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**.

Friday 11th April, 2025

Abstract

Dedication and Acknowledgements

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, Friday 11$^{\text{th}}$ 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, Friday 11th April, 2025

Contents

1 Introduction **1**

2 Background **2**
- 2.1 The Lambda Calculus . . . 2
- 2.2 Haskell: A Functional Programming Language . . . 4
- 2.3 Rust . . . 5
- 2.4 Frontend Technologies . . . 5
- 2.5 Web Assembly . . . 5

3 Lifecycle **6**

4 Cycle 1 - Proof of Concept **8**
- 4.1 Requirements Analysis . . . 8
- 4.2 Language Design . . . 10
- 4.3 Implementation . . . 11
- 4.4 Evaluation . . . 20

5 Cycle 2 - Types and Pattern Matching **22**
- 5.1 Requirements Analysis . . . 22
- 5.2 Design . . . 22
- 5.3 Implementation . . . 23
- 5.4 Evaluation . . . 25

6 Conclusion **28**

A AI Usage **30**

B Some Example Derivations Using the Type Checking Algorithm **31**
- B.1 Typechecking the Pair Function . . . 31
- B.2 Typechecking an Expression Involving Lists . . . 32

C Tokens for Lexical Analysis **34**

D UI Screenshots **36**

E Language Grammar **42**

List of Figures

3.1 Caption 7

4.1 The results of a survey performed during the testathon, where a Likert scale was used to gauge 15 participants feelings towards imperative (a) and functional (b) programming languages 9
4.2 A word cloud of answers entered into the free form text box presented to respondents who reported that they did not disagree that functional languages were hard to learn 10
4.3 The Rust code listing for the definition of the AST, with lifetime specifiers, accessibility modifiers, and the syntax information (see 4.3.1) removed for conciseness. 12
4.4 An alternative implementation with a few advantages over the actual implementation. . . 15
4.5 The Rust code 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. 16

5.1 Syntax of types and monotypes. Note that this is the external definition: as seen by the users of SFL. See 5.3 for the extra type system structures required internally for the typechecker 23
5.2 The Rust code listing for the definition of types. Existential is separated as it is more of an implementation detail than a part of the type system 24
5.3 Syntax of types, monotypes, and contexts as seen by the typechecker. Note that the existential type variables ($\hat{\alpha}$) can not actually be created by users, they are an implementation detail 25
5.4 Applying a context, as a substitution, to a type 25
5.5 Algorithmic subtyping. The rules with highlighted names are my additions, the rest are unchanged from [5] 26
5.6 Instantiation. These rules are unmodified from [5] 26
5.7 Algorithmic typing. The rules with highlighted names are my additions, the rest are unchanged from [5]. 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 27

D.1 The Web UI Minimum Viable Product (MVP), as presented to my client at the end of cycle 1. Note that this does have type assignments, but these were just ignored by the parser and typechecker at this stage. 36
D.2 Screenshot 1 of the Figma design of the web UI 37
D.3 Screenshot 2 of the Figma design of the web UI. This version shows the prelude extended 37
D.4 The UI as tested in the testathon 38
D.5 The UI as tested in the testathon, as it would have appeared on a Samsung Galaxy S20 . 38
D.6 The ”Help menu” in the testathon. This was spawned by pressing the ”?” button in the top left of the UI, and dismissed by pressing the ”X” button, or clicking outside of the box 39
D.7 The final product during lazy evaluation of the ‘sum of squares’ sample program 40
D.8 The final product during lazy evaluation of the ‘sum of squares’ sample program in light mode 40

D.9 The final product during free choice evaluation of the ‘sum of squares’ sample program, with the prelude visible 41
D.10 The final product during free choice evaluation of the ‘sum of squares’ sample program, with the prelude visible in light mode 41

List of Tables

Ethics Statement

Supporting Technologies

- I used React (https://react.dev/) 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: https://github.com/rustwasm/wasm-pack.
- 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

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.

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.

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.

Chapter 2

Background

2.1 The Lambda Calculus

The lambda calculus (λ-calculus) was described by Alonzo Church in 1936 [4]. Below is a definition of the λ-calculus[2]

Definition 1 (The λ-calculus)
The set of λ-terms, notation Λ, is built up from an infinite set of variables $V = v, v', v'', \ldots$ using application and (function) abstraction:

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

Using abstract syntax one may write the following:

$$\begin{aligned} V &::= v \mid V' \\ \Lambda &::= V \mid (\Lambda\Lambda) \mid (\lambda V \Lambda) \end{aligned}$$

We shall also use the following conventions:

Definition 2 (The λ-Calculus Conventions)

1. *$x, y, z, \ldots$ denote arbitrary variables*
2. *We may omit outermost parenthesis*
3. *Nested abstractions can be grouped. For instance, if we were to write the term $\lambda x\ y.M$, we would mean $(\lambda x.(\lambda\ y.M))$*
4. *We assume that the body of an abstraction extends as far to the right as possible. For instance, the term $\lambda x.M\ N$ means $(\lambda x.(M\ N))$ and not $((\lambda x.M)\ N)$*

2.1.1 Reduction

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 how this substitution works, followed by the definition of β-reduction.

Definition 3 (Substitution of free variables in terms of the λ-calculus)

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

Definition 4 (β-Reduction of the λ-calculus)

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

Definition 5 (Normal Form)
A term is said to be in normal form if it cannot be β-reduced

2.1.2 Types

The λ-calculus that we have discussed so far is untyped. This means that any λ term can be applied to any other term. When a term can no longer be β-reduced (i.e. no further reductions are possible), we say it is a value. This is why β-reduction is often referred to as *evaluation* — it is the process by which terms are reduced to their final, fully evaluated form (i.e. values).

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. 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' [12]

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.

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

Typechecking: Well typed programs do not go wrong

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

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

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.

We will now attempt to derive the type of x, in order to show that it is untypable. 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 'untypable'. 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 typable 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** [11]. Therefore, if we are able to exclude all terms in our functional language that are untypable, 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’ [6]. 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 [8]. 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

2.2.3 User Defined Data Types

Haskell allows users to define their own types. For instance, booleans can be defined as

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

Many languages, including Haskell, also 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.

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.

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 $Int \cup (Int \times Int)$.

