathon users as well as the Advanced Focus Group. There were no specific complaints about the language from either group.

The proof of concept UI had mixed feedback. During the Testathon, people cited its 'cleanness' i.e. lack of overcomplicating buttons. However, people did not like having to scroll to refer to the original program, and the confusing nature of the way the history was generated bottom up. The advanced focus group also had a lot of feedback on how to

The new UI as designed at the beginning of phase 2 4.2.2 was popular with the Advanced Focus Group. They had no specific thoughts on how to improve it, however they had many thoughts on features that could be added to the UI and the system as a whole on how to make reduction clearer.

By the end of the project, I implemented the new UI along with syntax highlighting, and many other clarifying features.

However, I unfortunately did not have time to work on grouping related reductions together or highlighting in source code when a progress option is hovered over what it would change, or improving the help menu. See the 'future work' section 7.3 for more detail.

Chapter 5

Phase 3 — Improving the UI/UX

Phase 3 was the first of the two shorter phases focusing on UI iteration. It spanned approximately two weeks, starting with the implementation of the new UI, and ending with the second of the three focus groups with first year undergraduate students.

5.1 Requirements Analysis

The motivations for this phase come mainly from the advanced focus group, however requirements from the autoethnographic phase of the project, as well as the proof of concept client meeting continue to be relevant.

The advanced focus group was generally very positive about the language, but they had many thoughts about the Proof of Concept UI they were presented with. During phase 2, I created a Figma prototype for the next UI (see 4.2.2). Many of their thoughts about the Proof of Concept UI were things that were already addressed with the new design. This prototype was presented to the advanced focus group, who much preferred it. The advanced focus group had no criticism of the new UI, so it should be implemented as designed for now.

The prototype for the new UI also included the functionality to 'undo progress', by clicking on a previous program state in the table to make this the current version of the program. The advanced focus group appreciated this functionality.

5.2 Implementation

Implementing the new UI mostly consisted of time-consuming React and CSS tweaks which are not worth mentioning here. However, there were some more challenging aspects that required some more interesting considerations and changes to be made. Screenshots of the result of implementing the new UI with these features is shown in 5.1, with another example showing free choice evaluation in the appendix I.2.

5.2.1 Diff

Our frontend requires the ability to see what has changed between two program states. Highlighting these changes make understanding the changes in the users program in the frontend easier. This function generates the strings for the two trees simultaneously, producing the similarities and differences. If two nodes are different structures, then we turn them into strings and regard them as differences. However, if two nodes are the same structure, we identify what parts of that structure are similarities and what parts are differences. For instance, if the algorithm is called on an application, we generate the diff for the function and for the argument separately, and then concatenate the diffs. G.1 shows a subsection of this algorithm, showing how it works for IDs, Literals and Pairs.

5.2.2 Reduction Messages

Rather than presenting the user with simply the before and after of the reduction, this design calls for presenting the user with a message describing what will happen. While generating the options for reduction (see 3.3.4), we can keep track of information relevant to how it was generated to inform the message displayed. For instance, if a reduction is generated from the application of a named function with name A to two arguments B, C, we can convert those arguments to strings and then broadcast the message 'Applied function A to B and C'.

If B or C are large pieces of syntax, this may generate a very large unintelligible string. To solve this, we can modify our stringification algorithm to do certain things different to normal:

- Do not show the cases of a match statement, as the condition should be enough differentiate it
- We can truncate the output to a fixed length

In past iterations, redex-contraction pairs were passed to the front end as two strings. We can make it three strings instead, where one of the strings is the reduction message which can be displayed before the reduction. The other two strings, the redex and contraction, can be displayed after the reduction in the history.

5.2.3 Revert Progress Functionality

We may wish to undo progress. This was functionality designed into the new UI that the advanced focus group specifically mentioned liking.

Undoing progress requires that previous AST states must be stored. Before now, the most recent AST state was stored at a known memory address so any of the functions in the binary could know where to find it. This was done to avoid having to pass the AST to the JavaScript module. If we wanted to store the history of all ASTs, one approach could be to store all the ASTs in a pre-allocated memory region in a stack, and then allow the JavaScript module to refer to each of the ASTs in the history by their stack index. However, pre-allocating enough memory for any potential program execution logs would be misguided, as it would cause accessibility problems for computers with less memory. Instead, we should employ dynamic allocation.

The issue with dynamic allocation of memory for the ASTs as they are added to our history is that we no longer know exactly where they will be located, meaning this information must be stored such that it will not be erased between calls to WASM library functions. One method of doing this is passing a pointer to where in memory the AST is located to the JavaScript module so that it can refer to it later, and use library functions on it. At first glance, this sounds like a bad idea, as when pointers are returned from a function for which wasm-bindgen (see **??**) is used to make a JavaScript binding, the pointer is represented as a JavaScript *number* type [28], which is a double-precision IEEE-754 value [9]. Storing pointers as floating point values, and then attempting to dereference them, sounds like a recipe for memory mismanagement. However, this is safe because WebAssembly 2.0 has 32 (see 2.5), and thus has 32 bit pointers, and a double precision floating point number has a 52 bit [13] mantissa meaning it can safely store the 32-bit memory location without issue.

In our JavaScript module, we can then store a stack of pointers to the ASTs, and display the options for reducing the one at the top. When an option is selected, we can apply the reduction and then store the new AST on the top of our stack and recalculate reduction options. If the user decides to start evaluating a new program, all the ASTs with pointers in this list are freed to avoid memory leaks.

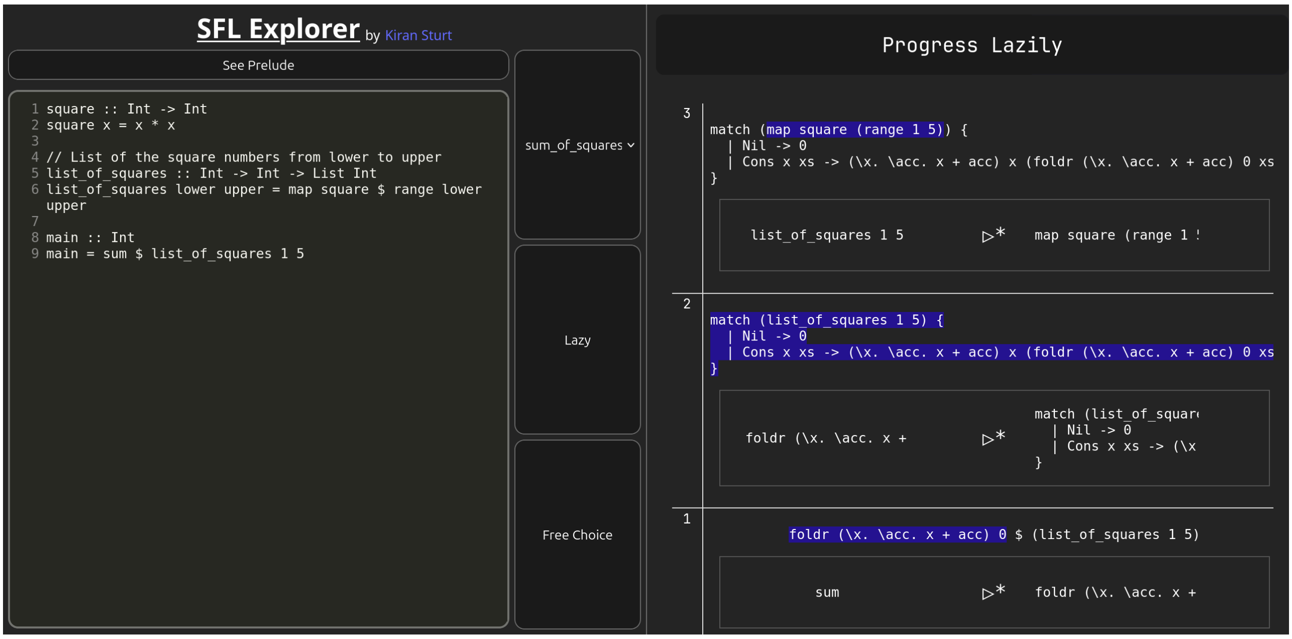


Figure 5.1: The new UI implemented, during lazy evaluation of the provided 'sum of squares' example program

5.3 The Intermediate Focus Group: Evaluation and Next Steps

The Intermediate focus group happened at the end of phase 3 of the project. An entirely different structure was employed for this focus group than the Advanced Focus Group 4.5, as the aims of this focus group were different.

Rather than just evaluating the language, I wanted to evaluate the system in the setting that I expected it to be used in: a lecture setting.

To act as a lecturer, I employed Amos Holland, a fourth year undergraduate student. I considered doing it myself, but I decided that it would be better to employ an undergraduate who was more experienced in teaching functional programming. Amos was selected as he had taken all of the functional programming units that were available in his degree so far: the same ones that the participants of the Advanced Focus Group had taken (for details of these units, see 4.5). Furthermore, Amos had acted as a teaching assistant on two of these units, Types and Lambda Calculus, where he ran weekly problem classes, where he would walk students through difficult problems from the lectures or worksheets, and COMS10016 where he had been a teaching assignment working in labs in the same capacity as I had.

5.3.1 Selection and Format

For this focus, to evaluate SFL explorer as a teaching tool and get feedback, I selected first year undergraduate students who had just taken the functional programming unit. I looked for people who had not explored functional programming outside this unit, and who found the unit difficult, as these are representative of group of people the tool would be most useful for.

5.3.2 Bugs

There were numerous bugs that Amos and I identified during the lecture phase of this focus group:

- The editor would sometimes add tab spaces randomly.
- Sometimes, the site would refresh on its own.
- The editor did not save input consistently.

These were all fixed as part of phase 4.

5.3.3 Outcomes

Due to technical issues, I was only able to get a transcript of the interview portion of the intermediate focus group; I made the mistake of holding this focus group in a room where there was too much background noise to get a high quality recording during this part of the session. However, during the interview, I set up multiple microphones, and was able to combine the AI transcripts from each of them into one transcript. The transcript is in the additional materials B.

Below is the summary of outcomes from the discussion with this focus group, along with timestamped quotes where relevant. The first phase of this interview was conducted with Amos, followed by an interview with each participant in turn.

All four participants, and Amos, liked SFL explorer as a whole.

> ‘I think it was good, other than the few bugs’ [Amos, 00:04]

> ‘Yeah, I liked it; this is what I do in my head basically when I’m looking at Haskell, it [the lecture using the tool] was good’ [Participant 1, 06:44]

> ‘I really like what the tool does, like it shows like what you can see because that’s basically what I do when I’m like debugging code but um it takes like a long time to do it like by hand and to go through everything and sometimes it’s like wrong if you do it yourself so this is like a good way to automate it’ [Participant 2, 15:20]

> ‘I liked it’ [Participant 3, 19:07]

> ‘I actually really love the functionality of this app’ [Participant 4, 24:44]

In the Advanced Focus Group outcomes section 4.5, I split the focus group’s thoughts into positives and negatives. However, in this focus group opinions were more split between participants, as well as participants being more frequently in two minds, so I did not do this. Instead, I have grouped by topic rather than by positivity and negativity, so one bullet point may have both positive and negative thoughts towards the same issue.

The Language

Below is the summary of Amos's and the participants' thoughts about SFL, along with timestamped quotes where appropriate.

Amos

- Amos liked the explicit match statement:

  'I quite like match, because I think it gets a bit more directly at what pattern matching is doing, like that it's a specific operation' [03:36]

- Amos was happy with the language and its features

  'There are features of Haskell it doesn't have, but Haskell is a very advanced language. For teaching functional programming, I think it's got the right range of features' [Amos, 00:04]

- Amos identified adding type classes as a potential next step

  'If you were to take this, like, a step further, make it more advanced, I think the next step is type classes' [Amos, 00:04]

- 'I like the stuff we put in the prelude because we have like sort of basic functions' [Amos, 00:04]

The Participants

- They had mixed opinions on whether I should add the usual syntax sugar for lists. Below are two quotes from two participants, showing their split minds on this topic:

  'I both love and hate the list, the lack of syntactic sugar for list, …it makes literal lists really hard to read, but it makes the types so much clearer. Having to interact with that rather than just going, that's kind of an array …It really explicitly forces you to think in the Haskell way' [Participant 1, 14:47]

  'I really liked that, like, well, because we don't have the constructor we had in Haskell for, like, lists. Abundantly much clearer to me what was going on …I prefer it with this list, this cons and the nil. But also you have to write it out for five hours as the teacher' [Participant 3, 22:52]

- They liked the explicit match statements.

  'I honestly really liked the match being, like, write match. Yeah. So the explicitness was good. I really liked that. It made it obvious to me in a way that it wasn't necessarily before … This is much easier to learn about pattern matching with than Haskell' [Participant 3, 19:07]

The User Interface, and the System as a Whole

An absolutely critical issue that became obvious during the focus group was the need for a light mode. The room was very bright, and it was barely visible in dark mode. This was resolved with a high priority in the next phase. Below a more detailed summary of their thoughts about SFL Explorer as a whole, and the UI/UX:

Amos

- Amos was generally happy with the UI for teaching.

  'As far as the design went, I think I was happy with it for teaching' [Amos, 00:04]

- A 'step to the end' options would be good, allowing the user to complete evaluation without having to click on the button many times.

  'It would have been nice to have a button that says steps to the end, because there were some cases where I would have liked to just see if it evaluated correctly, but instead you had to click through a lot of times' [Amos, 00:04]

- Amos wanted to be able to save more programs than just one.

  'It would be nice to be able to define more complete programs and save them so that you can jump between them a bit' [Amos, 02:57]

The Participants

- They wanted syntax highlighting, indeed 3 out of four explicitly asked for it.
- Two of the participants agreed that the history should be shown in reverse, so it would generate top down, the other two did not comment.
- One participant specifically mentioned liking the way the UI separates the editor from the output

  'I like that it's separate. Text editor here. Then this bit shows what it does' [18:02]
- They liked how it highlights what has changed between iterations:

  'I really like the highlighting in like what changed' [Participant 2, 15:20]

  'I, like, as a minor visual thing, … like, you have the blue bit of the bit you've changed on the right-hand side' [Participant 3, 19:07]

5.4 Phase 3 Conclusion

This phase resulted in a high quality UI that was much more polished than the previous iteration. The UI was generally popular with the focus group, and the language continued to be well liked. However, there were many ideas for improvements to be made, the clearest one being adding a light mode.

Chapter 6

Phase 4 — Further UI/UX Iteration

6.1 Requirements Analysis

This phase was limited by time, as the project was nearing its end. For this reason, this phase was mostly focused on fixing the high priority issues identified in the previous phase. This included adding a light mode, adding syntax highlighting, as well as fixing language and typechecker bugs.

At the end of this phase, I wanted to hold another focus group where Amos would give a lecture on functional programming, but this time to complete beginners. After a planning conversation with Amos at the beginning of Phase 4, we identified that it would be useful to add an 'untyped mode' so the beginners could be taught the basics of λ-calculus before trying to explain types, as they can be initially confusing.

6.2 Design and Implementation

6.2.1 Syntax Highlighting

In order to implement syntax highlighting, I found the source code for the Haskell syntax highlighting supported by the library I was using for my editor (CodeMirror 5). I edited this with SFL's syntax and keywords. Syntax highlighting was also applied to the prelude to make it easier to read.

6.2.2 Light Mode, and the Settings Menu

The 'light mode' colour scheme was designed by returning to the room where the Intermediate Focus Group was held on a day with similar amounts of sunshine, and testing different colours for visibility. The light mode scheme also had different syntax highlighting from dark mode.

To implement it, I added a floating settings menu with a button that would toggle from light mode to dark mode. This worked by adding or removing the class 'light' from the top level HTML element, where all elements descending from this node would be in light mode if it was set. I also added a button for toggling 'untyped mode', as well as toggling whether the prelude was included. These settings would be saved in the user's browser.

Using CSS media queries, I was also able to tell the user's preference for light or dark mode from their browser, and use this by default unless the user chose otherwise.

6.2.3 Bugfixes

Frontend

6.3 The Beginner Focus Group: Evaluation and Next Steps

6.3.1 Selection and Format

This focus group had the same format as the intermediate focus group 5.3; Amos was employed once more to give a 45-minute lecture, followed by a 45-minute interview. The full transcript of the lecture and interview is available in the additional materials B.

The goals of this focus group was to evaluate the use of SFL explorer in a lecture setting with complete beginners, with a wide variety of different experiences and perspectives on programming. Crucially, I wanted none of them to have learnt about or used functional languages before. As such, I decided to recruit non

Computer Scientists. Hoping to get a mix of more ‘practical’ and more ‘theoretical’ students, the four students I recruited were from the following disciplines:

- A second year undergraduate Mechanical and Aerospace Engineer from the University of Cambridge, with experience in Python
- A third year undergraduate Mechanical Engineer from the University of Bristol, with experience in Python and C/C++
- Two third year undergraduate Maths students from the University of Bristol, with experience in mostly Python and R

6.3.2 Bugs

There was a bug where the type checker would not correctly follow type aliases, causing crashes in a program with type aliases. This was fixed afterwards.

6.3.3 Outcomes

The Language

Participant’s did not have many comments on the language, as they had never seen a functional language before, so they couldn’t really compare.

However, Participant 2, a Maths student, in particular seemed excited about functional programming as a concept:

> ‘It’s cool to see a language that’s like, I don’t know, I feel like if I was going to do some maths in my head, this is how I’d actually do it’ [Participant 2, 01:05:22].

Participant 2 also asked several questions about the history of functional programming, showing their engagement.

Participant 1 was confused about the bar character used to separate cases in a match statement, as it was the same as the character used to separate tagged union variants.

The User Interface, and the System as a Whole

Most of the feedback was positive, and users had no specific feedback on what they would like to see be changed in the system.