2.2.4 Polymorphic Types

Haskell includes polymorphic types. These are ‘types that are universally quantified in some way over all types’ [7].

One example is a list that can hold any value of type a, written as *List a*”. Here, *List* is not a type in itself, but it represents a constructor that takes a type, and returns a concrete type. We could write this as ”$Type \to Type$”, indicating that it behaves like a function, but at the type level rather than the value level. If we were to apply the constructor *List* to the concrete type *Int*, the resulting type would be the concrete type *List Int*.

This is first-order polymorphism (rank 1 polymorphism), as opposed to higher-order polymorphism (also known as higher-kinded polymorphism) where a type can ”abstraction over type constructors” [15].

The ”Type of a Type” is its *kind* [12]. For example, the type constructor *List* has the kind $Type \to Type$

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

2.2.5 Pattern Matching

Functional languages have a long history that we do not need to go into

2.3 Rust

This project is written in rust. 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 [9]:

- 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 [9]:

- 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

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 [14]. This means we only have 4GB 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 webasembly 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

Chapter 3

Lifecycle

The project followed a development lifecycle inspired by Agile principles[3], structured into four iterative cycles. Each cycle was further subdivided into four phases:

- Requirements Analysis.
- Design
- Implementation
- Evaluation

In particular, the project benefitted from agile principles emphasising:

- bingus

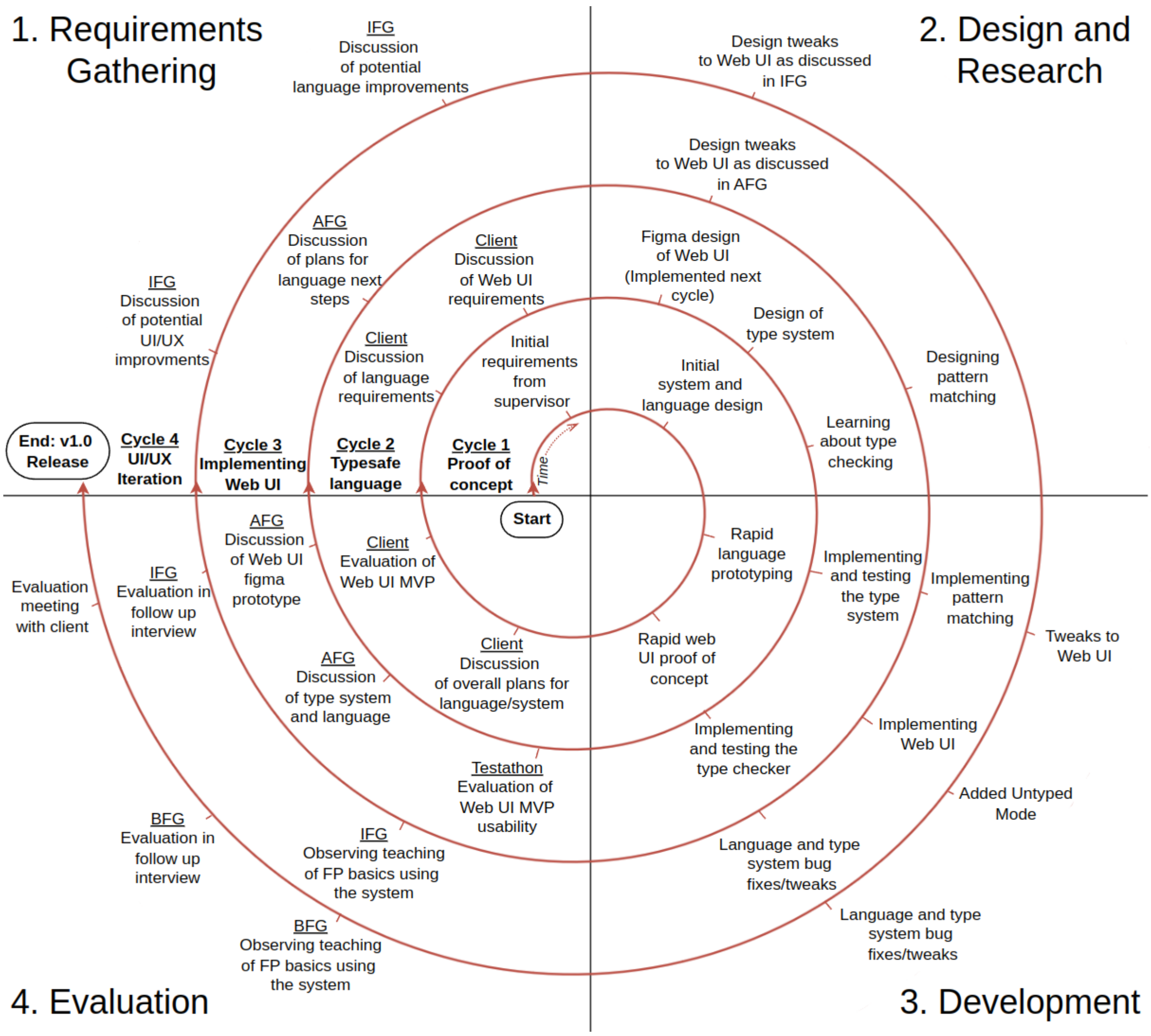


Figure 3.1: Caption

Chapter 4

Cycle 1 - Proof of Concept

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

In our initial meeting, Jess helped me to identify an appropriate client: Samantha Frohlich, a lecturer in the first year Functional Programming module at University of Bristol, who also wanted a tool that fit this brief. It was necessary to identify a client other than Jess, as Jess's existing role as my supervisor/primary marker could limit guidance she could give me as a client.

4.1 Requirements Analysis

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

4.1.2 Autoethnography

> 'Autoethnography is an ethnographic method in which the fieldworker's experience is investigated together with the experience of other observed social actors.'[13]

In this cycle, 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.