- Two participants said they preferred dark mode, but both agreed light mode is important to have.
- Free choice mode was very popular, with two participants specifically mentioning when asked for things they liked in the system.
- Participant 3 really liked the interface, and found the session particularly interesting:

  ‘I really like the coding part and the sections on the right, in terms of seeing how it goes from one to another. I think that was really useful. In terms of learning it. Yeah. And the free choice variation. I also agree that it shows nicely what the functions are actually doing’ [Participant 3, 01:03:11]
- Participant 2, a Maths student, was very positive about the UI and the tool:

  ‘I think you’ve done a really good job of making it quite beginner-friendly because it’s all quite easy to read’ [Participant 2, 01:06:05]
- ‘UX is very intuitive’ [Participant 1, 01:12:07]

6.4 The Final Client Meeting

[TODO: This bit] After the beginner focus group, I met my client. This was at the end of the project

My client liked my project, and shared it with Jamie Willis, a teaching fellow at Imperial College London, who is involved in their first year unit which includes functional programming [24]. When my client asked about whether he would use it, he responded:

> ‘I could see it being useful for sure, a lot of the time I end up writing out these step-by-step reductions by hand in my notes, but it would be nice for them to have access to a tool they could use to explore for themselves’

Chapter 7

Conclusion

The aims of this project were to create a system to help to build an intuitive understanding of functional programming languages

7.1 Strengths

7.1.1 The System is Useful, and Will be Used

As discussed previously 6.4, my client will use this project in future in teaching COMS10016. Furthermore, she shared it with a teaching fellow who teaches Haskell at Imperial, who agreed that it is useful.

7.1.2 The Language Achieves Its Design Aims

See 3.1.3 for the initial discussion of the design goals.

Design Aim 1: 'It Should be Simple and Easy to Understand' All three focus groups have supported the conclusion that the language is easy to understand. The advanced and intermediate focus groups appreciated the relatively small deviations from Haskell such as the explicit match expressions, and the small set of inbuilts. Indeed, one participant in the intermediate focus group said that they thought the explicit matching syntax was much easier to understand than Haskell's pattern matching, saying it was better for learning 5.3. The beginner focus group also had little difficulty grasping the syntax and semantics of the language in a lecture context, however they would have found it harder without guidance. Sentiment was more divided about the fact that `Cons` is not an infix operator

Design Aim 2: 'It Should be Similar to Existing Functional Languages' Both the advanced and intermediate focus groups, both with participants that had been formally taught Haskell, had very little trouble understanding the language.

Design Aim 3: 'It Should be Powerful Enough to Explain Key FP Concepts' Amos Holland used SFL explorer to explain key concepts to the intermediate 5.3 and beginner 6.3 focus groups.

> 'There are features of Haskell it doesn't have, but Haskell is a very advanced language. For teaching functional programming, I think it's got the right range of features'. Amos, During an interview after lecturing in the intermediate focus group

The language is also capable of doing everything that my client mentioned that she wanted to use such a system for (3.4).

7.2 Limitations

7.2.1 The Expressions Balloon During Evaluation

I believe that the languages lack of inbuilts is one of the languages best 'features'. However, it is also a curse: as everything is defined with match expressions, the expression balloons vertically with match statements during evaluation. For instance, in the provided 'square_sum' example:

```
square :: Int -> Int
square x = x * x

// List of the square numbers from lower to upper
list_of_squares :: Int -> Int -> List Int
list_of_squares lower upper = map square $ range lower upper

main :: Int
main = sum $ list_of_squares 1 5
```

Despite their being no match expressions in sight, the 'main' expression balloons to 3 match statements deep within 6 lazy steps:

```
match (match (match (infiniteFrom 1) {
    | Nil -> Nil
    | Cons x xs -> if ((5 - 1) > 0) (Cons x (take ((5 - 1) - 1) xs)) Nil
}) {
    | Nil -> Nil
    | Cons x xs -> Cons (square x) (map square xs)
}) {
    | Nil -> 0
    | Cons x xs -> (\x. \acc. x + acc) x (foldr (\x. \acc. x + acc) 0 xs)
}
```

The outer one comes from 'sum', the middle one comes from 'map', and the inner one comes from 'range', all prelude functions. Unfortunately, this is hard to avoid, as pattern matching is a key concept in functional programming languages. Furthermore, a conclusion of the intermediate focus group was that the explicit match syntax, where it was obvious where/how pattern matching was occurring, made understanding pattern matching much easier. Indeed, they agreed that they would have liked to have SFL to learn about pattern matching rather than Haskell (see 5.3).

This situation could be improved by being able to select which functions we are interested in seeing the expansion of, and which ones we are not. See 7.3

7.2.2 There is No Documentation

When designing the new UI (4.2.2), I included buttons that would create help menus, and more information about the project, as well as instructions. However, these are time-consuming to write,

7.3 Future Work

Add More Documentation to the Website As the language is quite similar to Haskell, an advanced user would not have much trouble figuring out how the website works. This works fine for a lecture tool as the lecturer would be able to figure it out, the lack of documentation is detrimental to other users.

Other Evaluation Strategies Users could have the option to pick the evaluation strategy (2.1.3).

Improve Free Choice Mode Inspiration could be taken from the UI used by λ-Lessons (2.3) where the expression to be evaluated could be selected by clicking on the input text itself.

Selective Skipping We are not always interested in all the functions involved in our program. For instance, if a lecturer is attempting to demonstrate `foldr` over a list, they may not be interested in the expansion of how `range` works in order to generate their list they are going to fold over. They may want the evaluation of some things to be 'skipped'.

We could mark certain expressions as uninteresting, and evaluate them as much as we can immediately. For instance, if the syntax for an uninteresting expression looked like '[e]':

```
main :: Int
main = sum $ [range 1 4]
```

We could fully evaluate '`range 1 4`' to '`Cons 1 (Cons 2 (Cons 3 Nil))`'. However, this could cause issues if the term does not evaluate.

Extensions to the language As suggested by Amos during the intermediate focus group (5.3) the language could be extended with typeclasses.

Bibliography

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Alternative eText Formats Series. Addison-Wesley, 2007. URL: https://books.google.co.uk/books?id=WomBPgAACAAJ.

[2] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. Accessed: 2025-04-06. URL: https://agilemanifesto.org/.

[4] MANUEL M. T. CHAKRAVARTY and GABRIELE KELLER. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, 2004. `doi:10.1017/S0956796803004805`.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[6] Chris Done. duet: A tiny language, a subset of haskell (with type classes) aimed at aiding teachers to teach haskell, 2019. URL: https://hackage.haskell.org/package/duet.

[7] Chris Done. Teaching haskell with duet, 2020. URL: https://chrisdone.com/toys/duet-delta/.

[8] Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Int'l Conf. Functional Programming*, September 2013. arXiv:1306.6032[cs.PL].

[9] Ecma International. Ecmascript language specification. https://tc39.es/ecma262/#sec-ecmascript-language-types-number-type, 2025. Section 6.1.6: The Number Type.

[10] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

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

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

[13] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic, 2008. IEEE Std 754-2008, IEEE, New York, NY, USA.

[14] Steve Krouse Jan Paul Posma. https://stevekrouse.github.io/hs.js/, 2014. Accessed April 19, 2025.

[15] Steve Krouse Jan Paul Posma. https://github.com/stevekrouse/hs.js/, 2014. Accessed April 19, 2025, commit hash `20e2ce74c4b0ce83604bde8e99671cbf3712c439`.

[16] S. Klabnik and C. Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023. URL: https://books.google.co.uk/books?id=a8l9EAAAQBAJ.

[17] C. Lewis. *Using the "thinking Aloud" Method in Cognitive Interface Design*. Research report. IBM Thomas J. Watson Research Division, 1982. URL: https://books.google.co.uk/books?id=F5AKHQAACAAJ.

[18] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[19] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144, doi:10.1016/0022-0000(78)90014-4.

[20] University of Bristol School of Computer Science. Unit information: Advanced topics in programming languages (teaching unit) in 2024/25, 2025. URL: https://www.bris.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%2F25&unitCode=COMSM0067.

[21] University of Bristol School of Computer Science. Unit information: Imperative and functional programming in 2024/25, 2025. URL: https://www.bristol.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%252F25&unitCode=COMS10016.

[22] University of Bristol School of Computer Science. Unit information: Programming languages and computation in 2024/25, 2025. URL: https://www.bristol.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%252F25&unitCode=COMS20007.

[23] University of Bristol School of Computer Science. Unit information: Types and lambda calculus (teaching unit) in 2024/25, 2025. URL: https://www.bristol.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%2F25&unitCode=COMS30040.

[24] Imperial College London Department of Computer Science. Computing practical 1 module unit website, 2025. URL: https://www.imperial.ac.uk/computing/current-students/courses/40009/.

[25] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002. URL: https://books.google.co.uk/books?id=hPL6DwAAQBAJ.

[26] Amon Rapp. *Autoethnography in Human-Computer Interaction: Theory and Practice*, pages 25–42. Springer International Publishing, 06 2018. doi:10.1007/978-3-319-73374-6_3.