4.1.3 The Brief

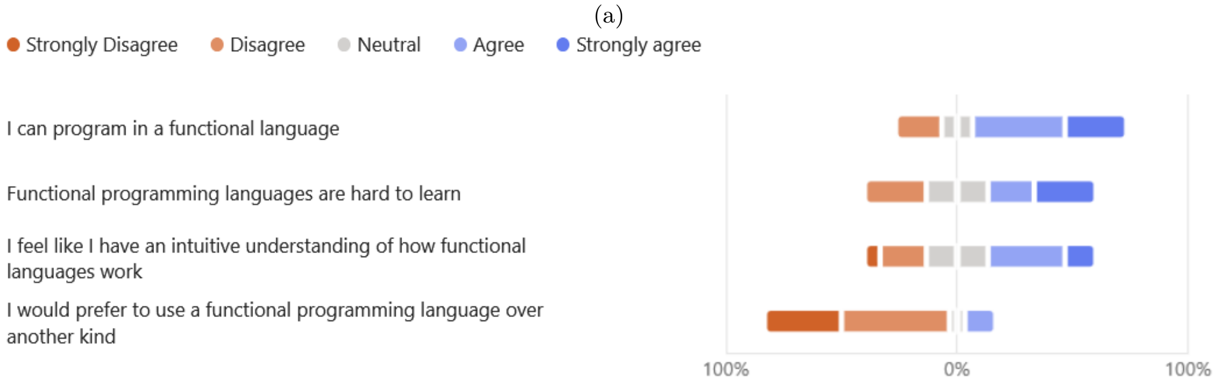
This project was proposed by my supervisor, Jess Foster. She is a lecturer at the University of Bristol, who lectures in the Functional Programming (FP) unit in first year. In our meeting in November 2024, we discussed how she wanted a tool that would help build intuition for how these languages are evaluated, that she could demonstrate in these lectures. We also discussed the benefits of the tool being accessible to students to use themselves during labs or at home.

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. Many students to the first year FP unit do not have any experience with functional programming, and find it difficult to gain an intuition from passively watching lectures.

During the testathon in cycle 2 [REFHERE: Testathon], participants were given a survey to fill in. There were 15 participants, who were a mixture of undergraduate and postgraduate computer scientists, all of whom had taken the first year FP unit. In one section of the survey, they were presented with a series of statements designed to gauge their feelings towards functional and imperative languages. A

(a)



(b)

Figure 4.1: The results of a survey performed during the testathon, where a Likert scale was used to gauge 15 participants feelings towards imperative (a) and functional (b) programming languages

Likert scale[10] was used to measure the attitudes of participants towards the statements. See 4.1a for imperative results, and 4.1b for functional results.

80% of respondents strongly/agreed that they can program in an imperative language, similarly 80% of respondents strongly/agreed that they had an intuitive understanding of them. 66.7% of respondents strongly/agreed that they knew functional languages, but only 47.6% of respondents would strongly/agree that they had an intuitive understanding of them.

The number of people who claim to know how to program in functional languages is less than imperative, but more striking is the difference in reported 'intuition'.

All participants who strongly/agreed/were neutral about functional languages being hard to learn were presented with a free form text box, and asked to describe why. These responses were turned into a word cloud 4.2. Half of responses included the word "different".

In my experience, a very effective way to build intuition for functional programming languages is to demonstrate their evaluation step by step. I found myself doing this on paper for students when trying to explain them. 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.

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 of 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.

Figure 4.2: A word cloud of answers entered into the free form text box presented to respondents who reported that they did not disagree that functional languages were hard to learn

Demonstrating FP languages The tool must contain, at its core, a Haskell-like functional programming language in order to demonstrate Haskell. It could be included/required 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.

4.1.4 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 maximise 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 in a similar way).

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

The features that should be selected for the language are the features that maximise 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 prioritise 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. [TODO: Finish this bit, summarise requirements]

4.2 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.

4.2.1 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): $Id ::= [A..Z][a..zA..Z0..9\_]*$

4.2.2 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. 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

```
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, use ”main” to represent a programs entry point, so we may use main.

We must also add a way to represent values, such as integers and booleans, to our language. 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.

These two features massively shorten and simplify programming in this language.

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

4.2.3 Reduction

As discussed in the background, functional programs progress via reduction [TODO: FINISH]

Values Values in SFL should be

4.3 Implementation

4.3.1 The Abstract Syntax Tree

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

- Identifier
- Literal
- Pair
- Application
- Abstraction
- Match
- Assignment

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

enum ASTNodeType {
    Identifier,
    Literal,
    Pair,
    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 4.3: The Rust code listing for the definition of the AST, with lifetime specifiers, accessibility modifiers, and the syntax information (see 4.3.1) removed for conciseness.

- Module

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 necessary frequently 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 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 4.3 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 pair, the first node is the first in the pair, and the second is the second in the pair.
- If it is a match expression, the first node represents the matched value, then after this it consists of the case followed by the resulting expression. Match expressions will always therefore have an odd number of children.

- 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 `Literal`s and `Identifier`s.
- 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

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 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 [9]. 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 4.4 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.

Methods on the AST

Below are a selection of the more important or interesting methods implemented on the AST and its nodes.

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.

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

enum ASTNodeType<'a> {
    Identifier{name: String},
    Literal{value: String, _type: PrimitiveType},
    Pair{first: &'a ASTNode<'a>, second: &'a ASTNode<'a>},
    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>>},
    Match{expr: &'a ASTNode<'a>, cases: Vec<&'a ASTNode<'a>>}
}

struct ASTNodeSyntaxInfo { ... }

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

Figure 4.4: An alternative implementation with a few advantages over the actual implementation.

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`. [TODO: fix the way that the dollar sign operator works. It can be redefined as an inbuilt as its just an infix op]
- 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.

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. 4.5 shows a subsection of this algorithm, showing how it works for IDs, Literals and Pairs.

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

type Diff = Vec<DiffElem>

function diff(ast1, ast2, expr1, expr2) -> Diff {
    node1, node2 = ast1.get(expr1), ast2.get(expr2)
    diff = new Diff;
    match (node1, node2) {
        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}) {
            diff += Similarity("(")
            diff += diff(ast1, ast2, first1, first2)
            diff += Similarity(",")
            diff += diff(ast1, ast2, second1, second2)
            diff ++ Similarity(")")
        }

        ...
    }

    return diff;
}
```

Figure 4.5: The Rust code 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.

4.3.2 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:

```
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 C 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 4.3.2), 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
  - An opening brace (indicating the end of parsing the matched expression of a match statement)
  - 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: ...
}
```

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 have 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 in4.3.1, 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.

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

4.3.3 Web UI MVP

As part of this section, I also developed the MVP for the Web UI.D.1 shows the web UI after this stage of the project.

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 an 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 (see2.4) that requires the package that results from compiling **libsfl_wasm**

[TODO: More stuff about wasm bindings??]

4.4 Evaluation

At the end of the cycle, 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 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

4.4.1 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:

- It was lacking many important features. The most difficult things to teach are concepts involving more complex data types
- There is a typechecker, but it does not allow polymorphism. Without the typechecker, polymorphism would work, so the typechecker reduces the number of possible programs without providing much benefit. Either there is a good type checker, or there should be no typechecker.

4.4.2 The Existing UI/UX (see D.1)

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 of the redexes at once as they were not aligned with each other.

4.4.3 Effectiveness as a Teaching Tool

- The project is already very useful as a teaching tool to demonstrate:
  - Evaluation order, and the value of laziness
  - Currying
  - The λ-calculus
- **Minimum** The project should be extended to demonstrate:
  - List and common list functions such as 'map' and 'fold'. These do not have to by polymorphic, they could be just defined over Ints.
  - Pattern matching.
  - Non-polymorphic user definable data types
  - A list of previous program states as we progress.
- **Extra** The project could be extended to demonstrate:
  - Syntax highlighting, to make the language easier to read
  - User definable data types, preferably polymorphic. This would mean we could define '$List$' as part of the language which would be good.
  - Recursive Types
  - Polymorphism
  - Type Aliases

Chapter 5

Cycle 2 - Types and Pattern Matching

The aim of this cycle was to develop the more technical aspects of the project. In this cycle, I moved away from the autoethnographic (4.1.2) approach, where most of my requirements came from within, to an externally motivated approach. The requirements for this cycle were motivated by my client meeting (4.4).

5.1 Requirements Analysis

Going into this cycle, we have a proof of concept system and an idea for how the system will look. The client had many valuable insights into what features would be valuable to add. Below is a list of features identified to add to the system in this cycle.

Language

- Pattern Matching
- User definable data types: tagged unions
- Pairs

5.2 Design

5.2.1 Language Changes

Type System

We must have types representing integers and booleans in our language, if we are to effectively represent the type of expression containing their respective literals. We also want our type system to be able to express 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 in a similar manner 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.

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.

We can avoid thinking about kinds by enforcing that a type constructor is always given the correct number of arguments

$$\begin{array}{llcl} \text{Types} & A, B, C & ::= & \text{Int} \mid \text{Bool} \mid \alpha \mid \forall\alpha.\, A \mid A \to B \mid (A, B) \mid \text{Name}[A_1, \ldots, A_n] \\ \text{Monotypes} & \tau, \sigma & ::= & \text{Int} \mid \text{Bool} \mid \alpha \mid \hat{\alpha} \mid A \to B \mid (A, B) \end{array}$$

Figure 5.1: Syntax of types and monotypes. Note that this is the external definition: as seen by the users of SFL. See 5.3 for the extra type system structures required internally for the typechecker

Note that our type constructor application definition above is more permissive than is correct, as it does not enforce correct arity. This can be handled by the parser maintaining the context of the arity of each type. It can then be double checked for debugging purposes via an assertion in the type checker.

Match

To support different execution based on a condition, we must have some structure that can differentiate between values 4.2.3.

Syntax Sugar

5.2.2 UI

I completely redesigned the UI based on the clients feedback, as well as some other factors. See D.1 for the current state. See D.2 and D.3 for screenshots of the new design.

This design was meant to be a work in progress but it looks quite similar to the final release of the product (Screenshots D.7, D.8, D.9 and D.10). Before implementing this design, I discussed this design with the Advanced Focus Group (see 5.4) and they were much more positive about this UI than the existing one

5.3 Implementation

5.3.1 The Parser

We must make some changes to the parser to include these new features.

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:

- Consume the "match" keyword.
- Parse the expression matched over
- Consume an open brace
- While the next token isn't a close brace:
  - Parse a pattern (5.3.1).
  - Consume a right arrow
  - Parse an expression
- Consume a close brace

Following this, a match node is created, where the `children` vector is set appropriately with the pattern and expressions.

Patterns A pattern must be a value 4.2.3; 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
- Identifiers must be either:
  - Unbound lowercase variables
  - Underscore (`_`) representing a wildcard pattern
  - A bound uppercase variable (a constructor)