[27] Andreas Rossberg. WebAssembly Core Specification. Technical report, W3C, 2022. URL: https://www.w3.org/TR/wasm-core-2.

[28] Rust and WebAssembly Working Group. `wasm-bindgen` guide. https://rustwasm.github.io/wasm-bindgen/, 2024. Accessed April 16, 2025.

[29] A. M. Turing. Computability and -definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937. doi:10.2307/2268280.

[30] Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *International Symposium on Functional and Logic Programming*, pages 119–135. Springer, 2014.

Appendix A

AI Usage

I did not directly prompt any Large Language Models, to assist with the writing of my dissertation or implementation. However, as listed in the Supporting Technologies list, I used GitHub Copilot to help with writing some tests for the parser and type checker. I used it via the VS Code extension, which uses the context of your file, to provide advanced AI autocompletion.

I also used AI to transcribe from the audio files of the focus groups (see B)

Appendix B

Additional Materials

File Name	**Description**	**How to Open**
`Figma_Design.fig`	The Figma Prototype of the design	Using Figma Desktop or Web
`afg_transcript.pdf`	The AI transcript from the audio recording of the advanced focus group	Using a PDF editor
`bfg_transcript.pdf`	The AI transcript from the audio recording of the beginner focus group	
`phase[2, 3, 4]_end.zip`	The ready-to-serve built application as it was at the end of phases 2, 3, 4 respectively.	Unzip, and then serve the 'dist' folder. An easy way is to run `python3 -m http.server 3000` in the 'dist' folder to serve on port 3000, and then go to `localhost:3000` in the browser. Unfortunately, I was not able to package the end of phase 1 product in a way that was as simple to serve, however 3.3 shows what it looked like
`source.zip`	The final source code for the project	Unzip. To build, follow instructions in `README.md`
`testathon_form.xlsx`	The results of the testathon survey	

Appendix C

The SFL Prelude

```
if :: Bool -> a -> a -> a
if cond then_branch else_branch = match cond {
  | true -> then_branch
  | false -> else_branch
}

data Either a b = Left a | Right b
data Maybe a = Just a | Nothing
data List a = Cons a (List a) | Nil

// List Operations
map :: (a -> b) -> List a -> List b
map f list = match list {
  | Nil -> Nil
  | Cons x xs -> Cons (f x) (map f xs)
}

foldr :: (a -> b -> b) -> b -> List a -> b
foldr f acc list = match list {
  | Nil -> acc
  | Cons x xs -> f x (foldr f acc xs)
}

filter :: (a -> Bool) -> List a -> List a
filter pred list = match list :: List a {
  | Nil -> Nil
  | Cons x xs -> if (pred x) (Cons x (filter pred xs)) (filter pred xs)
}

repeat :: a -> List a
repeat n = Cons n $ repeat n

length :: List a -> Int
length xs = foldr (\_ i. i + 1) 0 xs

infiniteFrom :: Int -> List Int
infiniteFrom x = Cons x (infiniteFrom (x + 1))

take :: Int -> List a -> List a
take n list = match list {
  | Nil -> Nil
  | Cons x xs -> if (n > 0) (Cons x (take (n - 1) xs)) (Nil)
}

range :: Int -> Int -> List Int
```

```
range lower upper = take (upper - lower) $ infiniteFrom lower

sum :: List Int -> Int
sum = foldr (\x acc. x + acc) 0
```

Appendix D

An Additional Derivation Using the Type Checking Algorithm

$$\Gamma = \alpha, \beta, x : \alpha, y : \beta$$

$$\Delta = \alpha$$

$$\dfrac{\dfrac{(x : \alpha) \in \{\Gamma\}}{\Gamma \vdash x \Rightarrow \alpha \dashv \Gamma}\ \textsf{Var}\ [8] \qquad \dfrac{}{\Gamma[\alpha] \vdash \alpha <: \alpha \dashv \Gamma[\alpha]}\ {<:}\textsf{Var}\ [9]}{\Gamma \vdash x \Leftarrow \alpha \dashv \Gamma}\ \textsf{Sub}\ [6]$$

$$\dfrac{\dfrac{(y : \beta) \in \{\Gamma\}}{\Gamma \vdash y \Rightarrow \beta \dashv \Gamma}\ \textsf{Var}\ [10] \qquad \dfrac{}{\Gamma[\beta] \vdash \beta <: \beta \dashv \Gamma[\beta]}\ {<:}\textsf{Var}\ [11]}{\Gamma \vdash y \Leftarrow \beta \dashv \Gamma}\ \textsf{Sub}\ [7]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[6]\Gamma \vdash x \Leftarrow \alpha \dashv \Gamma \qquad [7]\Gamma \vdash y \Leftarrow \beta \dashv \Gamma}{\Gamma \vdash (x, y) \Leftarrow (\alpha, \beta) \dashv \Gamma}\ \textsf{Pair}{\Leftarrow}\ [5]}{\alpha, \beta, x : \alpha \vdash \lambda y.\, (x, y) \Leftarrow \beta \to (\alpha, \beta) \dashv \Gamma}\ {\to}\textsf{I}\ [4]}{\alpha, \beta \vdash \lambda x\ y.\, (x, y) \Leftarrow \alpha \to \beta \to (\alpha, \beta) \dashv \Delta, \beta, \{x : \alpha, y : \beta\} \quad (= \Gamma)}\ {\to}\textsf{I}\ [3]}{\dfrac{\alpha \vdash \lambda x\ y.\, (x, y) \Leftarrow \forall \beta.\, \alpha \to \beta \to (\alpha, \beta) \dashv \{.\}, \alpha, \{.\} \quad (= \Delta)}{.\vdash \lambda x\ y.\, (x, y) \Leftarrow \forall \alpha\ \beta.\, \alpha \to \beta \to (\alpha, \beta) \dashv .}\ \forall\textsf{I}\ [1]}\ \forall\textsf{I}\ [2]$$

Figure D.1: An example derivation showing how we can typecheck the function $\lambda x \quad y.\, (x, y)$ against $\forall \alpha\ \beta.\, \alpha \to \beta \to (\alpha, \beta)$

Typechecking the Pair Function The pair function $\lambda x\ y.\, (x, y)$ takes two arguments, and returns a pair of the two values. Here, we type check it against its correct type $\forall \alpha\ \beta.\, \alpha \to \beta \to (\alpha, \beta)$.

1. Here, we begin typechecking with the $\forall$I rule to introduce $\forall \alpha$. We do this by adding α to the initially empty context. We then check the function against the type without the $\forall \alpha$: $\forall \beta.\, \alpha \to \beta \to (\alpha, \beta)$. The output of this checking is then split into 3 parts: everything before the α, the α itself, and the bits after the α. Our output context is everything in Δ before the alpha, which is nothing.

2. We apply the same rule as above, but we are introducing $\forall \beta$. We then check the function against $\alpha \to \beta \to (\alpha, \beta)$ Our output context for this rule is everything in Γ before the β: only α.

3. We then start to unwrap the abstractions. We strip the abstraction over x from the expression, leaving us with $(\lambda y.\, (x, y))$. We then add $(x : \alpha)$ to our context, and then progress by checking the remaining part of the expression against $\beta \to (\alpha, \beta)$. Our output context is Γ.

4. Same as above, with y against β. We unwrap the abstraction over y to give us (x, y). We then check this against (α, β). The output context is Γ.

5. We check (x, y) against the type (α, β). To check this, we check x against α and y against β. The output context is Γ.

6. To check x against type α we synthesise the type of x ([9]: trivial, as its in the context). We then check this against α, and a check of α against α ([10]) trivially passes.

7. Same as above with y against β. The output context is Γ

Appendix E

Typechecking Algorithm

Below is the full algorithm for typechecking SFL programs. This algorithm comes mostly from [8], my additions are highlighted.

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against input type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes output type A, with output context Δ

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C \dashv \Delta}$ Under input context Γ, applying a function of type A to e synthesizes type C, with output context Δ

$$\frac{}{\Gamma \vdash \mathit{IntLiteral} \Leftarrow \text{Int} \dashv \Gamma}\ \textsf{IntLit}{\Leftarrow} \qquad \frac{}{\Gamma \vdash \mathit{IntLiteral} \Rightarrow \text{Int} \dashv \Gamma}\ \textsf{IntLit}{\Rightarrow}$$

$$\frac{}{\Gamma \vdash \mathit{BoolLiteral} \Leftarrow \text{Bool} \dashv \Gamma}\ \textsf{BoolLit}{\Leftarrow} \qquad \frac{}{\Gamma \vdash \mathit{BoolLiteral} \Rightarrow \text{Bool} \dashv \Gamma}\ \textsf{BoolLit}{\Rightarrow}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A \dashv \Theta \qquad \Theta \vdash e_2 \Leftarrow B \dashv \Delta}{\Gamma \vdash (e_1, e_2) \Leftarrow (A, B) \dashv \Delta}\ \textsf{Pair}{\Leftarrow} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash e_2 \Rightarrow B \dashv \Delta}{\Gamma \vdash (e_1, e_2) \Rightarrow (A, B) \dashv \Delta}\ \textsf{Pair}{\Rightarrow}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsf{Var} \qquad \frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \textsf{Sub}$$

$$\frac{\Gamma, \alpha \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \Leftarrow \forall \alpha.\, A \dashv \Delta}\ \forall\textsf{I} \qquad \frac{\Gamma, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall \alpha.\, A \bullet e \Rrightarrow C \dashv \Delta}\ \forall\textsf{App} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\ {\to}\textsf{I}$$

$$\frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta} \dashv \Delta, x : \hat{\alpha}, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \hat{\alpha} \to \hat{\beta} \dashv \Delta}\ {\to}\textsf{I}{\Rightarrow} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Rrightarrow C \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow C \dashv \Delta}\ {\to}\textsf{E}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \bullet e \Rrightarrow \hat{\alpha}_2 \dashv \Delta}\ \hat{\alpha}\textsf{App} \qquad \frac{\Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash A \to C \bullet e \Rrightarrow C \dashv \Delta}\ {\to}\textsf{App}$$

$$\frac{\begin{array}{cc} \Gamma \vdash e \Rightarrow A \dashv \Theta_1 & (\Theta_i \vdash c_i \Leftarrow A \dashv \Theta_{i+1}) \text{ for } i \text{ in } [1, 2, \dots, n] \\ \Delta_1 = \Theta_{n+1}, \hat{\alpha} & (\Delta_i \vdash e_i \Leftarrow \hat{\alpha} \dashv \Delta_{i+1}) \text{ for } i \text{ in } [1, 2, \dots, n] \end{array}}{\Gamma \vdash \text{match e } \{c_1 \to e_1 \mid c_2 \to e_2 \mid \cdots \mid c_n \to e_n\} \Rightarrow \hat{\alpha} \dashv \Delta_{n+1}}\ \textsf{Match}{\Rightarrow}$$

Figure E.1: Algorithmic typing. The rules with highlighted names are my additions, the rest are unchanged from [8].

$\boxed{\Gamma \vdash A <: B \dashv \Delta}$ Under input context Γ, type A is a subtype of B, with output context Δ

$$\frac{}{\Gamma[\alpha] \vdash \alpha <: \alpha \dashv \Gamma[\alpha]}\ {<:}\mathsf{Var} \qquad \frac{}{\Gamma \vdash \mathrm{Int} <: \mathrm{Int} \dashv \Gamma}\ {<:}\mathsf{Int} \qquad \frac{}{\Gamma \vdash \mathrm{Bool} <: \mathrm{Bool} \dashv \Gamma}\ {<:}\mathsf{Bool}$$

$$\frac{}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} <: \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}\ {<:}\mathsf{Exvar} \qquad \frac{\Gamma \vdash B_1 <: A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 <: [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \dashv \Delta}\ {<:}{\to}$$

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A <: B \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash \forall\alpha.\, A <: B \dashv \Delta}\ {<:}\forall\mathsf{L} \qquad \frac{\Gamma, \alpha \vdash A <: B \dashv \Delta, \alpha, \Theta}{\Gamma \vdash A <: \forall\alpha.\, B \dashv \Delta}\ {<:}\forall\mathsf{R}$$

$$\frac{\hat{\alpha} \notin FV(A) \qquad \Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq A \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} <: A \dashv \Delta}\ {<:}\mathsf{InstantiateL} \qquad \frac{\hat{\alpha} \notin FV(A) \qquad \Gamma[\hat{\alpha}] \vdash A \leqq: \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash A <: \hat{\alpha} \dashv \Delta}\ {<:}\mathsf{InstantiateR}$$

$$\frac{\Gamma \vdash A_1 <: A_2 \dashv \Theta \qquad \Theta \vdash B_1 <: B_2 \dashv \Delta}{\Gamma \vdash (A_1, B_1) <: (A_2, B_2) \dashv \Delta}\ {<:}{\times}$$

$$\frac{(\Gamma_i \vdash A_i <: B_i \dashv \Gamma_{i+1}) \text{ for } i \text{ in } [1, 2, \dots, n]}{\Gamma_1 \vdash \mathrm{Name}[A_1, A_2, \dots, A_n] <: \mathrm{Name}[B_1, B_2, \dots, B_n] \dashv \Gamma_{n+1}}\ {<:}{\bigcup}$$

Figure E.2: Algorithmic subtyping. The rules with highlighted names are my additions, the rest are unchanged from [8]

$\boxed{\Gamma \vdash \hat{\alpha} :\leqq A \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $\hat{\alpha} <: A$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash \hat{\alpha} :\leqq \tau \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'}\ \mathsf{InstLSolve} \qquad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\alpha} :\leqq \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}\ \mathsf{InstLReach}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash A_1 \leqq: \hat{\alpha}_1 \dashv \Theta \qquad \Theta \vdash \hat{\alpha}_2 :\leqq [\Theta]A_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq A_1 \to A_2 \dashv \Delta}\ \mathsf{InstLArr}$$

$$\frac{\Gamma[\hat{\alpha}], \beta \vdash \hat{\alpha} :\leqq B \dashv \Delta, \beta, \Delta'}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq \forall\beta.\, B \dashv \Delta}\ \mathsf{InstLAllR}$$

$\boxed{\Gamma \vdash A \leqq: \hat{\alpha} \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $A <: \hat{\alpha}$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash \tau \leqq: \hat{\alpha} \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'}\ \mathsf{InstRSolve} \qquad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\beta} \leqq: \hat{\alpha} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}\ \mathsf{InstRReach}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash \hat{\alpha}_1 :\leqq A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \leqq: \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash A_1 \to A_2 \leqq: \hat{\alpha} \dashv \Delta}\ \mathsf{InstRArr}$$

$$\frac{\Gamma[\hat{\alpha}], \blacktriangleright_{\hat{\beta}}, \hat{\beta} \vdash [\hat{\beta}/\beta]B \leqq: \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{\beta}}, \Delta'}{\Gamma[\hat{\alpha}] \vdash \forall\beta.\, B \leqq: \hat{\alpha} \dashv \Delta}\ \mathsf{InstRAllL}$$

Figure E.3: Instantiation. These rules are unmodified from [8]

Appendix F

Pattern Matching Algorithm

A pattern consists of only:

- An application
- A literal
- A pair
- An identifier: could be a wildcard, a constructor.

Below is the algorithm for each of these cases, as well as the top level pattern matching algorithm.

```
fn get_redex_from_match(match_expression) -> Option<RedexContractionPair> {
    // get the expression being matched
    let expr = match_expression.to_be_matched
    // Iterate through all the patterns and their resulting expressions
    for ((pattern, resulting_expr) in match_expression.cases) {
        let result = pattern_match(expr, pattern);
        if (result == Refute) {
            // If refuted, then we can safely consider next pattern
            continue
        }
        if (result == Unknown) {
            // We get the reduction option for the expression
            // as we cannot refute this pattern
            return get_redex(expr)
        }
        if (result == Success(bindings)) {
            return Some(RedexContractionPair {
                from: match_expression,
                to: resulting_expr.substitute(bindings),
                reduction_message: "Match to pattern" + pattern.to_string()
            })
        }
    }
    // Refuted all patterns
    return None
}
```

Figure F.1: The algorithm for getting the redex-contraction pair from a match expression. If we sucessfully match, the result will be the expression corresponding to the matching pattern. If we cannot match expressions

```
fn pattern_match(expr, pattern) -> MatchResult {
    if (pattern is identifier) {match_against_identifier(expr, pattern)}
    if (pattern is a pair) {match_against_pair(expr, pattern)}
    if (pattern is an app) {match_against_application(expr, pattern)}
    if (pattern is a literal) {
        if (expr.to_string() == pattern.to_string())
            return Success([])
        } else {
            return Refute
        }
    }
}
```

Figure F.2: The algorithm for matching an expression against a pattern

```
fn match_against_identifier(expr, pattern) -> MatchResult {
    if (pattern is "_") {
        // Succeed but dont bind anything
        Success([])
    }
    if (pattern is a lowercase identifier) {
        // We suceed as a lowercase ID is a wildcard, and we must add to
        // our list of bindings the fact that the named wildcard now has a
        // value: the expr
        Success([(pattern.string, expr)])
    }
    if (pattern is a constructor (i.e. is uppercase)) {
        if (expr is also a constructor with the same name) {
            return Success([])
        }
        if (expr is an application) {
            // `Head' refers to the recursive front of an application. For
            // instance, The head of (Left ((Cons x) xs)) would be Left.
            if (the head of expr is a constructor) {
                // We can refute, as constructors never evaluate, so the
                // structure of the expression will never be the same as
                // the pattern.
                return Refute
            } else {
                // Otherwise further evaluation might lead to a pattern
                // that matches this constructor so we cant refute yet
                return Unknown
            }
        }
        return Unknown;
    }
}