5.3.2 Types

Rust allows us to represent our types (see 5.3 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 5.2 for the listing.

```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 5.2: The Rust code listing for the definition of types. Existential is separated as it is more of an implementation detail than a part of the type system

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.

Methods on Types

Below are a selection of the more important or interesting methods implemented on Types.

Substitution of type variables 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.

5.3.3 The Type Checker

The type checker will be bidirectional, and will follow and algorithm largely based on the one in [5]. The quote that follows from this paper, describes bidirectional type checking and its merits:

> '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.'[5]

It was the 'relative ease of implementation' that attracted me to bidirectional type checking. After running the algorithm by hand to convince myself the algorithm works on checking id ($\lambda x.x$) against type $\forall a.a \to a$ (photo of whiteboard derivation included in submission [TODO: is this the right way to cite this]), 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. Not including assignment and modules as these are not part of expression syntax). 5.3 shows the type system, including the typechecker implementation details, as well as the unmodified context structure, which keeps track of the state of the typechecker as it progresses recursively through the type system. B shows some example derivations using this algorithm including some of my rules. 5.4 shows the modified algorithm for substituting all of the information in a context into a type.

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

Figure 5.3: Syntax of types, monotypes, and contexts as seen by the typechecker. Note that the existential type variables ($\hat{\alpha}$) can not actually be created by users, they are an implementation detail

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

Figure 5.4: Applying a context, as a substitution, to a type

The full typechecking algorithm is listed here below. Most of these rules are untouched, the ones that I added or modified are highlighted .

5.4 Evaluation

The aims of this cycle were to develop the language as well as some of the other more technical features of this project.

At the end of this cycle, I held a focus group with the following aims:

- Discussing the language with programming language experts.
-

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

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

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

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

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

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

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

Figure 5.5: Algorithmic subtyping. The rules with highlighted names are my additions, the rest are unchanged from [5]

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

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

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

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

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

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

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

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

Figure 5.6: Instantiation. These rules are unmodified from [5]

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ 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

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

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

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

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

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

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

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

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

Figure 5.7: Algorithmic typing. The rules with highlighted names are my additions, the rest are unchanged from [5]. Note that the checking rules IntLit⇐, BoolLit⇐, Pair⇐ 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

Chapter 6

Conclusion

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

[5] 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].

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

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

[8] 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.

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

[10] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[11] 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`.

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

[13] 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`.

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

[15] 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, or any other AI model, 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.

Appendix B

Some Example Derivations Using the Type Checking Algorithm

B.1 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 $\forall \alpha\ \beta.\, \alpha \to \beta \to (\alpha, \beta)$. Type checking/synthesis is done recursively from bottom up, so read [1] upwards.

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

$$\Delta = \alpha$$

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

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

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

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

B.2 Typechecking an Expression Involving Lists

We shall attempt to use the algorithm to check the type of Cons 1 x against *List Int*. This derivation should serve as a demonstration of how more complex checking works. This derivation assumes *Cons* and *Nil* are defined over *Int*s only. 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.

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

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

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

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

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