```

Figure F.3: The algorithm for matching an expression against a pattern that is an identifier in rust like pseudocode

```
fn match_against_pair(expr, pattern) -> MatchResult {
    if (expr is also a pair) {
        let first = pattern_match(expr.first, pattern.first);
        let second = pattern_match(expr.second, pattern.second);

        // Propogate refute and unknown
        if (first == Unknown || second == Unknown) {
            return Unknown;
        }
        if (first == Refute || second == Refute) {
            return Refute;
        }
        // first and second have suceeded, return both sets of bindings
        return Success(first.bindings + second.bindings)
    }
    if (expr is an application) {
        if (the head of expr is a constructor) {
            return Refute
        } else {
            return Unknown
        }
    }
    if (expr is a literal || expr is an abstraction
        || expr is an uppercase identifier) {return Refute}

    return Unknown // catchall: only `match'
}
```

Figure F.4: The algorithm for matching an expression against a pattern that is a pair in rust like pseudocode. See F.3 for more detail about the ‘expr is application’ case

```
fn match_against_application(expr, pattern) -> MatchResult {
    if (expr is also an application) {
        let func = pattern_match(expr.func, pattern.func);
        let arg = pattern_match(expr.arg, pattern.arg);

        // Propogate refute and unknown
        if (func == Unknown || arg == Unknown) {
            return Unknown;
        }
        if (func == Refute || arg == Refute) {
            return Refute;
        }
        // func and arg have suceeded, return both sets of bindings
        return Success(func.bindings + arg.bindings)
    }
    if (expr is a literal || expr is a pair || expr is an abstraction
        || expr is an uppercase identifier) {return Refute}
    return Unknown
}
```

Figure F.5: The algorithm for matching an expression against a pattern that is a pair in rust like pseudocode. See F.3 for more detail about the ‘expr is application’ case

Appendix G

A Section of the Difference Algorithm

```
enum DiffElem {
    Similarity(String),
    Difference(String, String)
}

type Diff = Vec<DiffElem>

// rust-like psuedocode, not valid rust
// ast1 and 2 are the two ASTs, and expr1 and 2 are the indices
// of the terms we are considering for our diff.
fn diff(ast1, ast2, expr1, expr2) -> Diff {
    node1, node2 = ast1.get(expr1), ast2.get(expr2)
    diff = Diff::new();
    match (node1, node2) {
        // IDs and Lits are compared based on their string "values"
        case (ID, ID)
        case (Lit, Lit) {
            if node1.value == node2.value {
                diff += Similarity(node1.value)
            } else {
                diff += Difference(node1.value, node2.value)
            }
        }

        case (Pair {first1, second1}, Pair {first2, second2}) {
            // As both are pairs, their opening brackets,
            // commas, and closing brackets are in common.

            // We get the diff of the first and second
            // element to find the diff of the whole pair
            diff += Similarity("(")
            diff += diff(ast1, ast2, first1, first2)
            diff += Similarity(",")
            diff += diff(ast1, ast2, second1, second2)
            diff += Similarity(")")
        }

        ...
    }

    return diff;
}

```

Figure G.1: rust-like psuedocode listing for the type of the output of the AST::diff function, as well as a small section of the algorithm. There is also (not shown) a wrapper around the Diff type, to allow for conversion into JavaScript (see 2.5), as well as the some logic for combining diffs.

Appendix H

Tokens for Lexical Analysis

Below is the code for how tokens outputted by lexical analysis are defined.

```
enum TokenType {
    EOF,
    Newline,

    Id,
    UppercaseId,

    If,
    Then,
    Else,

    Match,
    LBrace,
    RBrace,

    IntLit,
    FloatLit,
    StringLit,
    CharLit,
    BoolLit,

    DoubleColon,
    RArrow,
    Forall,
    KWType,
    KWData,

    LParen,
    RParen,

    Lambda,

    Dollar,
    Dot,
    Comma,
    Bar,

    Assignment,
}

struct Token {
    tt: TokenType,
    value: String,
}
```

Appendix I

UI Screenshots

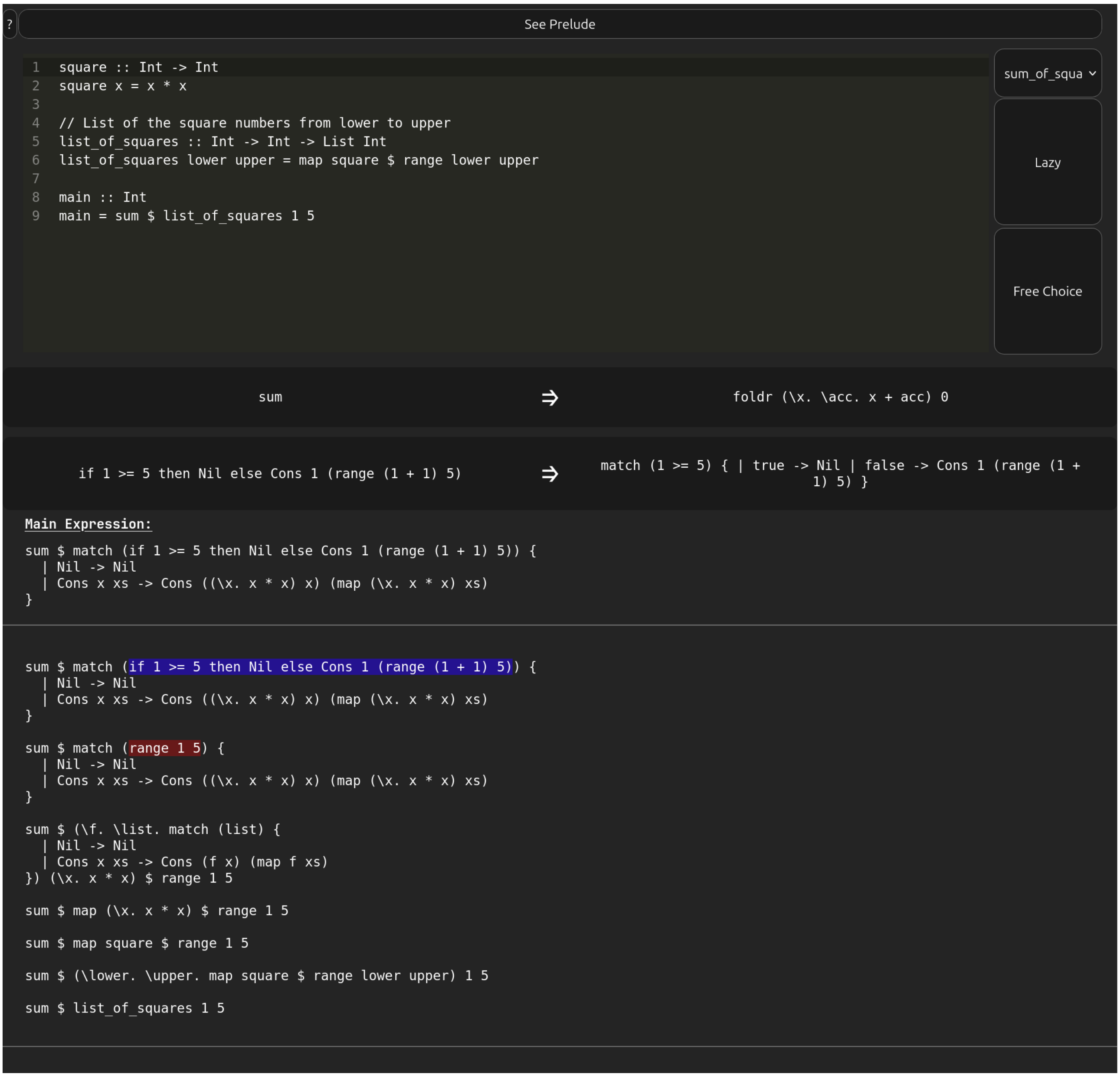


Figure I.1: The product at the end of phase two during free choice evaluation of the 'sum of squares' sample program, with the prelude dropdown contracted

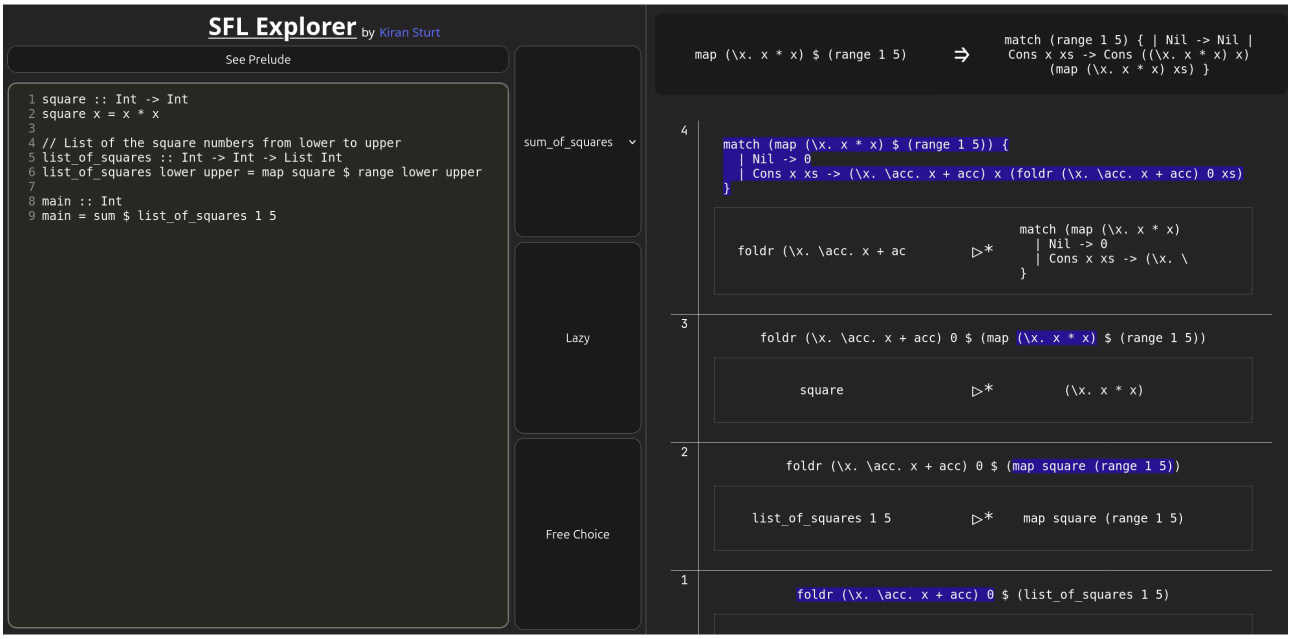


Figure I.2: The new UI implemented at the end of phase three, during free choice evaluation of the provided ‘sum of squares’ example program

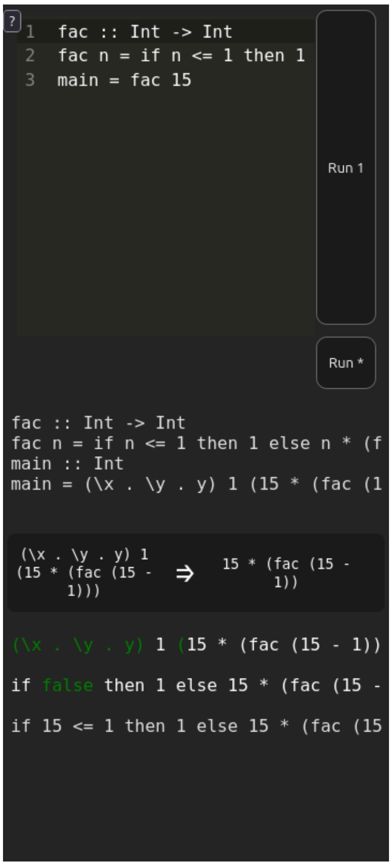
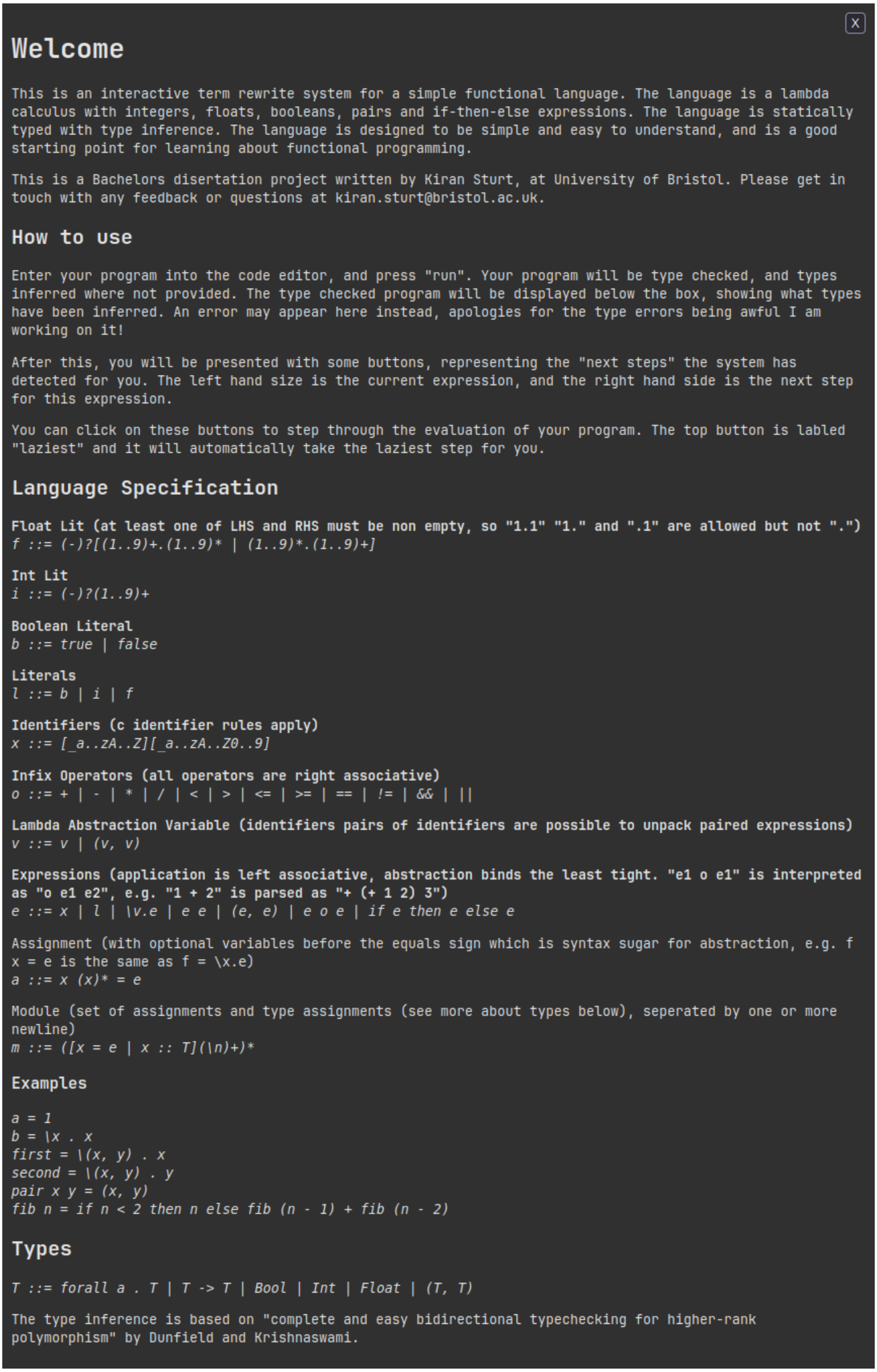


Figure I.3: The UI as it appeared at the end of phase 2, as it would have appeared on a Samsung Galaxy S20



Welcome

This is an interactive term rewrite system for a simple functional language. The language is a lambda calculus with integers, floats, booleans, pairs and if-then-else expressions. The language is statically typed with type inference. The language is designed to be simple and easy to understand, and is a good starting point for learning about functional programming.

This is a Bachelors disertation project written by Kiran Sturt, at University of Bristol. Please get in touch with any feedback or questions at kiran.sturt@bristol.ac.uk.

How to use

Enter your program into the code editor, and press "run". Your program will be type checked, and types inferred where not provided. The type checked program will be displayed below the box, showing what types have been inferred. An error may appear here instead, apologies for the type errors being awful I am working on it!

After this, you will be presented with some buttons, representing the "next steps" the system has detected for you. The left hand size is the current expression, and the right hand side is the next step for this expression.

You can click on these buttons to step through the evaluation of your program. The top button is labled "laziest" and it will automatically take the laziest step for you.

Language Specification

Float Lit (at least one of LHS and RHS must be non empty, so "1.1" "1." and ".1" are allowed but not ".")
```
f ::= (-)?[(1..9)+.(1..9)* | (1..9)*.(1..9)+]
```

Int Lit
```
i ::= (-)?(1..9)+
```

Boolean Literal
```
b ::= true | false
```

Literals
```
l ::= b | i | f
```

Identifiers (c identifier rules apply)
```
x ::= [_a..zA..Z][_a..zA..Z0..9]
```

Infix Operators (all operators are right associative)
```
o ::= + | - | * | / | < | > | <= | >= | == | != | && | ||
```

Lambda Abstraction Variable (identifiers pairs of identifiers are possible to unpack paired expressions)
```
v ::= v | (v, v)
```

Expressions (application is left associative, abstraction binds the least tight. "e1 o e1" is interpreted as "o e1 e2", e.g. "1 + 2" is parsed as "+ (+ 1 2) 3")
```
e ::= x | l | \v.e | e e | (e, e) | e o e | if e then e else e
```

Assignment (with optional variables before the equals sign which is syntax sugar for abstraction, e.g. f x = e is the same as f = \x.e)
```
a ::= x (x)* = e
```

Module (set of assignments and type assignments (see more about types below), seperated by one or more newline)
```
m ::= ([x = e | x :: T](\n)+)*
```

Examples

```
a = 1
b = \x . x
first = \(x, y) . x
second = \(x, y) . y
pair x y = (x, y)
fib n = if n < 2 then n else fib (n - 1) + fib (n - 2)
```

Types

```
T ::= forall a . T | T -> T | Bool | Int | Float | (T, T)
```

The type inference is based on "complete and easy bidirectional typechecking for higher-rank polymorphism" by Dunfield and Krishnaswami.