$$\dfrac{\begin{array}{c}[3]\Gamma \vdash Cons\, 1 \Rightarrow List\ Int \to List\ Int \dashv \Gamma \\ [7]\Gamma \vdash List\ Int \to List\ Int \bullet Nil \mathrel{\Rightarrow\!\!\!\Rightarrow} List\ Int \dashv \Gamma\end{array}}{\Gamma \vdash Cons\, 1\, Nil \Rightarrow List\ Int \dashv \Gamma}\ {\to}\mathsf{E}\,[2]$$

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

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.

7. To synthesise the type of Nil under the application of $List\ Int \rightarrow List\ Int$, 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

Appendix C

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 D

UI Screenshots

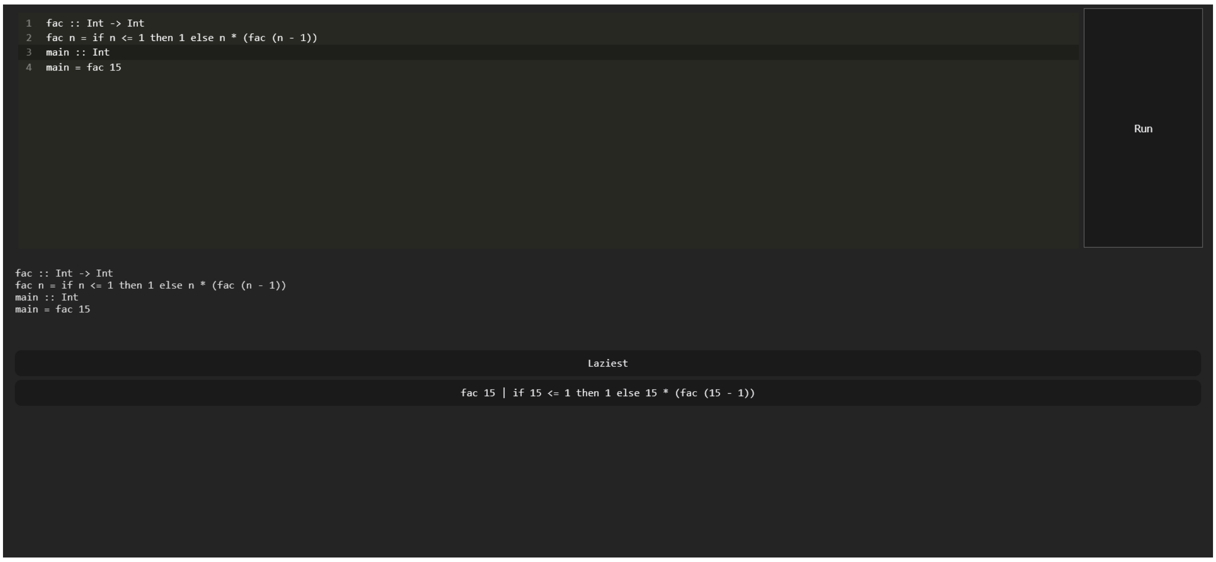


Figure D.1: The Web UI MVP, as presented to my client at the end of cycle 1. Note that this does have type assignments, but these were just ignored by the parser and typechecker at this stage.

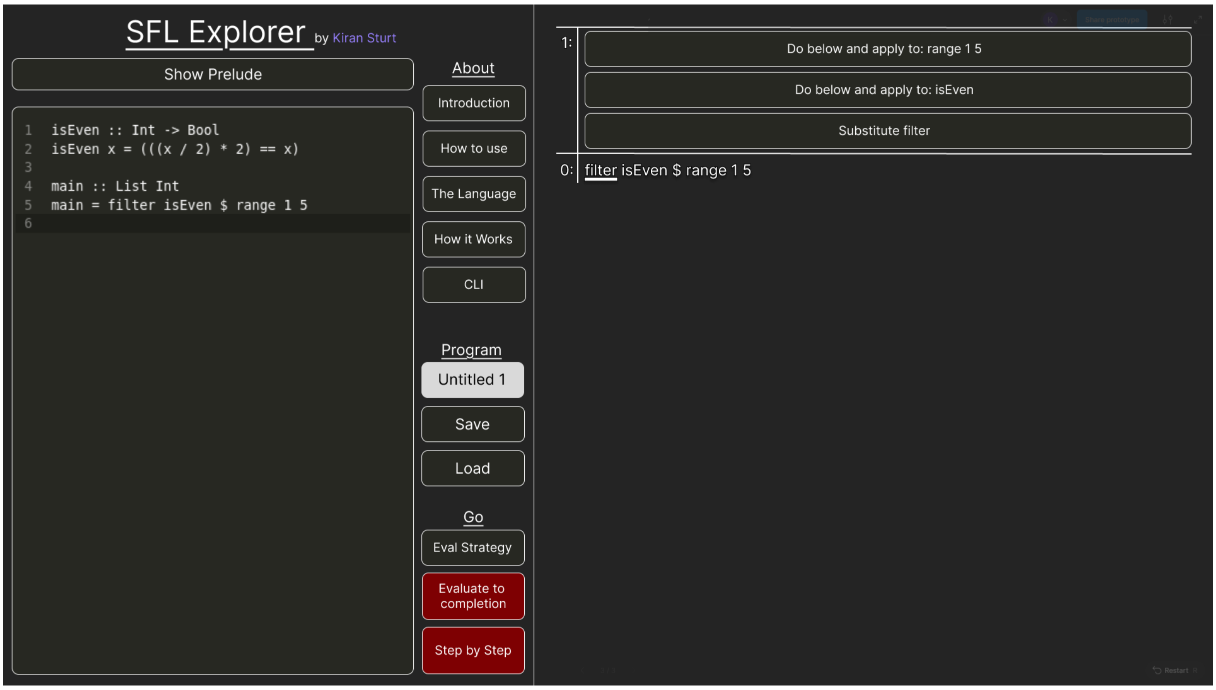


Figure D.2: Screenshot 1 of the Figma design of the web UI

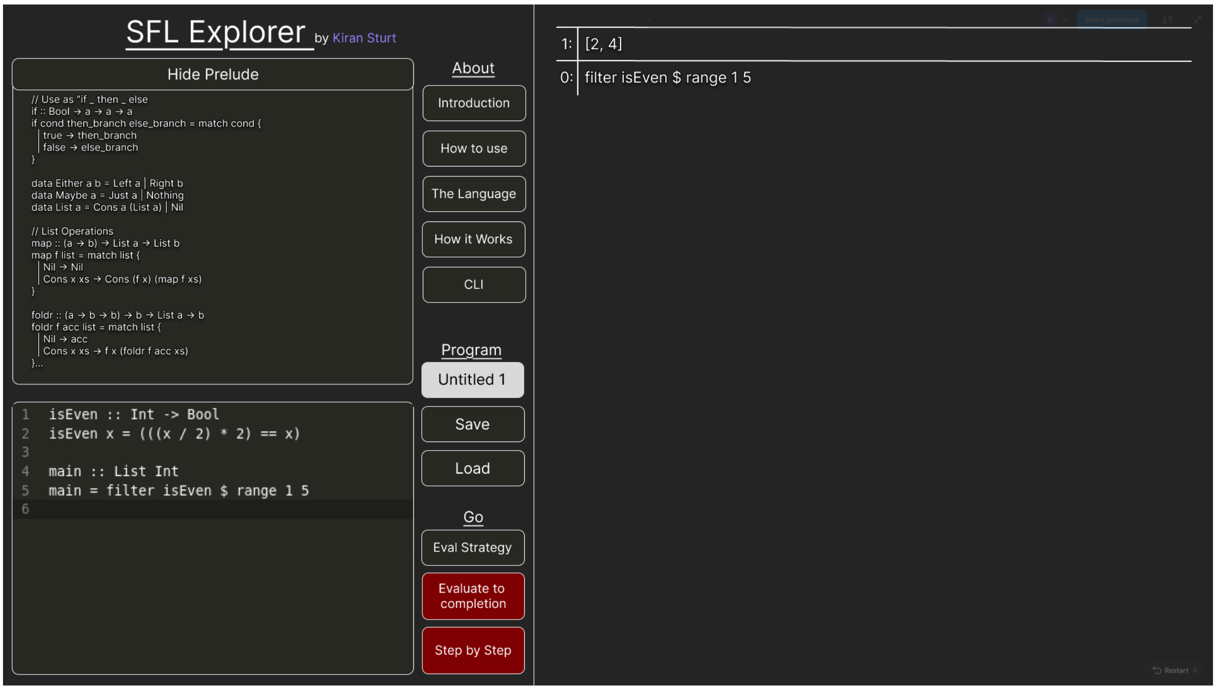


Figure D.3: Screenshot 2 of the Figma design of the web UI. This version shows the prelude extended

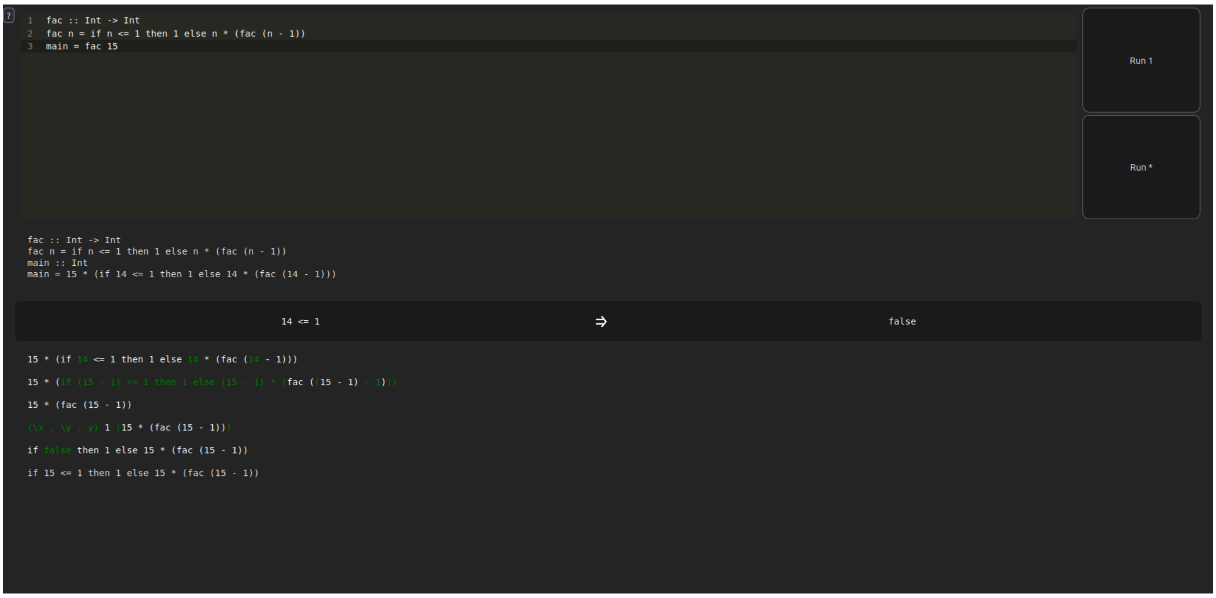


Figure D.4: The UI as tested in the testathon

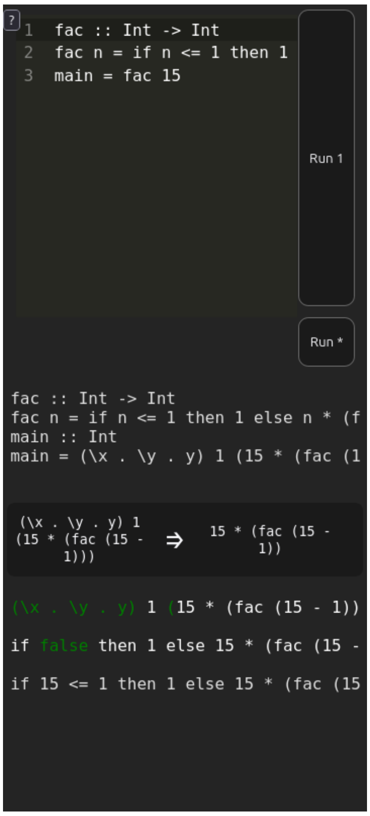
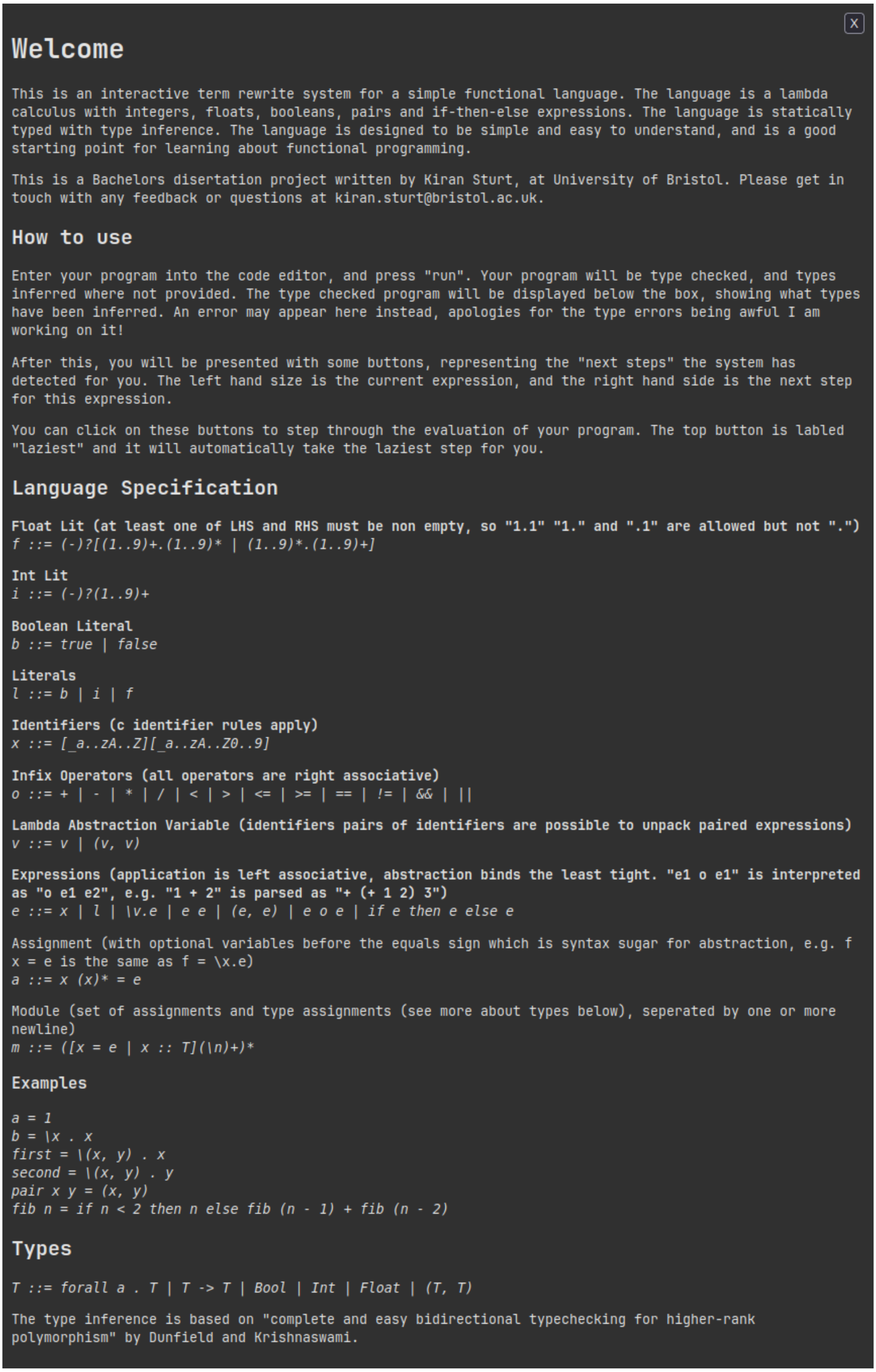


Figure D.5: The UI as tested in the testathon, as it would have appeared on a Samsung Galaxy S20



X

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 D.6: The "Help menu" in the testathon. This was spawned by pressing the "?" button in the top left of the UI, and dismissed by pressing the "X" button, or clicking outside of the box

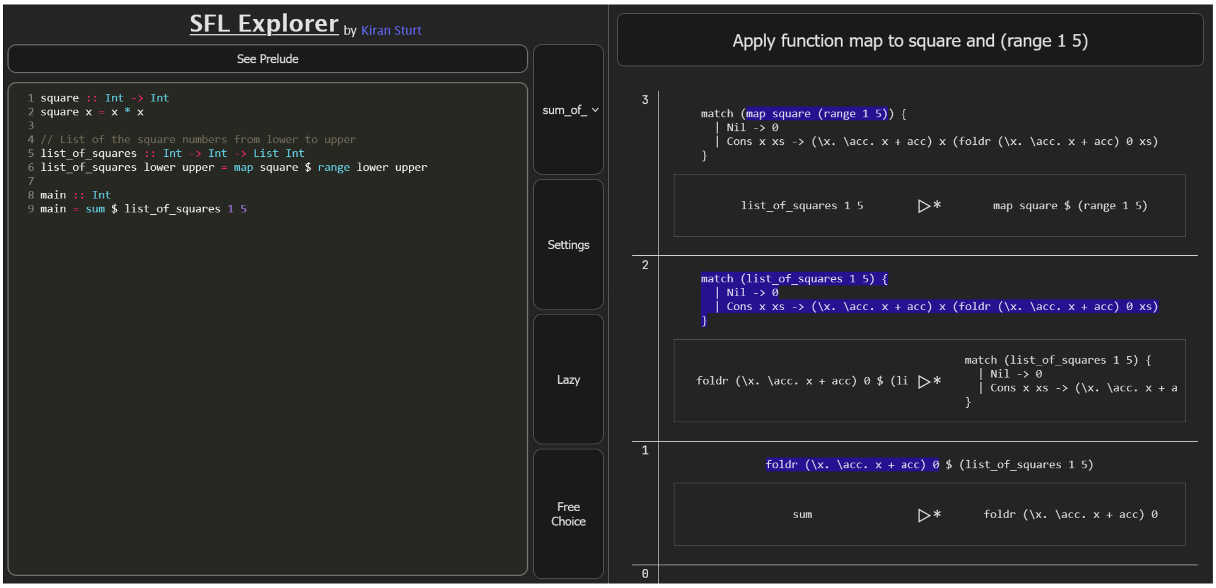


Figure D.7: The final product during lazy evaluation of the ‘sum of squares’ sample program

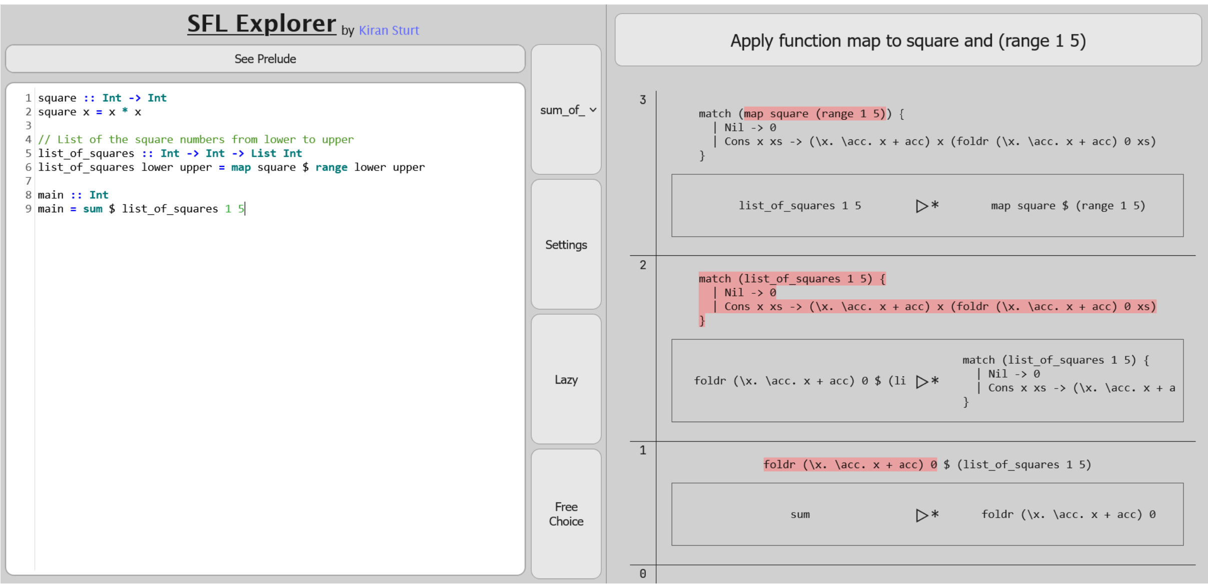


Figure D.8: The final product during lazy evaluation of the ‘sum of squares’ sample program in light mode

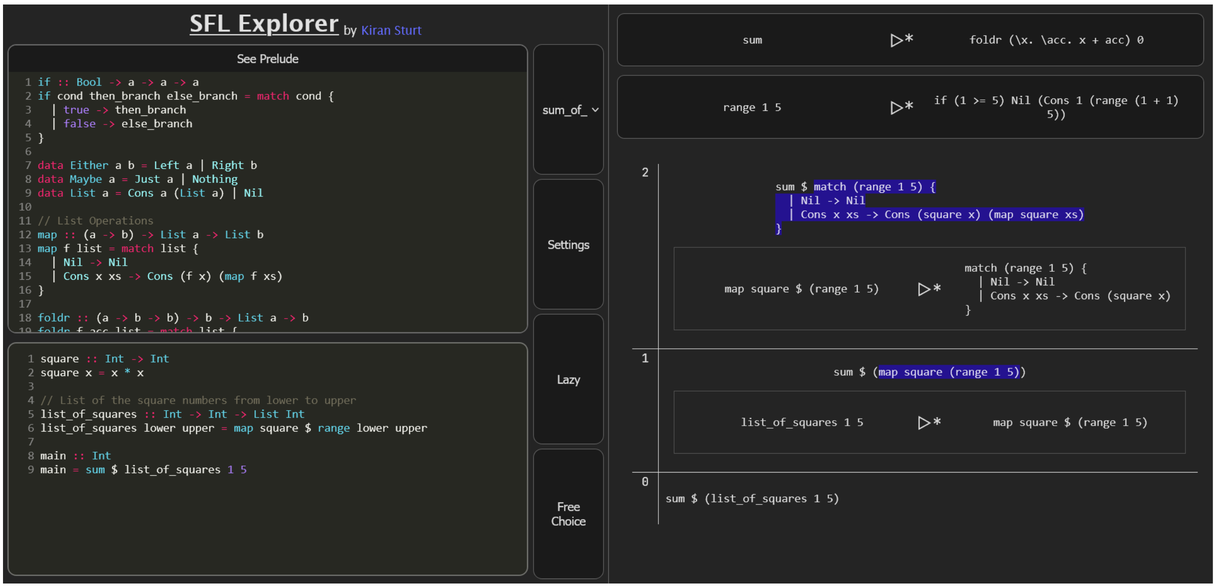


Figure D.9: The final product during free choice evaluation of the ‘sum of squares’ sample program, with the prelude visible

Figure D.10: The final product during free choice evaluation of the ‘sum of squares’ sample program, with the prelude visible in light mode

Appendix E

Language Grammar