Figure I.4: The ‘Help menu’ in the proof of concept UI. This was spawned by pressing the ‘?’ button in the top left of the UI, and dismissed by pressing the ‘X’ button, or clicking outside the box

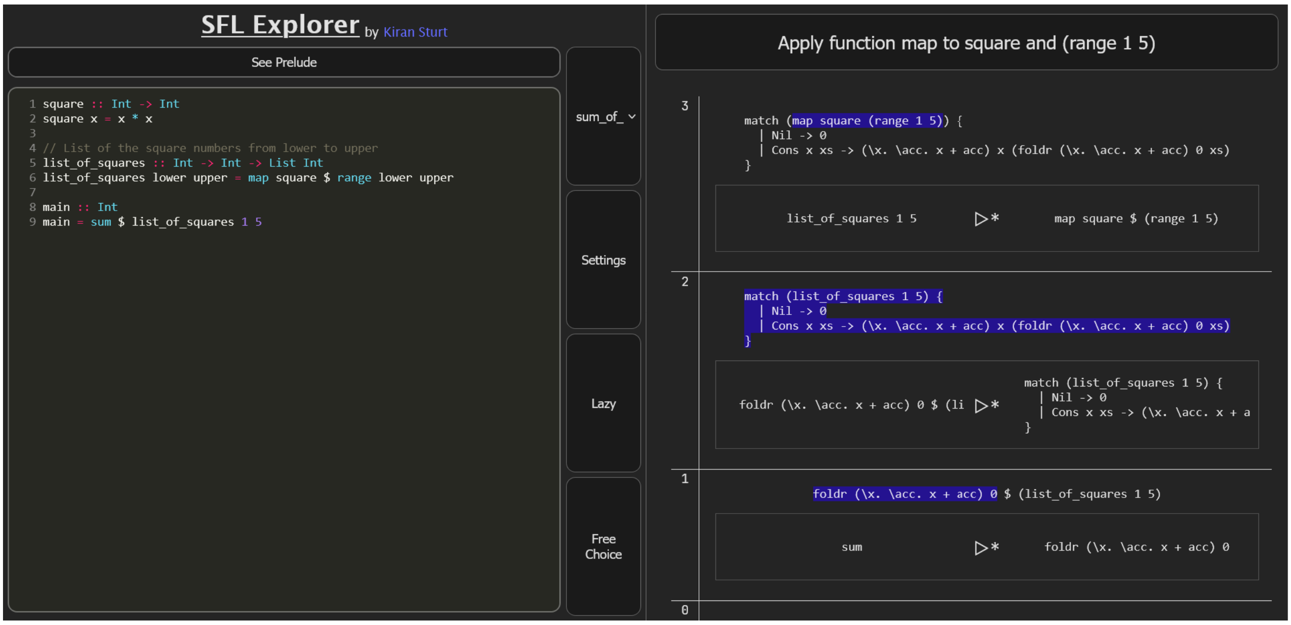


Figure I.5: The final product during lazy evaluation of the 'sum of squares' sample program

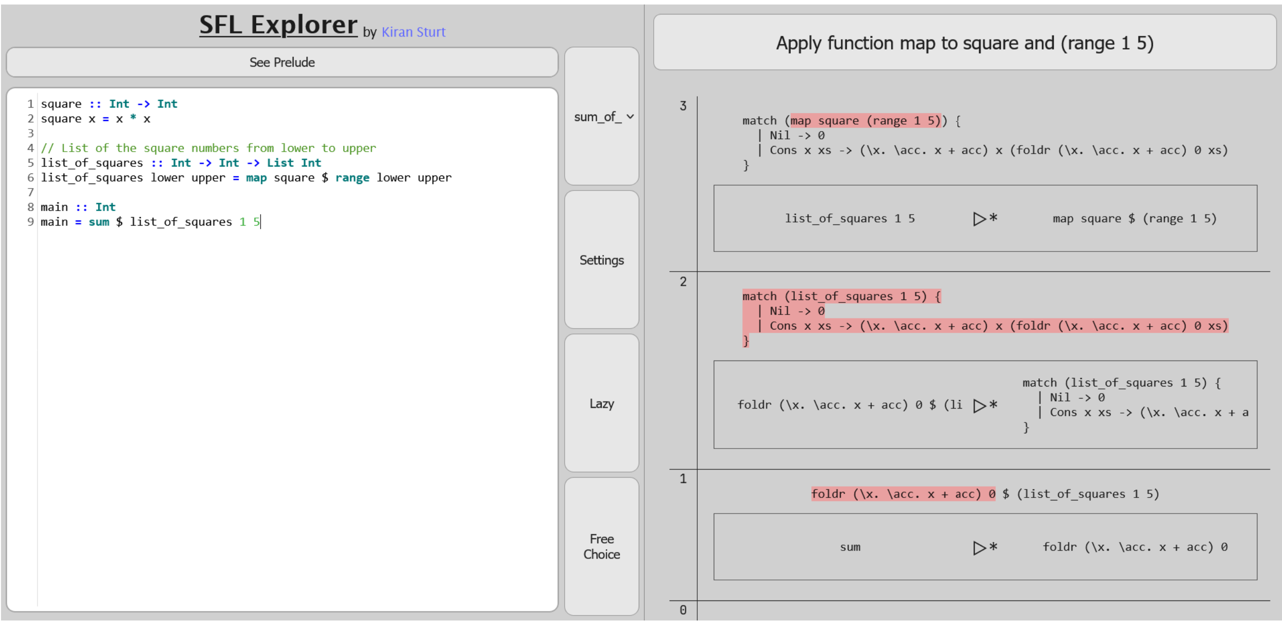


Figure I.6: The final product during lazy evaluation of the 'sum of squares' sample program in light mode

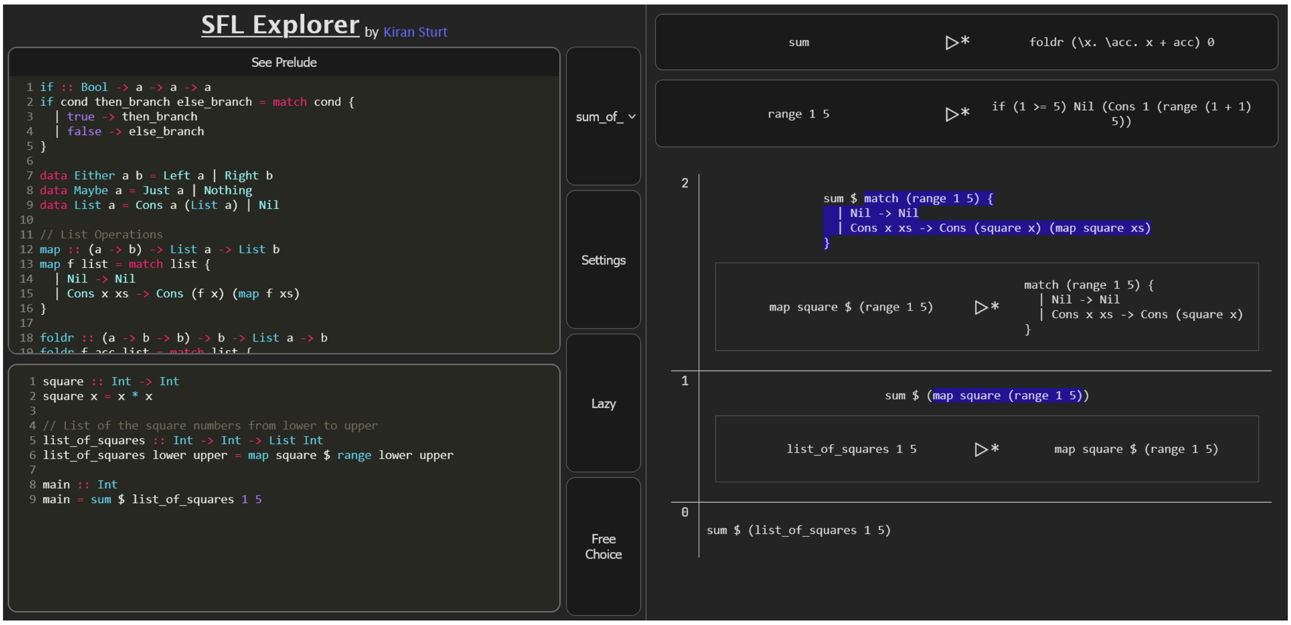


Figure I.7: The final product during free choice evaluation of the 'sum of squares' sample program, with the prelude visible

Figure I.8: The final product during free choice evaluation of the 'sum of squares' sample program, with the prelude visible in light mode